

COMPUTER SUPPORTED SOFTWARE INSPECTION

SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE,
UNIVERSITY OF STRATHCLYDE, GLASGOW
FOR
THE DEGREE OF DOCTOR OF PHILOSOPHY.

By
Fraser Macdonald
October 1998



The copyright of this thesis belongs to the author under the terms of the United Kingdom Copyright Acts as qualified by University of Strathclyde Regulation 3.49. Due acknowledgement must always be made of the use of any material contained in, or derived from, this thesis.

© Copyright 1998

Abstract

For over twenty years, software inspection has been advocated as a simple, cost-effective technique for defect detection in all types of documents. Essentially, a number of participants review a document with the aim of discovering defects. Many positive experience reports have been published demonstrating its benefits, and there are now a number of variations on the basic inspection method.

In recent years, there have been a number of attempts to further increase inspection efficiency by the introduction of tool support, resulting in a number of prototype systems. In general, however, existing systems tend to suffer from three major shortcomings. The first concerns their inability to easily support a number of different inspection processes, as well as the inspection of a variety of document types. Existing tools also treat the move to computer-based inspection as a simple change of medium, when there are opportunities to greatly enhance the process. Finally, evaluation of the effectiveness of these tools is sparse.

This thesis describes work tackling these deficiencies. Support for multiple inspection processes was achieved by developing a high-level process description language which can be used as input to an inspection tool. A prototype tool was developed which implements this language and also provides a simple mechanism for supporting inspection of multiple document types. This tool was also used to investigate a number of methods for increasing the efficiency of inspection, including cross-referencing within and between documents, active checklists and automatic collation of defect lists. In addition, two controlled experiments comparing paper-based and tool-based inspection were performed, the first of their kind. These experiments could reveal no performance difference between methods.

Acknowledgements

I am indebted to my supervisors Dr. James Miller, Dr. Marc Roper and Dr. Murray Wood for their invaluable advice and criticism. Without them this thesis would genuinely not have been completed. Thanks also to Dr. Andrew Brooks for his supervision during the early stages of my research.

Thanks to Computer Science System Support personnel past and present – Ian Gordon, Gerry Haran, David Lloyd, Tony Povoas and Gordon Ritchie – who helped with various technical problems.

ASSIST uses with permission a modified version of the Python RPC mechanism developed by Daniel Larsson. Python dialogue code by David Redish is also used. The code implementing the Porter stemming algorithm was based on C and Java implementations made freely available by IDOMENEUS. Some material used in the experiments described in this thesis was originally written by Gary Perlman, Christopher Lott and Eric Kamsties. This material is used with permission.

Finally, some personal acknowledgements. As ever, my parents are a constant source of support and encouragement. Their efforts do not go unnoticed, and are greatly appreciated. Last, but certainly not least, I am also indebted to my girlfriend Fiona for her support and understanding.

The research contained within this thesis was supported by a Graduate Teaching Assistant position with the Department of Computer Science, University of Strathclyde. Travel to conferences was supported by the Department, the Faculty of Science, and Software Research Institute, San Francisco, CA. Their support is gratefully acknowledged.

Publications

The research contained within this thesis has resulted in a number of publications. These are as follows:

- [78] contains an initial review of tools available to support software inspection. This paper was presented by the author at the Seventh International Workshop on Computer Aided Software Engineering, July 1995.
- [80] is an expanded version of the above review paper, including a framework for research in software inspection support tools.
- [73] describes the initial version of the process definition language (IPDL) described in this thesis. This paper was presented by the author at the Tenth International Software Quality Week, May 1997.
- [74] describes the final version of IPDL and an initial version of ASSIST, the prototype tool used to implement this research.
- [72] briefly describes IPDL and some of the features of the second version of ASSIST. This paper was presented by one of my supervisors, Dr. James Miller, at the First International Software Quality Week Europe, November 1997.
- [77] reports on the first experiment comparing tool-based and paper-based software inspection.
- [79] explores the application of inspection to object-oriented code, suggesting possible areas of exploration with regard to tool support.
- [88] discusses the use of metrics to manage the inspection process, and how they are implemented in ASSIST

A number of papers are currently under review:

- [76] is a review of all inspection support tools available at the time of completion of this research.
- [75] is an extended overview of the second version of ASSIST.
- [89] describes both experiments comparing tool-based and paper-based inspection, comparing the results.

A number of technical reports were also produced:

- [71] describes the initial version of IPDL and the inspection processes on which it is based.
- [69] is the manual for the first version of ASSIST.
- [70] is the manual for the second version of ASSIST.

Contents

1	Introduction	1
1.1	Background	1
1.2	Contributions of Thesis	4
1.3	Thesis Outline	4
2	A Review of Major Inspection Processes	6
2.1	Fagan Inspection	6
2.2	Structured Walkthroughs	8
2.3	Humphrey's Inspection Process	10
2.4	Gilb and Graham Inspection	11
2.5	Asynchronous Inspection	14
2.6	Active Design Reviews	17
2.7	Phased Inspection	18
2.8	N-Fold Inspection	19
2.9	Conclusions	21
3	Existing Tool Support for Software Inspection	22
3.1	Tool Support for Paper-based Inspection	22
3.1.1	COMPAS	22
3.1.2	Quality Group 4000	23
3.1.3	Inspection Process Assistant	24
3.1.4	Comparison of Tools to Support Paper-based Inspection	25
3.2	On-line Inspection Tools	25
3.2.1	ICICLE	27
3.2.2	InspeQ	30
3.2.3	Scrutiny	31
3.2.4	CSI	34

3.2.5	CAIS	35
3.2.6	AISA	36
3.2.7	Notes Inspector	37
3.2.8	CSRS	37
3.2.9	TAMMi	40
3.2.10	InspectA	41
3.2.11	hyperCode	42
3.2.12	WiP	42
3.2.13	Distributed Code Inspection	43
3.2.14	Comparison of On-line Tools	44
3.3	Research Framework	47
4	Supporting a Generic Software Inspection Process	49
4.1	Inspection Process Definition Language	50
4.1.1	Implementation Technologies Considered	50
4.1.2	Derivation of Generic Process	57
4.1.3	IPDL Definition	63
4.1.4	IPDL Example - Fagan Inspection	73
4.1.5	Conclusions	75
4.2	Introduction to ASSIST	75
4.2.1	Using ASSIST to Execute the Process	76
4.2.2	Inspection Facilities	77
5	Evaluation of Basic Tool Support	80
5.1	IPDL Evaluation	80
5.1.1	Scrutiny	80
5.1.2	CSRS	82
5.1.3	Conclusions	84
5.2	Comparing Basic Tool-based and Paper-based Software Inspection	85
5.2.1	Evaluations of Existing Inspection Support Tools	86
5.2.2	Experiment Design	87
5.2.3	Results and Analysis	92
5.2.4	Conclusions	105
6	Enhancing the Software Inspection Process	107
6.1	Automatic Cross-referencing	108

6.2	Active Checklists	110
6.3	A C++ Inspection Environment	113
6.4	Automatic Defect List Collation	117
6.5	Conclusions	120
7	Evaluation of Enhanced Tool Support	122
7.1	Comparing Enhanced Tool-based and Paper-based Software Inspection	122
7.1.1	Introduction	122
7.1.2	Experiment Design	123
7.1.3	Results and Analysis	124
7.1.4	Conclusions	137
7.2	Automatic Defect List Collation	138
7.2.1	Introduction	138
7.2.2	Method	138
7.2.3	Results	139
7.2.4	Conclusions	146
8	Conclusions	148
8.1	Summary	148
8.2	Contributions and Results	149
8.3	Further Work	151
8.4	Concluding Remarks	152
	Bibliography	154
A	Future Directions in Computer Supported Software Inspection	165
A.1	Applying Inspection to Object-Oriented Code	165
A.2	Data Collection and Analysis	169
A.2.1	Process Measurement	169
A.2.2	General Process Feedback	171
A.2.3	Checklist Formation and Improvement	173
A.2.4	Estimating Defects Remaining	174
A.2.5	Inspector Experience and Behaviour	175
A.3	Conclusions	175

B ASSIST V2.1 User Manual	177
B.1 Introduction	177
B.1.1 Requirements	177
B.1.2 Installation	177
B.1.3 Starting the Server	179
B.1.4 Starting the Client	179
B.2 Preparing for an Inspection	181
B.2.1 Introduction	181
B.2.2 The Document Database	182
B.2.3 The Personnel Database	184
B.2.4 The Process Database	186
B.2.5 Starting a New Inspection	189
B.3 Executing An Inspection	190
B.3.1 The Execute Window	191
B.3.2 Document Browsers	192
B.3.3 Process Phases	207
B.4 Writing IPDL Processes	211
B.4.1 Process Outline	211
B.4.2 Declarations	212
B.4.3 Process Definition	216
B.4.4 Putting It All Together	221
B.5 IPDL Reference	222
B.5.1 Structure of Process Description	222
B.5.2 Inspection Document, Participant and Responsibility Declarations	223
B.5.3 The Organisation Process	226
B.5.4 The Detection Process	227
B.5.5 The Completion Process	230
B.6 Formats	232
B.6.1 Checklists, Criteria, Reports and Plans	232
B.6.2 Help Documents	235
B.7 Metrics Available in ASSIST	237
B.8 Customising and Extending ASSIST	237
B.8.1 Altering the Printer Setup	237
B.8.2 The .assistrc file	237
B.8.3 Adding New Browsers	238

B.8.4	Adding New Classification Schemes	244
C	IPDL Processes	245
C.1	Fagan Inspection	245
C.2	Structured Walkthrough	247
C.3	Humphrey Inspection Process	249
C.4	Gilb and Graham	252
C.5	Asynchronous Inspection	254
C.6	Active Design Reviews	258
C.7	Phased Inspection	260
C.8	N-Fold Inspection	263
D	Experiment Materials	268
D.1	Timetable	268
D.2	C++ Checklist	269
D.3	Individual Defect Report Form	272
D.4	Master Defect Report Form	276
D.5	Training Program <code>count.cc</code>	279
D.5.1	Specification for program <code>count.cc</code>	279
D.5.2	Library functions used in <code>count.cc</code>	281
D.5.3	<code>count.cc</code>	281
D.5.4	Defects in <code>count.cc</code>	282
D.6	Training Program <code>tokens.cc</code>	283
D.6.1	Specification for program <code>tokens.cc</code>	283
D.6.2	Library functions used in <code>tokens.cc</code>	285
D.6.3	<code>tokens.cc</code>	286
D.6.4	Defects in <code>tokens.cc</code>	289
D.7	Training Program <code>simple_sort.cc</code>	290
D.7.1	Specification for program <code>simple_sort.cc</code>	290
D.7.2	<code>simple_sort.cc</code>	292
D.7.3	Defects in <code>simple_sort.cc</code>	292
D.8	Training Program <code>series.cc</code>	293
D.8.1	Specification for program <code>series.cc</code>	293
D.8.2	Library functions used in <code>series.cc</code>	295
D.8.3	<code>series.cc</code>	295
D.8.4	Defects in program <code>series.cc</code>	298

D.9	Experiment Program analyse.cc	300
D.9.1	Specification for program analyse.cc	300
D.9.2	Library functions used in analyse.cc	303
D.9.3	analyse.cc	303
D.9.4	Defects in program analyse.cc	307
D.10	Experiment Program graph.cc	308
D.10.1	Specification for program graph.cc	308
D.10.2	Library functions used in graph.cc	311
D.10.3	graph.cc	311
D.10.4	Defects in program graph.cc	314
D.11	Questionnaires	316
D.11.1	Questionnaire 1	316
D.11.2	Questionnaire 2	319
D.11.3	Questionnaire 3	322
D.11.4	Questionnaire 4	324
E	Raw Data	328
E.1	Comparing Paper-based and Tool-based Software Inspection	328
E.1.1	Experiment 1	328
E.1.2	Experiment 2	334
E.2	Automated Defect List Collation	339
E.2.1	Experiment 1	339
E.2.2	Experiment 2	342

Chapter 1

Introduction

1.1 Background

The chequered history of software engineering is well-documented. Spectacular failures, such as the Ariane 5 launch [107] still occur. Systems which are late and over-budget, such as the UK's new air traffic control system [45], are commonplace. Unfortunately, even the most carefully written software contains defects. Finding these defects is difficult, time-consuming and therefore expensive.

Despite over 30 years of research, software engineering techniques which receive universal acclaim and acceptance are the exception rather than the rule. Software inspection is such a technique. Originally described by Michael Fagan over twenty years ago [37], it has become well-known as an effective defect finding method. The basic technique is simple: a number of participants review a document with the aim of discovering defects. The approach is effective because of the involvement of people other than the author of the document. These participants are not intimately familiar with the document, hence they tend to find more defects. Inspection can be used on any type of document, including specifications, designs, code and test plans.

The original inspection process defined by Fagan employs four to six people, each with specific roles. The *moderator* is the person in overall charge of the inspection. During the inspection meeting, a *reader* is required to paraphrase the document and a *recorder* notes all defects found along with their classification and severity. The *author* of the document under inspection is another team member. Any remaining participants are cast as inspectors, whose only duty is to look for defects in the document. The process used consists of five phases. During *overview* the author presents the document undergoing inspection to the rest of the

team. Each team member then carries out *individual preparation*, consisting of studying the document to gain an understanding of it. Checklists of common defect types can be used to aid inspectors. An *inspection meeting* is then held. The reader paraphrases the document, while inspectors raise any issues they have discovered. The team then discuss the issue until a consensus is reached. If an issue is agreed to be a defect, it is classified and noted by the recorder. No attempt is made to find a solution to the defect; this is carried out later. After the meeting, the moderator writes a report detailing the inspection and all defects found, which is passed to the author. During *rework*, the author carries out modifications to correct defects detailed in the moderator's report. The moderator then performs a *follow-up* phase, ensuring that all required alterations have been made.

There are now many variations of the basic inspection method. For example, Active Design Reviews [95] are intended to associate responsibilities with each reviewer. N-Fold inspections [82] aim to increase inspection effectiveness by replicating the inspection a number of times using independent teams. Formal Technical Asynchronous review method (FTArm) [117] is an asynchronous review method which removes the need to have group meetings. The variation used will depend on the document being inspected, past experience of inspection, team preference, criticality of inspection, and so on.

The benefits of inspection are generally accepted, with success stories regularly published. In addition to Fagan's papers describing his experiences [37, 38], there are many other favourable reports. For example, Doolan [31] reports industrial experience indicating a 30 times return on investment for every hour devoted to inspection of software requirement specifications. Russell [103] reports a similar return of 33 hours of maintenance saved for every hour of inspection invested. Gilb and Graham [41] present many positive experience reports. The benefits of inspection are a direct consequence of its ability to be applied early in the software development lifecycle. The longer defects remain in the system, the more expensive they are to remove: the cost of removing a defect when the system is operational is up to 1000 times the cost of removal during the requirements stage [41]. Inspections are cited as one of nine best practices for software management [14] and appear at Level 3 of the Capability Maturity Model [50].

The manner in which inspections are performed has remained essentially unchanged over its lifetime. Inspection is a low-tech, paper-based technique, and as such has met resistance [103]. Some feel that such a simple approach has no place in today's advanced development environments. Inspection is also labour intensive, requiring the simultaneous participation of three or more people. As a consequence, recent research has started to explore the application

of tool support to software inspection. By automating some parts of the process and providing computer support for others, the inspection process has the capability of being made more effective and efficient, thus potentially providing even greater benefits than are otherwise achieved.

There are many possible benefits from moving to a computer-supported inspection. One desirable attribute of an inspection is rigour. The process must be rigorously followed to ensure repeatability, which is essential if feedback from the process is to be used to improve it, as advocated by Gilb and Graham [41]. Rigour is also important to ensure the inspection is as effective as possible. At the same time, some descriptions of inspection can be ambiguous or misleading. It therefore becomes difficult to enforce the proper inspection process, since the interpretation of guidelines will differ between individuals. Using computers to support the process can help provide this rigour. Most documents are produced electronically, hence on-line inspection is a natural consequence. If the support tool is integrated with the version control system in use, the most up-to-date version of the document is then automatically available. Inspecting an electronic version of the document allows annotation of the appropriate part of the document, instead of producing a completely separate defect list. Annotations are stored on-line and can easily be shared amongst participants. Electronic versions of documents can be presented in ways to enhance the inspection, assisting inspectors with the defect finding task. Computer support can also reduce the need for the face-to-face group meeting, which is expensive to set up and run. Instead the meeting could be held in a distributed and/or asynchronous fashion. Finally, computer support allows metrics from the inspection to be automatically gathered for analysis. This is more accurate than manual capture and allows more finely-grained data to be gathered.

A number of prototype tools have been developed to support software inspection, and are reviewed in detail in Chapter 3. Although existing systems present innovative approaches to supporting software inspection, in general they suffer from several shortcomings. Primarily, they support only a single, usually proprietary, inspection process. They also only support inspection of a single document type, while today's software development environments produce a number of different document types, from plain text to PostScript and other graphical formats. Support for inspection of all of these formats is desirable. Existing inspection tools tend to treat the defect detection process as a simple change of medium, assuming that inspectors will use the same process for finding defects in an on-line document as when inspecting a paper copy. Moving to a computer supported inspection, however, gives an opportunity to provide more active support for finding defects. Finally, no comprehensive evaluations of

these tools have been carried out to determine their effectiveness in comparison with traditional paper-based inspection. This issue must be addressed if tool-supported inspection is to become an accepted alternative to, or even replace, paper-based inspection.

1.2 Contributions of Thesis

The work presented in this thesis is intended to address the above deficiencies. In particular, it makes the following contributions:

- A high-level inspection process description language which can be used as input to a supporting tool to allow the support of any inspection process, or simply as a means to unambiguously communicate inspection processes.
- A prototype inspection support tool which implements this language and also allows support for any type of document.
- An investigation of several facilities which could increase inspection efficiency and make on-line inspection easier, including a method for automatically collating multiple defect lists into a single list, removing duplicates.
- The first reported controlled experiments comparing paper-based and tool-based software inspection.

1.3 Thesis Outline

The remainder of this thesis takes the following form:

Chapter 2: A Review of Major Inspection Processes

The body of the thesis begins with a review of eight of the most common inspection processes described in the literature, introducing basic concepts and terminology in inspection.

Chapter 3: Existing Tool Support for Software Inspection

A review and comparison of a number of tools currently available to support software inspection is presented in this chapter. A number of weaknesses in existing tools were identified from this review, which suggested the main areas of research which should be undertaken.

Chapter 4: Supporting a Generic Software Inspection Process

This chapter introduces IPDL, a language designed to allow easy definition of inspection processes. The first version of ASSIST (Asynchronous/Synchronous Software Inspection Support Tool) is also described, a prototype tool which implements IPDL and provides a means to compare basic tool-based inspection and paper-based inspection.

Chapter 5: Evaluation of Basic Tool Support

Two evaluations of the work presented in Chapter 4 are discussed. The first compares IPDL with other attempts at providing process-independent inspection tool support. The second is a controlled experiment comparing paper-based and tool-based inspection. This experiment shows there is no significant difference between methods, and provides feedback on the usability of the tool.

Chapter 6: Enhancing the Software Inspection Process

Several techniques for improving the efficiency of software inspection are presented. They include an automatic cross-referencing system, active checklists and automatic collation of defect lists.

Chapter 7: Evaluation of Enhanced Tool Support

A second controlled experiment, this time comparing enhanced tool-based and paper-based software inspection, is described. An investigation into the effectiveness of the auto-collation mechanism described in Chapter 6 is also presented.

Chapter 8: Conclusions

The final chapter summarises the content and contribution of this thesis, considers further work related to this research, and presents some conclusions.

Chapter 2

A Review of Major Inspection Processes

As a prelude to describing existing tools for supporting software inspection, this chapter reviews the most important inspection methods described in the inspection literature. Several are well-known and well-used, while others are less well-known, but provide important concepts and ideas on an effective inspection process. For each method, background and a description of the method are provided, along with a summary of the process. Except where absolutely necessary, the description is limited to the details provided in the original article. Obvious gaps which occur in the descriptions of some methods have not been filled, and are noted. The terminology used to describe each process is that used in the original description.

2.1 Fagan Inspection

The original inspection process was defined by Michael E. Fagan in 1976 [37], with an update published ten years later [38]. A Fagan inspection team consists of four to six people, with each person having a well-defined role in the inspection. The *moderator* is the person in overall charge of the inspection. It is the moderator's task to invite suitable people to join the inspection team, distribute source materials and to organise and moderate the inspection meeting itself. The inspection requires the presence of the *author* of the product under inspection. The author can give invaluable help to the inspectors by answering questions pertaining to the intent of the document. Any remaining team members are cast as inspectors. Generally, their only duty is to look for defects in the document. However, at the inspection meeting, two inspectors are given special roles. The *reader* paraphrases the document out loud. The

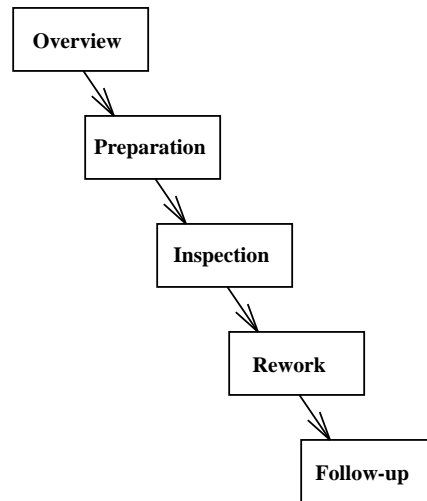


Figure 2.1: The original inspection process defined by Michael Fagan.

recorder is tasked with noting all defects found, along with their classification and severity. Although Fagan indicates that this task is accomplished by the moderator, another member of the team is usually chosen, since the workload involved can be quite high, though mainly secretarial.

Fagan describes five stages in the inspection process, depicted in Figure 2.1. The inspection begins with an *overview*, involving the entire team. The author describes the general area of work then gives a detailed presentation on the document produced. This is followed by distribution of the document itself, and any necessary related work, to all team members. Each team member then carries out individual *preparation*, consisting of studying the document to gain an understanding of it. Although defect detection is not an explicit objective here, some defects will be found. Checklists may be used to help inspectors focus their effort. The next stage is the *inspection meeting*, involving all team members, where defect detection occurs. The reader paraphrases the document, covering all areas. During this process inspectors can stop the reader and raise any issues they have discovered, either in preparation or at the meeting itself. The team then discuss the issue until agreement is reached. If an issue is agreed to be a defect, it is classified as *missing*, *wrong* or *extra*. Its severity is also classified (*major* or *minor*). At this point the meeting moves on. No attempt is made to find a solution to the defect; this is carried out later. After the meeting, the moderator writes a report detailing the inspection and all defects found. This report is then passed to the author for *rework*, where the author carries out modifications to correct the defects found in the document. A *follow-up* phase then occurs, where the moderator ensures that all required alterations have been made. The moderator then decides whether the document should be reinspected, either partially or

Phase	Timing	Participants	Documents used	Documents produced
Overview	S	Moderator Author	Product Sources	
Preparation	A	Inspector Moderator Author	Product Sources Checklists	Individual defect logs
Inspection	S	Inspector Moderator Author Inspector	Individual defect logs Product Sources Checklists	Master defect log Inspection report
Rework	-	Author	Product Sources Master defect log	
Follow-up	-	Moderator	Product Master defect log	Follow-up report

Table 2.1: Summary of Fagan inspection phases and the possible timings, participants, resources and products of each phase. Timing is either synchronous (S) or asynchronous (A). Documents used indicates documents which are usually made available during the phase. Documents produced indicates those documents which are created during the phase.

fully. Although not explicitly stated by Fagan, it is assumed this verdict is presented in a report. It is also unclear whether partial or full reinspection is a continuation of this inspection, or whether a new inspection is convened on the same document. The latter is assumed.

The Fagan inspection process is summarised in Table 2.1. For each phase the table lists the phase timing (where appropriate), the participants, the documents made available and the documents produced. “Product” refers to the document undergoing inspection, while “sources” indicates the documents used when creating the product. For example, a low-level design document may be the source document for a section of code.

2.2 Structured Walkthroughs

Another popular method is Yourdon's Structured Walkthrough [124], which has aims similar to those of inspection, but tends to be less formal and less rigorous. Yourdon defines seven possible participant roles. The *coordinator* is the person tasked with planning and organising the walkthrough, and also takes the role of moderator during the walkthrough meeting. The role of the *scribe* is to take notes on the walkthrough, including any defects found and suggestions made. The *presenter* is tasked with introducing the product and is usually the author

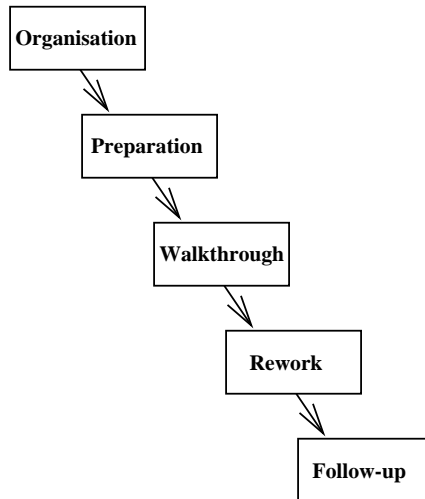


Figure 2.2: The Structured Walkthrough process presented by Yourdon.

of the product. The role of presenter is optional. There are also a number of *reviewers*, whose task is to find defects in the product. The remaining three roles are *maintenance oracle* (who is concerned with future maintenance of the project), the *standards bearer* (whose remit concerns adherence to standards) and the *user representative* (whose task is to ensure the product meets the user's needs). Although Yourdon describes these as separate roles, it can be seen that they are simply reviewers with special responsibilities.

The Structured Walkthrough process is shown in Figure 2.2. The first phase is *organisation* which begins with the producer requesting a walkthrough. The producer supplies the appropriate documentation to the coordinator, who then distributes it to all participants. The coordinator also arranges a time and place for the walkthrough and contacts all participants to confirm the arrangements. The participants now spend time preparing for the walkthrough by reviewing the product. During this stage the producer should be available to answer questions and help participants familiarise themselves with the document. Although this preparation phase is not explicitly described by Yourdon as a separate phase, it shall be treated as such. The *walkthrough* itself begins with the presenter providing an overview of the product. The length of this overview will depend on the familiarity of the participants with the document. The product is then presented in its entirety and reviewers have the opportunity to make comments. Comments from the preparation phase which require no explanation can be passed straight to the producer and the scribe. As reviewers present other comments, the producer may ask for clarification, but should not spend time arguing about the validity of the comment. As with other review methods, there should be no discussion on how each defect may

Phase	Timing	Participants	Documents used	Documents produced
Organisation	-	Coordinator Producer		
Preparation	A	Coordinator Producer Reviewer	Product	Individual lists
Walkthrough	S	Coordinator Producer Reviewer	Product Individual lists	Master list
Rework	-	Coordinator Producer	Product Master list	Summary
Follow-up	-	Coordinator	Product	

Table 2.2: Summary of the Structured Walkthrough phases and the possible timings, participants, resources and products of each phase. Timing is either synchronous (S) or asynchronous (A). The documents used and produced by each phase are also listed.

be corrected. The walkthrough phase should last between thirty and sixty minutes and finishes with a vote on the status of the product. After the walkthrough, the coordinator prepares a management summary and a list of detailed comments. These comments are distributed to all participants. The producer then makes the required alterations to the product during the *rework* phase, deciding on the validity of each comment and seeking guidance from the other participants as appropriate. Finally, a *follow-up* phase occurs to ensure that the required changes have been made to the product. The phases, participants and documents present during a structured walkthrough are summarised in Table 2.2.

2.3 Humphrey's Inspection Process

The inspection process described by Humphrey [50] is very similar to that described by Fagan; however, there are some major differences. The inspection team consists of a number of people with the expected roles, although they are named *moderator*, *producer* and *reviewer*. The phases described are virtually identical in name to those described by Fagan, but the actual process is different.

The process is depicted in Figure 2.3. The *planning* stage allows for selection of participants and preparation of entry criteria. The *overview* stage is identical to that of Fagan. It is during the *preparation* stage that the first deviation from Fagan's process occurs. Here, reviewers are asked to find and log defects, unlike Fagan's method where defect detection is deferred until the meeting. These defect logs are then passed to the producer for what could

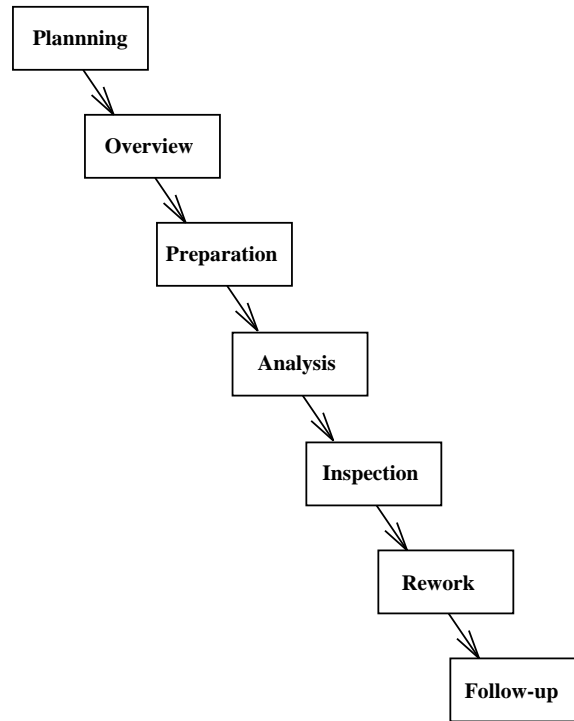


Figure 2.3: The inspection process described by Watts Humphrey.

be termed the *analysis* phase, where the individual logs are analysed and consolidated into a single defect list. At the inspection meeting itself, the producer addresses each defect and can ask reviewers to clarify the meaning of each defect. A list of agreed defects is then produced. The meeting is followed by the typical post-inspection activities of *rework* and *follow-up*. This inspection process is summarised in Table 2.3, which describes the possible timing of each phase and the documents required and produced.

2.4 Gilb and Graham Inspection

One of the most comprehensive texts on software inspections is that of Gilb and Graham [41]. The method they describe is obviously based on Fagan's work; however, it also incorporates other lessons. One such lesson is the defect prevention process described by Jones [59]. This discussion is limited to the inspection itself.

There are three defined roles in this type of inspection. The *leader* is in overall charge of the process and is tasked with planning and running the inspection. The *author* of the document is a required participant. As well as attending the logging meeting, the author should also take part in checking. The remaining team members are *checkers*, whose duty is

Phase	Timing	Participants	Documents used	Documents produced
Planning	-	Moderator Producer		Inspection objectives Participants list
Overview	S	Moderator Producer Reviewer		
Preparation	A	Reviewer	Product Checklists Standards	Defect logs Preparation report
Analysis	-	Producer	Defect logs	Consolidated log
Inspection	S	Moderator Producer Reviewer	Consolidated log	Master defect log Inspection report Inspection summary
Rework	-	Producer	Product Master defect log	
Follow-up	-	Moderator Producer	Product Master defect log	

Table 2.3: Summary of Humphrey inspection phases and the possible timings, participants, resources and products of each phase. Timing is either synchronous (S) or asynchronous (A). The documents used and produced during each phase are also listed.

simply to find and report defects in the document. During the logging meeting, one of the checkers is assigned the role of *scribe* and logs the issues found during the inspection.

The process is illustrated in Figure 2.4. It begins with ensuring that some entry criteria are satisfied. This ensures that the inspection is not wasted on a document which is fundamentally flawed. This is followed by inspection *planning*, where the leader determines inspection participants and schedules the meeting. This phase produces a master plan for the entire inspection. The next phase is *kickoff*, where the relevant documents are distributed and the inspectors briefed. Participants are assigned roles and goals are set. Such goals include checking rates to be met and expected defect rates. The next phase, *checking*, is where each checker works alone to discover defects in the document. These potential defects are recorded for presentation in the next phase, the *logging meeting*. The logging session is a highly structured meeting where potential defects (“issues”) found by the checkers are collected. The emphasis here is on logging as many issues as possible and to this end the meeting is moderated by the inspection leader, who ensures that discussion is kept focused and criticisms are minimised. In addition to defects found during checking, other potential defects may be found at the meeting itself. The meeting can be followed by a *brainstorming* session to record process improvement suggestions. After all potential defects have been logged, the author takes

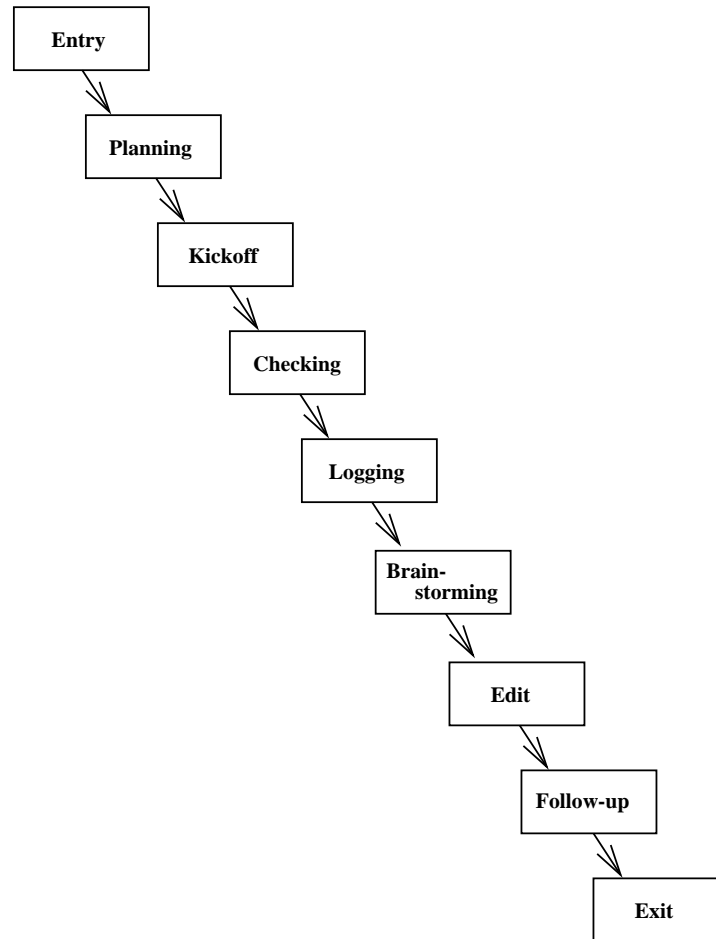


Figure 2.4: The inspection process as described by Gilb and Graham.

the issue list and performs an *edit* on the product. At this point, the issues are also classed as defects. A *follow-up* phase then occurs where the leader ensures that the edit phase has been properly executed. Finally, some *exit* criteria must be satisfied before the inspection can be declared complete. These criteria typically consist of such items as checking rates, which must be within certain limits, and predicted number of defects left in the document.

From the above description it can be seen that the fundamental difference between this process and that of Fagan is the stage where defect detection is carried out, i.e. during an individual phase rather than in a group phase. The process is similar to Humphrey's, but the similarity is not complete, since the producer is not expected to analyse defect logs before the meeting. Instead, each checker simply presents defects when they are reached in the document. Process improvement is also not an explicit feature of Humphrey's process, nor are the entry and exit phases. A summary of Gilb and Graham's process is given in Table 2.4.

Phase	Timing	Participants	Documents used	Documents produced
Entry	-	Leader	Entry criteria	
Planning	-	Leader		Master plan
Kickoff	S	Leader Author Checker		Goals
Checking	A	Leader Author Checker	Product Sources Standards Checklists Procedures Master plan	Issue lists
Logging	S	Leader Author Checker	Product Sources Standards Checklists Procedures Master plan Issue lists	Issue log
Brainstorming	S	Leader Author Checker		Process improvements
Edit	-	Author	Product Issue log	
Follow-up	-	Leader	Product	
Exit	-	Leader	Exit criteria	

Table 2.4: Summary of Gilb and Graham inspection process. Timing is either synchronous (S) or asynchronous (A). The documents used and produced during each phase are also listed.

2.5 Asynchronous Inspection

All the inspection methods described so far have had one common element: a meeting where the entire team get together to log and discuss defects. This meeting can be expensive and/or difficult to set up and run, however, since one must ensure that all team members are available at the same time and the same place, arrange suitable meeting space and so on. An alternative to an inspection meeting is to hold the entire inspection asynchronously, by providing some means of supporting discussion without the entire team being present at the same place and time. The simplest example of an asynchronous activity is that of Usenet, the worldwide electronic news forum. Discussion proceeds by one person posting an article, which is then read by many people. Some of these people then reply to this article by posting a reply,

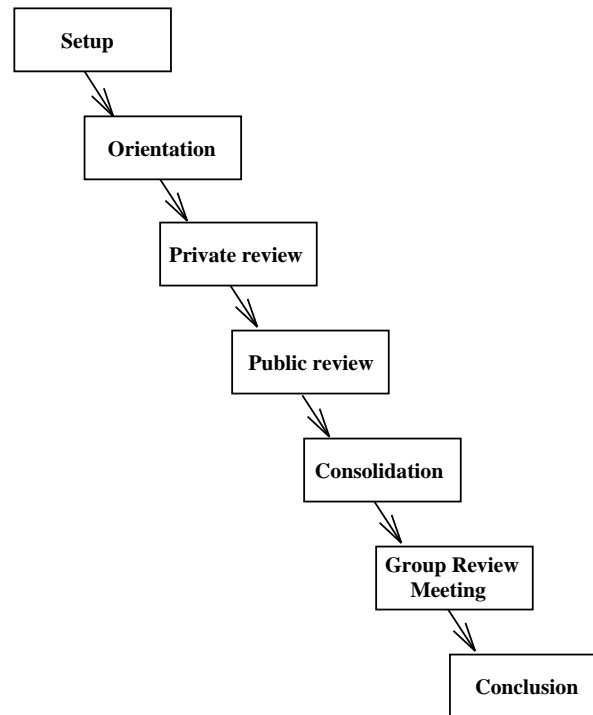


Figure 2.5: The FTArm asynchronous inspection process.

usually containing an edited version of the original article. This process continues, allowing discussion to take place without everyone being present at the same time.

A similar system can be used for inspection. By allowing users to access an on-line version of the document, they can add comments to the document (indicating potential defects) using some type of annotation technology. These comments can then be made available to other inspectors, who can comment on the comments. This procedure can continue until a consensus is reached on the status of the original comment, as either a defect or a non-issue. Once discussions have been completed on all comments, the inspection is complete and the document can enter rework. Such an inspection method has been implemented using a review tool called Collaborative Software Review System (CSRS) [57]. The tool implements an inspection method known as Formal Technical Asynchronous review method (FTArm) [55].

As with traditional inspection, FTArm defines several roles: *moderator*, *producer* and *reviewer*. The process itself is shown in Figure 2.5 and consists of seven phases. The first phase is *setup*, which involves choosing the members of the inspection team and preparing the document for inspection via CSRS. This involves organising the document into a hypertext structure and entering it into the database. *Orientation* is equivalent to overview in the Fagan process, and may involve a presentation by the author. *Private review* is similar to preparation.

Phase	Timing	Participants	Documents used	Documents produced
Setup	S	Moderator Producer		
Orientation	S	Moderator Producer Reviewer		
Private review	A	Moderator Producer Reviewer	Product Checklists	Issues Comments Actions
Public review	A	Moderator Producer Reviewer	Product Issues Comments Actions	Issues Comments Actions
Consolidation	-	Moderator	Issues Comments Actions	Consolidated issues Consolidation report
Review meeting	S	Moderator Producer Reviewer	Product Issues Actions	
Conclusion	-	Moderator		Reports

Table 2.5: Summary of the FTArm asynchronous inspection phases. The timing of each phase can be synchronous (S) or asynchronous (A). Roles defined for the process are moderator (M), producer (P) and reviewer (R). The documents used and produced during each phase are also listed.

The reviewers read each source node in turn, and have the ability to create new nodes containing annotations. These annotations can include issues indicating defects, comments pertaining to the intention of the document, which may be answered by the producer, and actions, which indicate a possible solution to remove a defect. When each reviewer has covered each node (or sooner, if required), the inspection moves on to the next phase. In *public review*, all nodes become public and inspectors can asynchronously examine each one and vote on its status. Votes cast can either confirm the issue, disconfirm the issue or indicate neutrality. Additional nodes can be created at this stage, immediately becoming publicly available. When all nodes have been resolved, or if the moderator decides that further voting and on-line discussion will not be fruitful, the public phase is declared complete. During *consolidation*, the moderator analyses the results of the private and public review phases, and summarises unresolved issues. The moderator can then decide whether a *group review meeting* is to be held to resolve the remaining issues. The final inspection report is then produced by the moderator during *conclusion*, along with a metrics report.

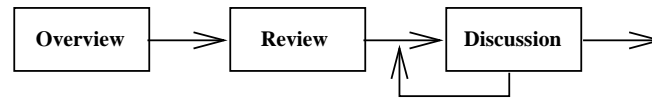


Figure 2.6: The Active Design Review process.

From this description it can be seen that FTArm is inherently computer-based. Computer support is essential for providing an asynchronous discussion environment. A summary of the FTArm process and the artifacts used and created during the process is given in Table 2.5.

2.6 Active Design Reviews

The Active Design Review (ADR) process [95] was designed to ensure thorough coverage of design documents. The technique differs from traditional inspection processes in that instead of one review involving a large number of people, several smaller reviews are held, each one concentrating on different types of defects and involving a subset of reviewers. Reviewers are chosen based on their specific skills and assigned such that each section of the document undergoes each type of review. Although referred to as design reviews, the same technique could be applied to other documents.

Essentially, only two roles are defined for the ADR process. A *reviewer* has the expected responsibility of finding defects, while the *designer* is the author of the design being scrutinised. There is no indication of who is responsible for setting up and coordinating the review.

The process is different to those discussed so far, in that it consists of a variable number of phases (Figure 2.6). It begins with an *overview* phase, where the designer presents an overview of the design and reviewers are assigned review types and document sections, and meeting times are set. The next phase is the *review* itself, which consists of each reviewer individually completing questionnaires specific to their assigned defect type. Although each reviewer has an assigned responsibility, there are also reviewers looking at the document overall, since the other reviewers are tightly focused. This phase is the equivalent of the checking phase in Gilb inspection.

The final stage, *discussion*, is where the designers read the completed questionnaires and meet with the reviewers to discuss issues raised. Several meetings are held, usually one for each reviewer responsibility, involving only that reviewer and members of the design team. Hence the meetings are kept very small; at no point does the entire inspection team come together for a meeting. When issues are agreed on, the review is complete and the designers make appropriate changes to the design document. Active Design Reviews are summarised in Table 2.6.

Phase	Timing	Participants	Documents used	Documents produced
Overview	S	Designer Reviewer	Product	Completed questionnaires
Review	A	Reviewer	Product	
Discussion	S	Designer Reviewer	Completed questionnaires	

Table 2.6: Summary of Active Design Review phases. Timing is either synchronous (S) or asynchronous (A). The review is based around the completion and discussion of appropriate questionnaires. The documents used and produced during each phase are also listed.

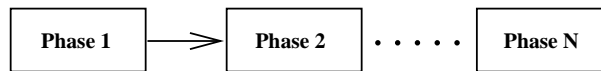


Figure 2.7: The Phased Inspection process.

2.7 Phased Inspection

The Phased Inspection technique was developed by Knight and Myers with the goal of permitting the inspection process to be “rigorous, tailorable, efficient in its use of resources, and heavily computer supported” [87]. A phased inspection consists of an ordered set of phases, each of which is designed to ensure the product possesses either a single, specific property or a small set of related properties. The phases are ordered so that each phase can build on the assumption that the product contains properties that were inspected for in previous phases. The properties that can be checked for are not necessarily those concerned purely with defects of functionality. They can include such qualities as reusability, portability and compliance with coding standards. The process, consisting of a variable number of phases, is depicted in Figure 2.7.

There are two types of phase: *single-inspector* and *multiple-inspector*. A single-inspector phase uses a rigorous checklist, with the inspector deciding whether the product does or does not comply with each item. The phase cannot be completed until the product satisfies all checks. These phases are carried out by lone inspectors. In contrast, multiple-inspector phases are designed for properties which cannot easily be described by the binary questions in single-inspector checklists. The appropriate documents are initially distributed to each participant, who begin by examining this information and generating questions which clarify and improve the documentation. The product is then inspected individually by each inspector. This individual checking makes use of a checklist that is both application specific and domain specific,

Phase	Timing	Documents used	Documents produced
Single inspector	-	Product Checklist	Defect list Completed checklist
Multiple inspector <i>Examination</i>	A	Product Sources	Question list
<i>Inspection</i>	A	Product Sources Checklist	Completed checklist
<i>Reconciliation</i>	S	Product Completed checklists	

Table 2.7: Summary of the Phased Inspection phases. All phases are asynchronous (A), except for the group meeting, which is held synchronously (S). Unlike other inspection procedures, no roles are defined for participants. The major inspection artifacts are checklists, which are used for both single and multiple inspector phases. The documents used and produced by each phase are also listed.

though the questions are not binary, as they are in the single-inspector phase. The individual checking is followed by a meeting, called a *reconciliation*, in which the inspectors compare their findings. Note that although it is not designed to do so, the reconciliation provides a further opportunity for defect detection.

Phased inspections are designed to allow experts to concentrate on finding defects they have specialised knowledge of, thus making more efficient use of human resources. For example, it may be more efficient to have domain analysts inspecting code for reusability, since they will have expert knowledge in that particular field. The phased inspection technique is summarised in Table 2.7, including the documents required and produced at each stage. Computer support for Phased Inspections is discussed in Section 3.2.2.

2.8 N-Fold Inspection

The N-Fold inspection process [82] is based on the idea that the effectiveness of the inspection can be improved by replicating it. While two teams may individually find 40-50% of the total number of defects in the document, one team may find defects not found at all by the other and vice versa. There will be some overlap between the defects found but the new defects found can outweigh this disadvantage. By increasing the number of teams performing the inspection, the percentage of defects found overall will gradually increase, until a point where the cost of finding more defects (i.e. using more teams) is greater than the benefit gained

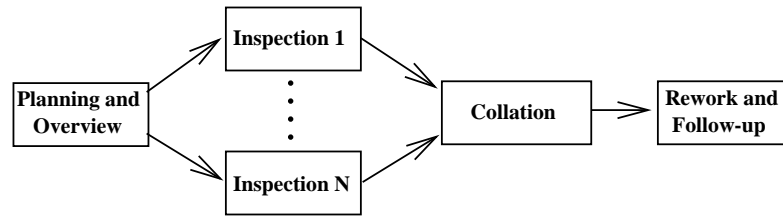


Figure 2.8: The N-Fold inspection process.

from removing those extra defects. The technique was originally designed to be used for user requirements documents, since defects injected here are the most expensive to fix, but it could be used any time that removal of defects is of paramount importance, such as in safety critical systems.

In addition to the personnel required to hold each inspection, N-Fold inspection requires the services of a *coordinator*¹, whose task is to coordinate the teams and collect and collate the inspection data. This is achieved by meeting with the moderator from each team. The description of N-Fold inspection given in [82] is rather vague, apart from the essential feature that multiple inspections are carried out by independent teams. Therefore, the following description is an extrapolated process, which would be necessary to effectively implement an N-Fold inspection.

The process is shown in Figure 2.8. It begins with the usual planning stage of a traditional inspection; however, this stage also includes deciding the number of teams to participate and other details relevant to an N-Fold inspection. There will also be an overview stage to familiarise the participants with the context and content of the document and the goals of the inspection. This is followed by a number of (usually concurrent) inspection stages, each of which is entirely independent. This means they use completely independent teams, and possibly completely different inspection processes. For example, one team may use Fagan inspection, another may use Active Design Reviews and yet another may use an asynchronous inspection. Using different inspection processes improves the independence of each inspection, and will hopefully find more defects in the document. Once each inspection has been completed, the process enters a *collation* phase, where the results of each inspection are tallied and collated by the coordinator. This stage produces a master list of defects which are given to the document's author for the traditional *rework* phase. This would be followed by a *follow-up* phase ensuring the required items have been addressed.

The N-Fold inspection process is summarised in Table 2.8. Note that many of the details of

¹This role was originally termed “moderator”, but since each inspection team already contains a moderator, and to avoid conflicting terminology the title “coordinator” has been adopted.

Phase	Timing	Participants	Documents used	Documents produced
Planning	-	Coordinator		
Overview	S	Coordinator Moderator Inspector	Product	
Inspection	-	-	Product	Defect lists
Collation	S,A	Coordinator Moderator	Defect lists	Master defect list
Rework	-	Author	Product Master defect list	
Follow-up	-	Coordinator	Product Master defect list	

Table 2.8: Summary of the N-Fold inspection phases. The only group phase is the collation stage, which can be asynchronous (A) or synchronous (S). Documents and participants of each phase are listed, except for the inspection stage, where they depend on the exact inspection method(s) being used.

this process depend on the inspection method being used. Generally, the inspections produce defect lists which must be collated into a single list.

2.9 Conclusions

Initially, there appeared to be a large variety of inspection processes. Under closer scrutiny, a number of these are quite similar, consisting of minor variations of the individual preparation/group meeting structure originally proposed by Fagan. The most radical variations come in the form of the N-Fold and asynchronous processes. The N-Fold process is designed to increase the confidence in the quality of the document by using multiple, independent inspection teams. Asynchronous inspection is designed to obviate the need for a synchronous group meeting. Most process differences occur in terminology: the terms used for process phases, participants and documents all vary between methods, which can cause confusion when comparing processes. A standard terminology and means of process description would therefore be useful, both for communicating and comparing processes.

Chapter 3

Existing Tool Support for Software Inspection

This chapter begins by describing existing tool support for software inspection. A number of such tools have been described in the literature over the past eight or so years. Table 3.1 shows the history of research in computer-supported inspection in terms of the production of tools. As can be seen, there has been a steady interest in the topic throughout the decade. The research contained in this thesis began in 1995.

There are two main types of tool: those that provide a means for capturing data from paper-based inspections and those that provide on-line inspection of documents. As it would be unfair to compare both types of tools, they are treated separately. The chapter ends with a description of the major weaknesses of existing tools that were decided to be addressed.

3.1 Tool Support for Paper-based Inspection

3.1.1 COMPAS

COMPAS [6] is a development-process support tool which began life in 1984 as a simple document management system. In the form described, its major features are modification request management, document management and an inspection and review subsystem, along with various other housekeeping functions.

Initially, a document is entered into COMPAS along with various attributes about that document. Each document has a status associated with it, which is initially set to `draft`. A

Year	Tools
1990	COMPAS ICICLE
1991	InspeQ
1992	-
1993	CSI CSRS Scrutiny
1994	CAIS QG4000
1995	Notes Inspector
1996	AISA TAMMi
1997	DCI hyperCode InspectA IPA
1998	WiP

Table 3.1: History of inspection support tools.

general purpose query facility is available to retrieve information about documents tracked by COMPAS.

In terms of support for inspection, COMPAS first allows the inspection to be scheduled. The system allows the required participants to be named and the time and place of the inspection to be set. An electronic notification can then be sent to each participant. The system can also generate a set of inspection forms to be used. COMPAS automatically collects certain data about each inspection and provides facilities for entering other types of data. Summary inspection statistics can be extracted from the tool. For example data concerning one document type may be extracted. These statistics include amount of material inspected, preparation rates, inspection rates, detection rates and effort.

While details of COMPAS are sparse, it appears to provide useful metrics collection and analysis for inspection. These facilities seem to be better than those found in many of the on-line inspection tools.

3.1.2 Quality Group 4000

Quality Group 4000 at Telefónica Investigación y Desarrollo [52] report on the use of a tool developed to support software inspections. The tool is never actually named, but is based on several simple UNIX tools.

On-line production of comments during reviews is supported via simple text files. A single command allows the moderator to collect all comments into a single file. The tool can then generate a paper report of these comments to be passed to the author for repair. The author then marks up the paper copy of the report to reflect the changes performed, and passes it back to the moderator. The moderator then uses the tool to produce a change agreement report, which summarises the author's acceptance or rejection of each comment. This report is passed to all reviewers, who can negotiate with the author about comments which have not been addressed. The tool can store data from multiple reviews and provides a menu system allowing access to the appropriate documents. It can also calculate a number of metrics.

3.1.3 Inspection Process Assistant

Inspection Process Assistant (IPA) [16] is a recent addition to the field of tool support. Its main use is to allow defects in the product to be entered on-line. IPA models a process consisting of planning (checking work product and organising the inspection team), individual preparation (inspectors identify defects), meeting (group discussion of defects found), rework (fixing of defects), verification (ensuring defects have been correctly fixed) and finalisation (moderator validates results). It allows a database of available inspectors to be kept, and also allows information on documents to be stored.

An inspection begins with the moderator using IPA to select the documents to be inspected and the participants who will be involved. A viewpoint (area of responsibility) can be allocated to each participant, and the reader and recorder for the group meeting set. Finally, a date and time for the group meeting is set.

When the moderator has defined the inspection, other participants must then access IPA to customise the definition for themselves. IPA can then be used to record defects for that inspector. Each defect can have a location, a summary, a detailed description and a classification. When the inspector has finished preparation, the amount of time spent in preparation can be entered.

When all inspectors have finished preparation, the moderator can use IPA to merge the individual defects together into one single list to be discussed at the meeting. Unfortunately, this is just a simple merge and there is no facility for the moderator to edit this list, e.g. to remove duplicates.

During the group meeting, IPA is used by the recorder to record the disposition for each defect (e.g. accept, reject, duplicate, etc.) and the overall result of the inspection (such as "work product accepted"). The producer then can then use IPA during rework to browse the defect list and fix the appropriate items. The producer can mark each defect as corrected or not

corrected. During verification, the verifier uses IPA to browse the defect list and mark each item as verified or not verified, commenting on the decision as appropriate. After verification, the moderator finalises the inspection and can produce an inspection report. Finally, inspection data can be exported from IPA for further analysis.

IPA provides fairly comprehensive support for the production of defect lists and collection of some data. One major deficiency is the inability of the moderator to edit the merged defect list before the meeting. This could reduce the size of the list, thereby reducing the length of the meeting. And of course, like the other tools in this section, the possibilities provided by on-line document presentation are not exploited.

3.1.4 Comparison of Tools to Support Paper-based Inspection

It is quite obvious that IPA has the most comprehensive support of the three tools. COMPAS only supports the tracking of documents and inspection planning. QG4000 supports individual comment preparation while IPA supports both individual preparation and the group meeting. Although comment preparation on-line may make inspection more efficient, it may also be inconvenient to work with both paper and on-line material at the same time. Hence, most work in tool support for inspection has concentrated on moving the entire process on-line.

3.2 On-line Inspection Tools

On-line inspection tools go one stage further than those already discussed: the entire process is carried out on-line. These tools tend to be far more complex than their relatives discussed in the previous section. The features provided by these tool can be classed under four broad categories: document handling, individual preparation, meeting support and data collection.

Document Handling Paper-based inspection requires the distribution of multiple copies of each document required. Apart from the cost and environmental factors associated with such large amounts of paper, cross-referencing from one document to another can be very difficult. Since most documents are produced on computer, it is natural to allow browsing of documents on-line. Everyone has access to the latest version of each document, and can cross-reference between documents. Also, the actual presentation of these documents can be designed to enhance their inspection.

Computer support allows on-line annotation of documents, with annotations linked to the part of the document to which they refer. They can then be made available for all inspectors to study before and during the inspection meeting. This has the added advantage of helping

to reduce the inaccuracies and mistakes which can occur during the inspection meeting, including the failure to record some comments altogether. This effect has been observed by Votta [118] and can occur in several situations, including when inspectors are unsure of the relevance of their comments. By storing all comments on-line, it is easier to ensure that each one is addressed.

Individual Preparation There are several ways in which tool support can assist in individual preparation. Tools can be used to find simple standard violations. While not as important as logic defects, these must still be found to produce a correct document. If finding them can be automated, inspectors can concentrate on the more difficult defects that have a potentially greater impact if not found. This may be achieved by the introduction of new tools, or the integration of the inspection environment with existing tools. There are various levels of integration, from simply reporting defects to actually producing an annotation relating to the defect for the reviewer to examine.

Generally, inspectors make use of checklists and other supporting documentation during preparation. By keeping these on-line, the inspector can easily cross-reference between them. On-line checklists can also be used by the tool to ensure that each check has been applied to the document, thereby enforcing a more rigorous inspection, while on-line standards assist the inspector in checking a document feature for compliance.

Meeting Support Prior to the group meeting, computer support may be used to monitor inspectors' effort. The moderator can use this information to decide when is the best time to move from the preparation stage to the meeting, taking account of the amount of preparation performed by each inspector. The moderator can also exclude anyone who has not prepared sufficiently for the group meeting, or encourage them to invest more effort.

Since guidelines state that a meeting should last for a maximum of two hours [37], it may take many meetings to complete an inspection. There is a large overhead involved in setting up each meeting, including finding a mutually agreeable time and place, a room to hold the meeting and so forth. There is also an overhead involved for each participant travelling to the meeting. By allowing a distributed meeting to be held using conferencing technology, it may be easier for team members to "attend" the meeting using any suitably equipped workstation.

An alternative solution to the meeting problem is to remove the synchronous meeting stage altogether, performing the inspection *asynchronously*. In this type of inspection, each inspector can perform their role independently. The inspection moves from stage to stage when every inspector has completed the required task. This type of inspection can also reduce

meeting losses which occur when participants fail to report defects.

When a meeting is taking place, it can sometimes be useful to conduct polls to quickly resolve the status of an issue. This is especially important if the meeting is being held in a distributed environment. Computer support can allow polls to be quickly taken, thus helping the inspection meeting progress more rapidly.

Data Collection An important part of inspection is the collection of data which can be used to provide feedback to improve the inspection process. The measures will include time spent in meeting, defects found, overall time spent in inspection, etc. Collecting this data is time-consuming and error-prone when carried out manually. In fact, Weller [119] suggests “...you may have to sacrifice some data accuracy to make data collection easier...”, which is obviously undesirable. Computer support allows data from the inspection to be automatically gathered for analysis. This allows inspectors to concentrate on the real work of finding defects. Furthermore, the data can be analysed with little further work, unlike manual data collection where the data has to be entered before it can be analysed which is an error-prone process.

These four areas represent the major facilities which can be provided by an on-line inspection support tool. A number of such tools are now reviewed in terms of their support for these four areas.

3.2.1 ICICLE

ICICLE (Intelligent Code Inspection in a C Language Environment) [9, 12, 13, 104], is designed to support the inspection of C and C++ code. This tool is unique in making use of knowledge to assist in finding common defects. Since the knowledge is of a very specific kind, ICICLE is less suitable for supporting inspection of other document types. It can, however, be used to inspect plain text files by turning off the initial analysis. The tool supports both individual preparation and the inspection meeting itself. During the inspection meeting, the tool provides the functionality available in individual checking, supplemented by support for cooperative working.

Document Handling The source code is displayed in a large window with each line numbered (see Figure 3.1). This window can be augmented by a second code window, allowing the user to compare two parts of the code simultaneously. Next to the line numbers are two symbols referring to comments. A letter indicates the status of the comment. This can include *deferred* (not dealt with yet), *ignored* (user decides the comment is inappropriate or otherwise suspect) or *transferred* (chosen to be discussed at the inspection meeting). The second symbol

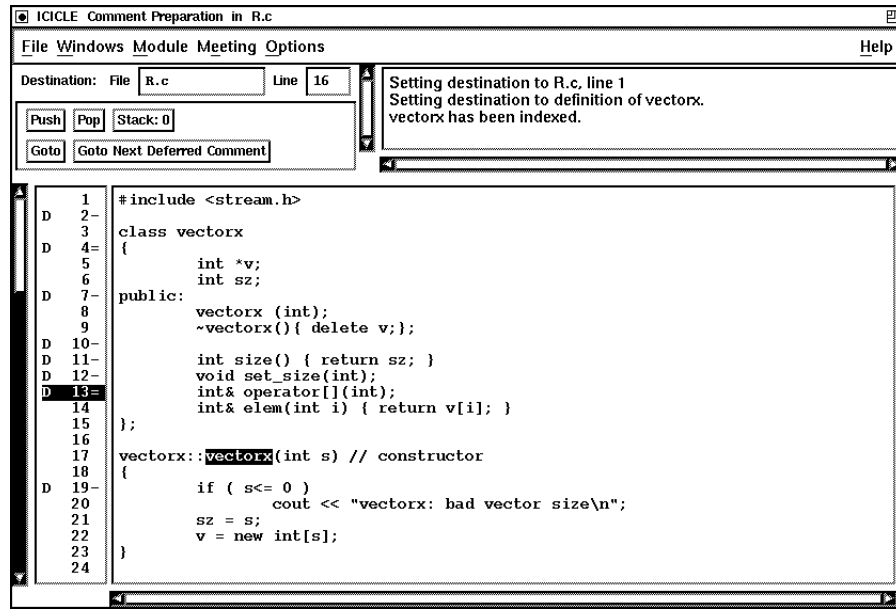


Figure 3.1: The main ICICLE display.

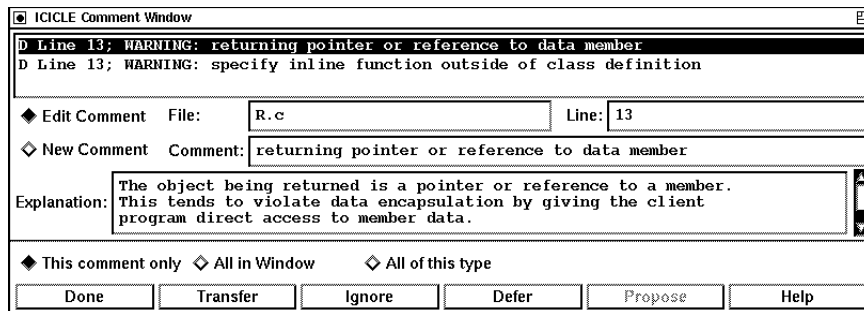


Figure 3.2: The ICICLE comment preparation window.

indicates the presence of a comment for this line. A hyphen indicates a single comment, while an equals represents multiple comments.

Clicking on a line opens a comment window like that shown in Figure 3.2 for the line. This window allows a comment to be modified or inserted and its status changed. Any changes to this comment can be propagated to all comments on the line or even all comments in the code which have the same text.

Individual Preparation ICICLE can automatically prepare comments on source code using its analysis tools. These include the UNIX tool `lint` and ICICLE's own rule-based static debugging system. `lint` can be used to detect certain defects in C code, such as unreachable statements and possible type clashes. The ICICLE rule-based system can be used to flag

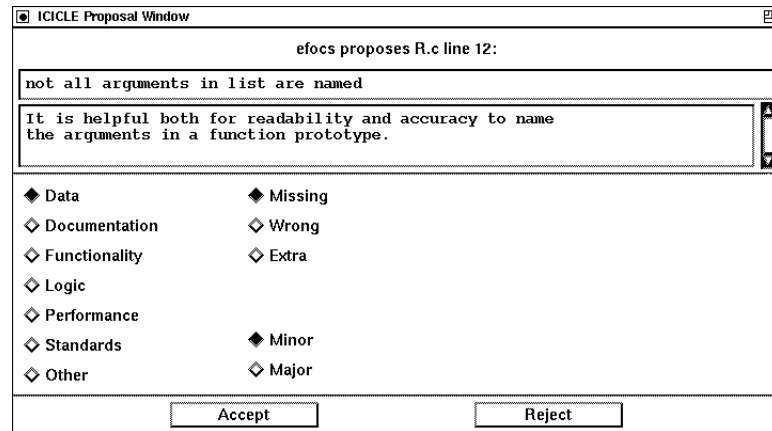


Figure 3.3: The ICICLE comment proposal window.

both serious defects, such as failure to deallocate memory, and more minor defects, such as standards violations. There is also the ability to include customised analysis procedures. The comments produced by all these tools can either be accepted by the inspectors if they agree with them, modified or else completely rejected. ICICLE also provides a facility to allow browsing of UNIX manual pages. The system also provides cross referencing information for “objects” such as variables and functions. For example, clicking on the use of a variable would give the user an option to move to the point of declaration, or any other usage of the variable. This facility is available over multiple source files.

Meeting Support The inspection meeting is held with every inspector using ICICLE in the same room. Distributed meetings are not supported, since the authors “do not wish to supplant the ordinary verbal medium by which the bulk of meeting communication occurs” [104]. During the meeting, each inspector has access to all documents as well as their own comments. Each inspector has the code window displayed on screen. The reader controls the traversal of this window for all participants, just as a single inspector does during comment preparation. Every code window is locked to the reader's view, although an inspector can open an extra window to allow simultaneous inspection of two sections of the code.

The reader proceeds through the document until an issue is proposed by an inspector, opening a proposal window on all displays. The scribe's proposal window is shown in Figure 3.3. The team discuss the comment, and when discussion is complete, the scribe is able to classify the comment and accept it, or reject the comment completely. If the comment is accepted it is stored in a file which becomes the output of the meeting. During the meeting, participants can send single line text messages to all other participants.

Data Collection When the inspection meeting is complete, ICICLE generates a list of all accepted defects to be given to the author of the product under inspection. A summary of the defects by type, class and severity is also generated. The scribe can also prepare a report detailing the total time spent in preparation and in meeting, the inspectors present and other such process information.

3.2.2 InspeQ

InspeQ (Inspecting software in phases to ensure Quality) is a toolset developed by Knight and Myers [65, 66, 87] to support their phased inspection technique. The technique was developed to allow the inspection process to be “rigorous, tailorable, efficient in its use of resources, and heavily computer supported” [65]. Phased inspections were described in detail in Section 2.7.

Document Handling The *work product display* is used to browse the document under inspection. By using multiple copies, the inspector can simultaneously examine separate parts of the same document. The browser allows the inspector to search the document. The *comments display* allows the inspector to note any issues found. InspeQ carries out formatting of these comments before they are passed on to the author.

Individual Preparation A *checklist display* is used to display the checklist associated with the current inspection. The checklist also allows the inspector to indicate completion of each check, by marking each item as *complies*, *does not comply*, *not checked* or *not applicable*. To help enforce a rigorous inspection, InspeQ ensures that all checklist items are addressed by the inspector before the product exits the phase. The authors plan to extend the system to ensure that each checklist item is applied to every feature associated with that item. Checklists usually ensure compliance with one or more standards, therefore a *standards display* is available which presents each standard in full.

The *highlights display* can allow the inspector to quickly identify specific features of the document. These can be highlighted but can also be displayed in a separate window for examination. An example would be to highlight all the `while` statements in a C program to allow them to be checked for correctness, without the distraction of the surrounding code. This function requires syntactic information about the document, which is more readily available for code than any other type of document.

Meeting Support Since InspeQ is designed for individual inspector use, there is no support for group meetings. It can, however, generate the comment list for each inspector. These lists

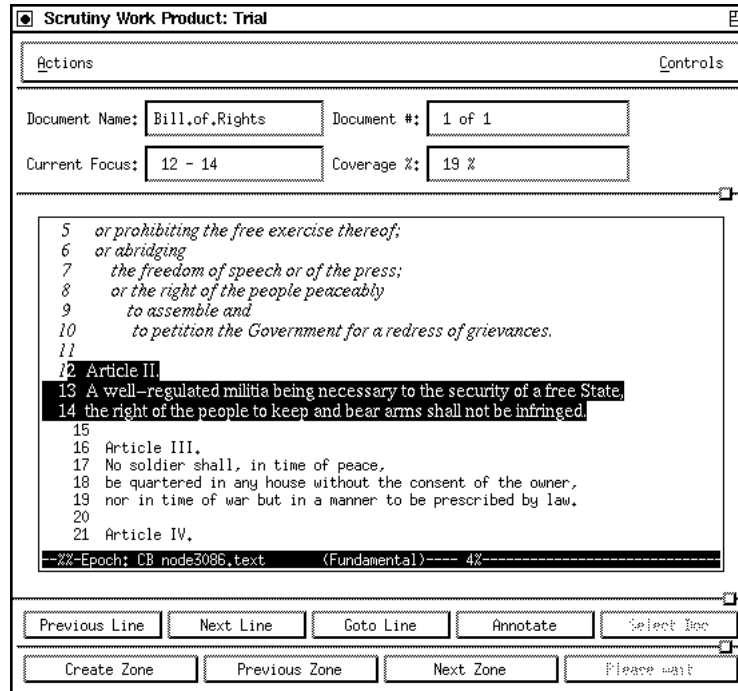


Figure 3.4: The Scrutiny work product window.

are then compared at the reconciliation.

Data Collection No data collection facilities are described.

3.2.3 Scrutiny

Scrutiny [15, 43, 44] is a tool based on the inspection method used at Bull HN Information Systems. This process consists of four stages. The first stage is *initiation* and is comparable to overview in the Fagan model. The second stage is *preparation*, as in the Fagan model. The inspection meeting itself is called *resolution*, while the final stage, *completion*, encompasses both rework and follow-up. The roles taken by each participant are also similar, however Scrutiny also implements some changes. First, the moderator's role is changed to include the duties of the reader. In addition, the recorder role can be taken by more than one person. Scrutiny also explicitly implements the role of the producer, who can answer questions regarding the document. Finally, there is another role in the form of the *verifier* who ensures the defects found by the inspection team have been correctly addressed by the author. This role may be assigned to any participant. Any other members of the team are cast as inspectors. Each stage of the process, along with each of the three roles, is modelled in Scrutiny.

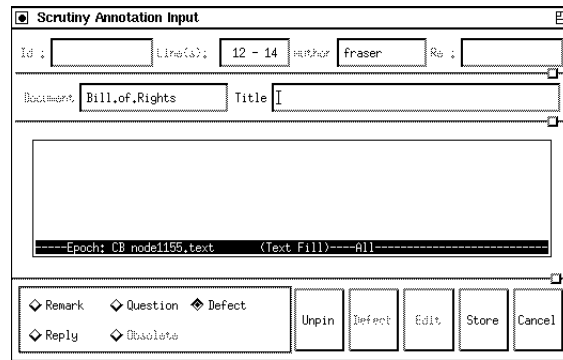


Figure 3.5: The Scrutiny annotation window.

Document Handling The *work product window* allows each inspector to view the document under inspection (see Figure 3.4). The document is displayed with each line numbered and the current focus indicated by reverse video. The current focus is usually a single line but may also be a zone of several lines. Text which has been inspected is italicised, and the percentage of the document covered is displayed in the top right hand corner. The window has controls to move through the document line by line, and also has controls to mark a zone. Finally, there is a button to enable the creation of a new annotation.

When an annotation is created or modified, it appears in an *annotation window*, an example of which is given in Figure 3.5. This displays the line numbers to which the annotation refers and the author of the annotation, along with its content and a title. Buttons allow the type of annotation to be recorded as either a question, potential defect, remark or reply. When an annotation is created, an icon appears beside the line or zone to which it refers. Scrutiny currently only supports text documents.

Individual Preparation Here, Scrutiny simply allows the inspector to traverse the document, making annotations which can be used during the resolution stage. There is no assistance with checklists or other supporting documentation.

Meeting Support Before the inspection meeting is started, the moderator can view the preparation time of each inspector, to ensure that enough time has been given to allow adequate preparation. Each inspector also has the opportunity to add time for any off-line preparation which they may have engaged in.

During the meeting, the work product window is used by each participant to view the document, with the moderator having additional controls to change the current focus and to initiate a poll. The moderator guides the inspectors through the document, while they read

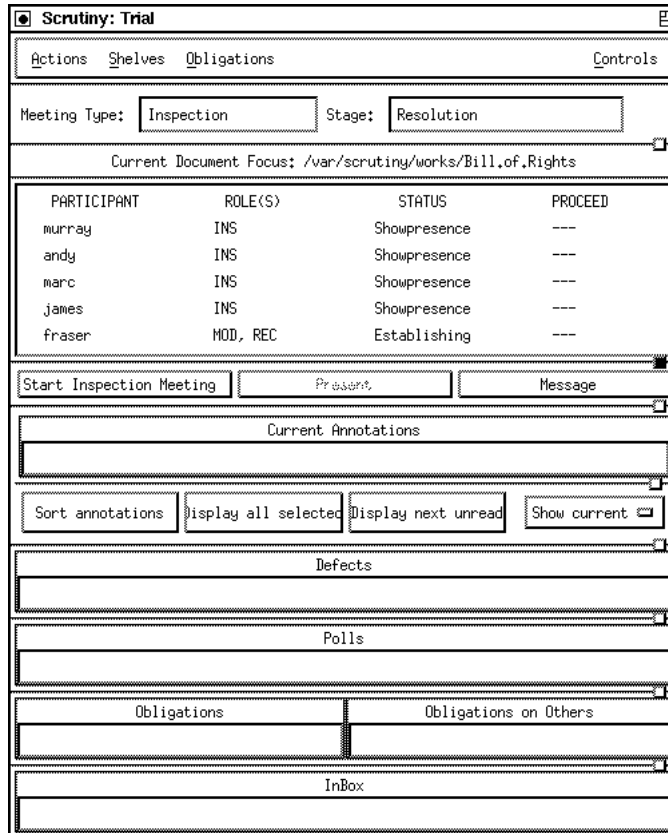


Figure 3.6: The Scrutiny control window.

and discuss the annotations made. Polls are used to resolve the status of an annotation.

Scrutiny also provides a main control panel called the *control window*, a copy of which is seen by each inspector (Figure 3.6). This window consists of four major subwindows. The *participant status* display contains a list of the participants along with an indication of their current activities. The *annotations* subwindow contains a list of annotations made on the current document, along with their owners and a type. The *defect* subwindow lists defect reports that have been discussed and their status agreed. The status includes the type and severity of the defect. Finally, every time a poll is taken during the inspection meeting to resolve an issue, a record of it is kept in the *polls* subwindow.

Scrutiny can be used for both same-place and distributed inspection. The latter makes use of teleconferencing facilities. It is also possible to hold distributed inspections without these facilities by making use of Scrutiny's built in textual communications systems. The discussion client allows inspectors to exchange textual points of discussion. Each participant has a list of the current discussion points which can be read and replied to. Replies have a reference to the original point, and participants can traverse these chains of points, allowing them to follow

a discussion and then add their own comments. Scrutiny also provides a means of sending a simple message to meeting participants. In addition to composing your own message, there are several frequently required messages, such as a request to move to the previous line, which can quickly be sent. These messages can be sent to named individuals, or the group as a whole. It is not clear how effective these mechanisms are when holding a synchronous meeting, since the medium is obviously not as information rich as face-to-face communication.

Data Collection Scrutiny automatically generates an inspection report containing all the relevant information about the inspection and its participants, details of the time spent by each participant in the inspection and the coverage of the document they achieved. It also contains a complete defect list with summary information.

3.2.4 CSI

Vahid Mashayekhi at the University of Minnesota has created three prototype inspection support tools, described in his doctoral thesis [83]. The first of these, Collaborative Software Inspection (CSI) [84], is designed to support inspection of all software development products. The tool is described as applied to the Humphrey model of inspection [50]. In this variation, each inspector creates a list of defects during individual inspection, which are then given to the author of the document before the inspection meeting. It is the author's task to correlate these defect lists and to then address each defect at the inspection meeting.

Document Handling CSI provides a browser for viewing the document under inspection, which automatically numbers each line. When a line is selected, an *annotation* window pops up, allowing the inspector to make a comment about that particular line. This annotation is supported by hyperlinks between the annotation itself and the document position to which it refers. Since annotations can only refer to one line, and there may be a need for general comments about an area of the document, CSI also supports a *notepad* system, which allows annotations about missing material.

Individual Preparation Support is available from CSI for detecting defects by provision of on-line *criteria* which help the inspector determine defects. Also, when recording annotations, the inspector is given guidance in categorising and sorting defects. After all inspectors have finished individual inspection, the author can access all annotations associated with the document and correlate them into a single defect list, supported by CSI through automatically summarising and integrating the individual defect lists. The author can then categorise each

defect, either accepting it or rejecting it. CSI also allows the author to sort the defect list on multiple keys, including severity, time of creation and disposition.

Meeting Support At the inspection meeting, the document under inspection is made visible on a window on each inspector's screen. The author guides the meeting using the correlated defect list. Each item is discussed, and when agreement is reached regarding its severity, this is noted by the recorder in the *action list*. The original annotations are available at this point to help inspectors understand the defect, and further annotations can be added during the meeting. When the end of the defect list is reached, the inspectors agree on the status of the meeting, indicating whether the material under inspection is to be accepted or reinspected. CSI provides support for distributed inspections through an audioconferencing tool called Teleconf [102].

Data Collection The *inspection summary* is used to record meeting information such as team members present, their roles and the status of the inspection meeting. CSI also provides a *history log*. This collects several metrics from the process, such as the time spent in the meeting and the time taken to find a defect, as well as the number and severity of defects found.

3.2.5 CAIS

The next prototype developed by Mashayekhi is Collaborative Asynchronous Inspection of Software (CAIS) [85]. It is designed to be used asynchronously and therefore does not rely on having all inspection participants present for any part of the process. It is hoped that asynchrony can reduce the amount of time required to complete the inspection, since there is no need to find a common time when all inspectors are free to carry out the meeting. An asynchronous meeting can also solve some of the social problems which occur in synchronous meetings, such as inspectors free riding, production blocking (where an inspector has to withhold a contribution until an appropriate time) and limited air time (only one person can speak at a time).

Document Handling This system uses CSI for displaying and annotating documents.

Individual Preparation Again, CSI is used for individual preparation, therefore the facilities here are identical.

Meeting Support The asynchronous meeting is supported by a meeting object. This provides a discussion and voting mechanism, and also notifies each participant when new discussion has taken place. If an issue cannot be resolved during the asynchronous meeting it can be sent to a synchronous meeting held later.

Data Collection The history log is used to collect data on the inspection. The metrics collected are: number of comments per person, number of votes per person, time for individual defect collection, total meeting time, and when each participant made use of the system.

3.2.6 AISA

The final prototype developed by Mashayekhi is Asynchronous Inspector of Software Artifacts (AISA) [108]. This prototype is designed to allow asynchronous inspection of graphical documents, such as Data Flow diagrams. The tool is based on the Mosaic WWW client and supports a three stage inspection process: defect collection (individual detection of defects), defect correlation (where the producer integrates individual defect lists into a single master list) and the inspection meeting (held asynchronously).

Document Handling The use of a WWW client allows AISA to support most graphical documents. The document is prepared as a clickable image map (whether this is done automatically or manually is not mentioned). Each document has a hierarchy of graphical images, allowing the user to successively zoom in to smaller areas, although only two levels are implemented. Each component document has a button allowing that component to be annotated, along with a list of annotations for that component. Annotations can be viewed by clicking on them.

Individual Preparation AISA simply support viewing and annotation of the document under inspection. No other help is provided, except that AISA allows each participant to signal their completion. When all participants have finished, a message is sent to the producer indicating the end of the defect collection phase. The producer then uses AISA to correlate the defect lists, removing duplicates and arranging them in the order in which they are to be discussed during the meeting.

Meeting Support The correlated defect list generated by the producer becomes the agenda for the asynchronous meeting. Each defect has an associated thread of discussion that participants can add to. When discussion of a defect is complete, a proposal is generated for that

defect, which each participant can vote to accept or reject. The output of this meeting consists of a summary of all defects and proposals.

Data Collection No data collection facilities are mentioned.

3.2.7 Notes Inspector

Notes Inspector (NI) [115] was also developed at the University of Minnesota. This tool was built using Lotus Notes and, like CAIS and AISA, implements an asynchronous model of inspection. In this model, the inspection consists of an individual defect finding phase followed by an asynchronous meeting where participants discuss and vote on defects found.

Document Handling The product is stored as multiple Notes documents, with each line of text being stored as a single document. This appears a somewhat artificial way of storing the document, but it allows annotations to be associated with individual lines of text. Non-textual documents cannot be accommodated, however. Annotations can appear either as additional lines within the document, or as symbols next to the appropriate line.

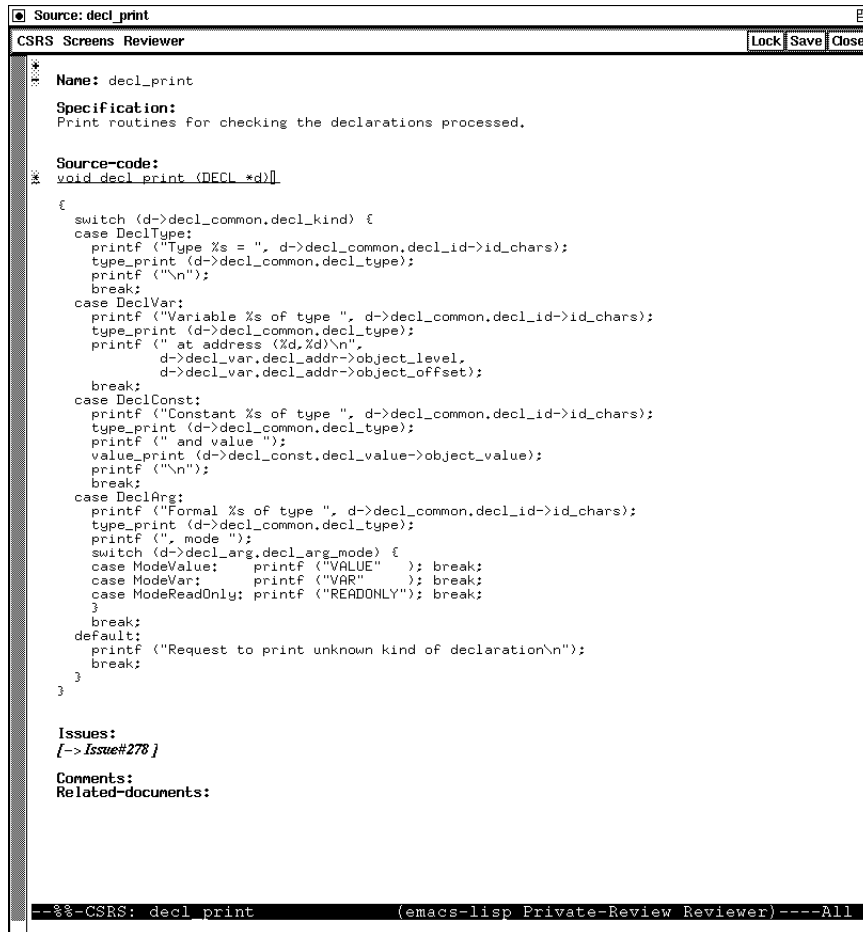
Individual Preparation This tool only permits the document to be read and annotated. No further facilities are available to help individual preparation.

Meeting Support Notes Inspector is an asynchronous system and does not support the traditional group meeting. However, a discussion and voting system is available for use during the asynchronous meeting. Each defect can have a thread of discussion, which inspectors can view and extend. To resolve an issue, a proposal is created. Each inspector can vote to accept or reject this proposal, or to abstain from the vote. Defects which are not resolved during the asynchronous meeting can be set aside for a traditional synchronous meeting.

Data Collection No data collection facilities are available.

3.2.8 CSRS

Collaborative Software Review System (CSRS) [55, 56] is probably the most flexible of all tools described here as it can be customised to support different variants of the inspection process. This is accomplished using a process modelling language [117]. This language has several facilities, including constructs for defining phases of the method, a construct for defining the role of each participant, and constructs to define the artifacts used during the



```

Source: decl_print
CSRS Screens Reviewer [Lock Save Close]
**
* Name: decl_print
Specification:
Print routines for checking the declarations processed.
Source-code:
* void decl_print (DECL *d){
{
switch (d->decl_common.decl_kind) {
case DeclType:
printf ("Type %s = ", d->decl_common.decl_id->id_chars);
type_print (d->decl_common.decl_type);
printf ("\n");
break;
case DeclVar:
printf ("Variable %s of type ", d->decl_common.decl_id->id_chars);
type_print (d->decl_common.decl_type);
printf (" at address (%d,%d)\n",
d->decl_var.decl_addr->object_level,
d->decl_var.decl_addr->object_offset);
break;
case DeclConst:
printf ("Constant %s of type ", d->decl_common.decl_id->id_chars);
type_print (d->decl_common.decl_type);
printf (" and value ");
value_print (d->decl_const.decl_value->object_value);
printf ("\n");
break;
case DeclArg:
printf ("Formal %s of type ", d->decl_common.decl_id->id_chars);
type_print (d->decl_common.decl_type);
printf (" mode ");
switch (d->decl_arg.decl_arg_mode) {
case ModeValue: printf ("VALUE" ); break;
case ModeVar: printf ("VAR" ); break;
case ModeReadOnly: printf ("READONLY"); break;
}
break;
default:
printf ("Request to print unknown kind of declaration\n");
break;
}
}
}

Issues:
[-> Issue#276 ]
Comments:
Related-documents:

--%CSRS: decl_print (emacs-lisp Private-Review Reviewer)---All

```

Figure 3.7: The main CSRS window.

inspection. The latter also includes support for checklists. The language can also be used to define the user interface, as well as to control the type of data analysis carried out by CSRS. The description of CSRS presented here is based on its use to support an asynchronous methods of inspection known as Formal Technical Asynchronous review method (FTArm). This method is described in more detail in Section 2.5, but essentially consists of a phase of individual review of the product (where all comments are kept private), followed by public review (where all comments become publicly available and are discussed asynchronously).

Document Handling A document is stored in a database as a series of nodes. For source code, these nodes would consist of functions and other program constructs. Source nodes are created at the start of the inspection by the document author with the aid of the moderator. The nodes are connected via hypertext-style links, allowing the inspector to traverse the document. A typical source node is displayed in Figure 3.7. The name of the function is given, followed

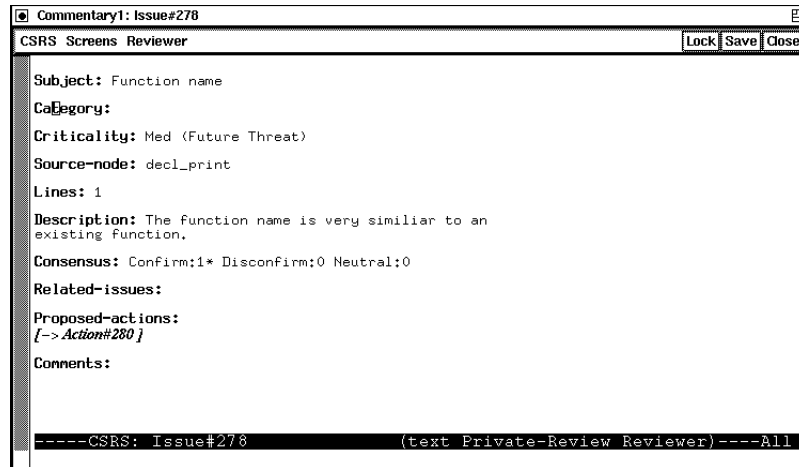


Figure 3.8: A CSRS issue node.

by a specification of its intended function. This is followed by the source code itself.

Annotations are also stored as nodes, and can be one of three types. The first type is a *Comment* node, which is used to raise questions about the document and to answer them. These are made public to all inspectors. An *Issue* node indicates a perceived defect in the source node. Issue nodes are initially private to individual reviewers. An example issue node is given in Figure 3.8. This issue is linked to the source code in Figure 3.7, where a link to the issue node can be seen near the bottom of the display. Finally, an *Action* node is a suggestion of the action that should be taken to resolve an issue. These are also private to reviewers. The action node given in Figure 3.9 details a possible fix for the issue raised previously.

Individual Preparation The FTArm method predominantly consists of individual work, and this is where CSRS provides the most support. During the private review phase, each inspector has a summary of which nodes have been covered and which have still to be covered. This information is also available to the moderator, who will use it to decide when to move on from private review to the next phase. Since additional nodes may be created after a reviewer has reviewed all the currently available nodes, CSRS has the facility to automatically e-mail all reviewers when new nodes are created and have to be reviewed. CSRS also provides an on-line checklist of standard issue types to assist the reviewer.

Support during public review is similar to that for private review, except now all nodes are accessible to all participants. This time the main focus is on issue nodes. Each reviewer has to visit each node, where CSRS can be used to vote on that node's status. Again, the reviewer has summary information available, indicating which nodes have still to be visited. The moderator can also use this information to decide when to terminate public review, usually

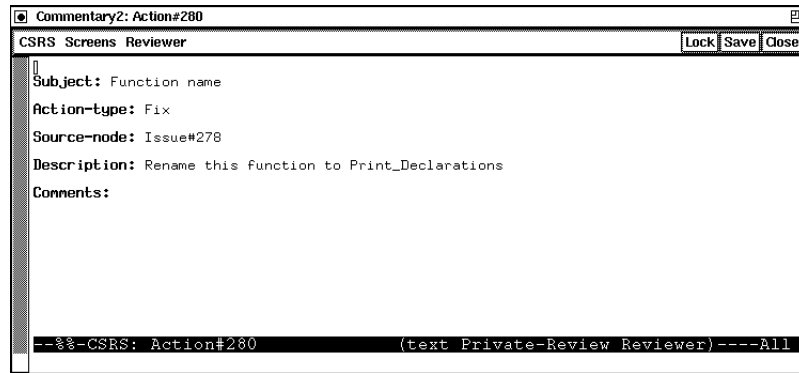


Figure 3.9: A CSRS action node.

when all reviewers have visited all nodes.

Meeting Support CSRS has little in the way of group meeting support, due to the predominantly asynchronous nature of the inspection method implemented. The group review meeting must be held face-to-face in the traditional manner. CSRS does not provide any support except to help the moderator summarise the results and to produce a $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ formatted report.

Data Collection CSRS provides automatic collection of such data as number and severity of defects and time spent reviewing each node. It also has the ability to keep an event log, which details the entire inspection from start to finish, allowing detailed (manual) analysis later on.

3.2.9 TAMMi

TAMMi [101, 114] is a tool developed to support the GRCM quality model [113]. The model is based on goals (G), rules (R), checklists (C) and metrics (M) and is designed to support the sharing of consistent information between designers and inspectors. Each quality goal is broken down into a number of rules, which are in turn broken down to form checklists. Rules are used to guide software design, while checklists are used to help inspectors check for compliance with rules.

Document Handling TAMMi is designed to support inspection of PostScript documents, allowing inspection of graphical as well as textual documents. Annotations can be marked directly on the document, with vertical lines in the margin being used to signal their occurrence. Each annotation may have a subject, a description, the rule and checklist item which classify this item, and an indication of its criticality. A separate window provides a summary of all annotations made and allows the user to jump to any given annotation.

Individual Preparation TAMMi provides explicit support for support for the GRCM model by presenting the checklist on-line to guide inspectors. The checklist is also used to classify items found.

Meeting Support TAMMi does not explicitly support group meetings. However, a report listing all the annotations entered by an inspector can be printed, which can then be used during a traditional face-to-face meeting. The tool could also be used as an aid to the scribe.

Data Collection No data collection support is mentioned.

3.2.10 InspectA

InspectA [93] is another attempt to explore the possibilities offered by asynchronous inspection. The inspection process used starts with a phase of individual inspection, where inspectors generate their initial list of comments. This is followed by review, where copies of these initial lists are exchanged amongst inspectors, allowing them to discuss the validity of each comment. This discussion proceeds asynchronously. The review phase is followed by another round of individual inspection, with all comment lists being available to each inspector. Comments can be reclassified, new comments added, and so on. At the end of this phase, the moderator prepares a master list of comments to send to the author for repair.

Document Handling InspectA supports only plain text documents. It also allows a list of defects to be entered. Each defect may include the product text which is incorrect, a description of the defect, a class (Missing, Wrong or Extra) and a severity (Major or Minor). The defects are not linked to the position in the document where they occur.

Individual Preparation InspectA allows the traversal of the document under inspection, and can also display a single source document and a checklist. It also provides a mail facility allowing participants to exchange comments and ideas. This mail facility is also used to distribute defect lists at the end of the individual phases.

Meeting Support InspectA is designed to perform a completely asynchronous inspection, so no synchronous meeting facilities exist. It supports the review phase of the inspection by providing a find facility to help locate defects in the text (since no position is stored with the defect). A notepad is also available for making comments. The tool also provides facilities for the moderator to combine multiple defect lists into a master list.

Data Collection No data collection facilities are supplied.

3.2.11 hyperCode

hyperCode [97] is a WWW-based tool also designed to support different-time, different-place inspection, however the process used is much simpler than that of InspectA. Preparation and collection are performed concurrently, while resolution of issues is performed during the rework phase. When an inspection has been started, the relevant inspectors are notified by e-mail. During a designated time-span, an inspector uses a standard WWW browser to study and annotate the code under inspection. All annotations are public. When this period has elapsed, e-mail is again sent to the inspectors and the author may then examine annotations made and decide on the rework to be performed. On completion of the rework, the moderator is informed, who then verifies the rework.

Document Handling hyperCode makes use of a standard web browser to allow code to be viewed and annotated. Code listings have the latest changes marked and are automatically translated into HTML. Line numbers in the source listing becoming hyperlinks for adding annotations, page numbers are used to build a table of contents, and so on. Although the example application is code inspection, it should be possible to apply the same techniques to other document types.

Individual Preparation hyperCode only allows the inspector to traverse the code making annotations. There is no mention of the use of checklists or other supporting documentation such as design documents or standards.

Meeting Support No meeting support is present in hyperCode - the entire inspection is held in an asynchronous fashion.

Data Collection Data collection facilities are not mentioned by the authors

3.2.12 WiP

WiP (Web Inspection Prototype) [46] is yet another WWW-based inspection support tool, originating from the same institution as TAMMi (described above). Like TAMMi, WiP is based around the GRCM quality model. An inspection begins with a setup phase, where the documents required for the inspection are passed to WiP, inspectors are selected and roles defined. During individual inspection, the inspector has access to the document under inspection

and can create annotations on a line-by-line basis. Related documents and checklists can be accessed, and help concerning the process and the tool is always available. Statistics about the document can also be accessed. The next phase is public inspection, where WiP combines inspectors individual lists into a single list. All inspectors can now view all annotations, and add more as required. At the end of this phase, a report summarising the inspection can be created.

Document Handling WiP makes use of a standard web browser to allow documents to be viewed and annotated. Documents are restricted to text only and the view is limited to twenty lines at a time. Annotations may only refer to single lines, but can be classified and have a checklist reference associated with them.

Individual Preparation Inspectors can view and annotate the document. Checklists and other supporting material can also be viewed.

Meeting Support No meeting support is implemented in WiP. All inspection phases are held asynchronously.

Data Collection WiP collects the time spent inspecting the document and the number of issues generated, and calculates the inspection rate and issues per thousand lines.

3.2.13 Distributed Code Inspection

The Distributed Code Inspection (DCI) prototype proposed by Doherty and Sahibuddin [30] is designed to implement their distributed inspection process. The process starts with a planning activity, where a synchronous or asynchronous methods is chosen. In the synchronous model the next activity is a kick-off meeting where the participants are briefed on the source code and the objectives of the inspection. In an asynchronous method, a briefing document is distributed by e-mail. Both models continue with individual preparation, where participants attempt to find defects in the code. A group activity then follows in both methods. Confusingly, the authors state that all participants must be available at the same time in both methods – surely this contradicts the asynchronous model? Both methods then end with a follow-up phase.

Details of the actual tool are sketchy, and written in the future tense, implying that the system has not yet been created. It is (or will be) another WWW-based tool, allowing any user with an ordinary WWW browser to make use of the system.

Document Handling A code viewer is used to display the code under inspection. Although not explicitly stated, it can be assumed that this viewer is text only. Comments about the code can be stored.

Individual Preparation On-line help is available, though it is not clear what this help refers to, e.g. the tool, the code, etc. The specification of the code is made available on-line. An e-mail facility is also available to allow inspectors to seek clarification regarding the code.

Meeting Support Chat and e-mail facilities are available for group discussion. The comments created during individual preparation can be viewed and discussed.

Data Collection No data collection support is mentioned.

3.2.14 Comparison of On-line Tools

Table 3.2 summarises the features of existing on-line inspection tools. It can be seen that while basic document inspection and annotation are well-supported, the more advanced features are less common. This section compares the features supported by each tool.

Document support Most tools handle only plain text documents. ICICLE, Scrutiny, CSI, CAIS and hyperCode use the same technique of displaying the document with each line numbered. Annotations can then be made which are linked to an individual line. Scrutiny also uses the idea of a current focus, which is a current area of interest upon which an annotation can be made. CSRS divides the document up into smaller chunks called nodes, each of which can be inspected on its own and comments made via new nodes linked to this one. InspeQ and InspectA are the least well supported in this area, since comments are completely separate from the source document, with only cut and paste facilities available to give a context to a comment. In essence this only gives the facilities that a text editor can supply. Only two tools support non-textual documents: AISA and TAMMi. Both allow annotation of graphical documents, the first via HTML and the second as PostScript. This lack of support for different document types is one of the major shortcomings of existing tools, and must be addressed if inspection support tools are to become a standard feature of software development support environments. Annotations linked to the area of the document to which they refer also appears to be a fundamental feature.

ICICLE, CSI, CAIS, AISA, Scrutiny, TAMMi, CSRS, InspectA and WiP all allow classification of annotations. The other tools only allow their creation or deletion. This limits

	ICICLE	CSI	InspeQ	Scrutiny	TAMMi	DCI	CSRS	CAIS	AISA	NI	InspectA	hyperCode	WiP
Linked Annotations	•	•		•	•		•	•	•	•		•	•
Defect Classification	•	•		•	•		•	•	•		•		•
Cross-referencing	•												
Automated Analysis	•												
Checklists		•	•		•			•			•		•
Supporting Material	•		•			•					•		•
Distributed Meetings		•		•		•							
Decision Support				•			•	•	•	•			
Data Collection	•	•		•			•	•					•

Table 3.2: Summary of features of existing on-line inspection tools.

the scope for collection of defect type metrics, although it still allows the overall number of defects to be measured. Classification of annotations is important for providing feedback on the software development process, in terms of the most frequently occurring defect types.

Individual Preparation Checklists are supported by InspeQ, which uses them to enforce a rigorous inspection by ensuring each item on the checklist is attended to by the inspector. In a similar vein, CSI has the concept of a criteria list which helps inspectors find and categorise defects, and this is also available in CAIS. TAMMi, WiP and InspectA also provide checklist support. As checklists are a fundamental feature of virtually all inspection types, it is surprising that less than half of all tools provide any support.

In terms of other supporting material, InspeQ supports the displaying of standards, while ICICLE can provide a browsing facility for manual pages like those provided in UNIX. InspectA can display a single source document, DCI can display a specification, while WiP can display various supporting documents. Once again, supporting documents are vital part of any inspection, yet they are overlooked by the majority of existing tools.

ICICLE is the only tool to provide any automatic defect detection. This is currently provided using the UNIX tool `lint` and ICICLE's own rule based system, which contains knowledge about C source code that can be used to detect such defects as coding violations. ICICLE is also the only tool to provide cross-referencing, but this is limited to C. Clearly there is much scope for research in this area.

Meeting Support To ensure that each inspector has spent sufficient time in preparation, CSRS can provide details on the amount of time spent on inspection by each inspector. This prevents inspectors misleading the moderator about their state of preparation. The checklists in InspeQ also perform this function. Scrutiny stores the percentage of document covered by each inspector, as well as the time spent by each inspector in both preparation and meeting. Such data is useful for controlling the inspection, and must be made easily available.

Support for distributed meetings is only relevant to the synchronous inspection tools. CSI uses Teleconf, which provides an audio channel only. Scrutiny also supports the use of an audio channel, in addition to its discussion and messaging facilities. DCI provides textual discussion via chat and e-mail. ICICLE lacks these facilities and is designed to be used when the inspection meeting takes place in one room with all inspectors present. InspeQ and TAMMi are designed for individual inspector use only, and lack any such facilities. Videoconferencing facilities are vital for allowing geographically distributed meetings.

Decision support is available through polls in Scrutiny, CSRS, CAIS and AISA. A voting

mechanism would seem to be desirable for an asynchronous inspection tool, since it provides a good way to reach a consensus, yet not all asynchronous tools support such a system. Even in a synchronous meeting, it may provide a useful means of issue resolution.

Data Collection ICICLE automatically gathers metrics on the number and type of issues raised, as well as their severity, as noted by the scribe during the inspection meeting. CSI and CAIS use a history log to record metrics. CSRS and Scrutiny have the most comprehensive metric gathering capability. CSRS has the ability to gather defect metrics, as well as fine-grained metrics on the amount of time spent by each inspector reviewing each node. Scrutiny has similar collection facilities, including the time spent in inspection and the coverage of the document achieved by each inspector. WiP collects the number of issues found and the total time spent in inspection, and can calculate the inspection rate and defect detection rate. Actual data collected is specific to each development environment, therefore an inspection support tool should be tailorable in terms of the data collected and the analysis performed.

3.3 Research Framework

Having investigated existing tools, a number of weaknesses were identified. It was decided to implement a prototype support tool to tackle these weaknesses. The first step was to provide the basic features required of an inspection support tool and also to tackle some fundamental omissions from existing support tools, namely:

- Support of any inspection process. Computer support should not be tied to a particular inspection method. Instead, the tool should be rigorous in its enforcement of the inspection process, but tailorable as to which process it enforces.
- Support of any document type. The system cannot be restricted to a single document type, such as ASCII. Instead it must provide an extensible system for supporting multiple types.
- An annotation mechanism, where annotations are linked to the area of the document to which they apply. Less than half of the existing tools provide such a mechanism, yet linking defects to the section of the document in which they occur would appear to be a basic requirement.
- Classification of annotations, allowing data on defect types to be gathered and used to pinpoint weaknesses in the software development process.

- Ability to display supporting documentation, such as checklists and standards, since these are important documents in the inspection process.
- A synchronous meeting mechanism allowing users to share data and vote on issues. This would allow the exploration of the effectiveness of an on-line meeting.

These features were identified as being vital to any inspection support tool, and would form the foundation of a comprehensive tool.

A common problem when developing new tools to support software development is lack of proper evaluation, and the area of tool support for software inspection is no exception. With this in mind, controlled experiments were planned to evaluate this research. The first would compare paper-based inspection with basic tool-supported inspection, using the prototype tool implementing the features described above. This study would provide a baseline for further research, and ensure there were no fundamental flaws with the concept.

A second version of the tool would then be developed. This would implement more advanced features concerned with enhancing performance or reducing effort during inspection, both for individuals and the team as a whole. Moving to a computer supported inspection gives an opportunity to provide more active support for finding defects, along with other process improvements. Features for this version would be based on weaknesses in existing tools and feedback from the first experiment. A second experiment would then be staged to compare paper-based inspection with advanced tool support, in an attempt to explore the effect of providing additional features.

Chapter 4

Supporting a Generic Software Inspection Process

This chapter describes work carried out to achieve the goals identified in the previous chapter. The main aim of this work is to provide support for all inspection models. This allows the most effective process for a given situation to be implemented, optimising the costs and benefits associated with the inspection. There are two possible solutions. The first is to make use of an existing technology, while the second concerns the exploration of a new approach. It was decided that the most appropriate approach was to derive a purpose-built process definition language. This language can be used as input to an inspection support tool, allowing support of multiple processes.

The chapter begins by considering workflow tools and general-purpose process modelling languages, two existing technologies which could be used to provide support for multiple inspection models. It then introduces Inspection Process Definition Language (IPDL), a language capable of describing all existing inspection processes. The use of IPDL to describe the Fagan inspection process is discussed in Section 4.1.4. (IPDL descriptions of the other seven processes described in Chapter 2 can be found in Appendix C.) A comparison of IPDL and other attempts at modelling software inspection processes can be found in Section 5.1.

Section 4.2 introduces ASSIST (Asynchronous/Synchronous Software Inspection Support Tool), a prototype system used to implement the research presented in this thesis. It discusses the execution of the IPDL implementation of the Fagan process and the facilities provided to users. This section also discusses other goals identified in the previous chapter, including support for multiple document types.

4.1 Inspection Process Definition Language

4.1.1 Implementation Technologies Considered

There are several existing technologies which could be used to provide support for multiple inspection processes. Two such technologies were considered: workflow management tools and process modelling languages.

Workflow Management Tools

Within an organisation, there can be three types of process [40]. Material processes are those concerned with physical components. Information processes are concerned with the creation, processing and management of data. Business processes are descriptions of activities, implemented as material and/or information processes, concerned with fulfilling a customer need or satisfying a contract. A workflow is the description of the sequence of steps in a business process, performed automatically and/or by human intervention. Defining a workflow allows that process to be understood, evaluated and modified. Workflow management systems are tools which support coordination and execution of tasks within a workflow, and its redesign.

Three types of workflow systems have evolved over the past decade or so, with increasing levels of sophistication. Initially, image-based systems were created to automate the flow of paper through an organisation. Paper documents were digitised and the workflow system used to route these documents in the appropriate manner [27]. Form-based systems were the logical progression: instead of digitised document, electronic forms containing machine readable data are routed around the organisation. This opens up the opportunity of automating tasks. Finally, coordination-based systems are designed to facilitate the making and fulfilling of commitments necessary for completion of work. As their name suggests, they aid the coordination of personnel within the process.

A wide range of commercial workflow tools are available. A recent comparison of some of these can be found in [25]. There are also a number of research prototypes. Since inspection is simply a process involving documents and people, workflow tools could be used to support it. Hence, a number of representative tools will be considered, with specific reference to their applicability to inspection.

Regatta [111] is a system which represents work as a network of requests. A request, representing a responsibility, is made from one person to another, and will have one or more options for the recipient of the request. The recipient can accept or decline the request. A policy is a set of requests which defines the process, and is represented using a visual process language. Each plan can be decomposed into a subplan, supporting abstraction of the process.

These processes do not need to be complete before they are activated, and can be edited as the process is followed. While the Regatta system could be used to represent the stages and personnel in an inspection process, and a description of the task to be performed at each stage, it does not represent the documents within the system. The language has the goal of being executable [112], but it is not clear how this is achieved, or how the appropriate tools could automatically be selected to support the process.

TriGS_{flow} [64] is an architecture for a workflow management system incorporating object-oriented and rule-based concepts and based on a commercial object-oriented database. A workflow is modelled as a number of activities, each of which is performed by one or more agents. Agents can either be automatic (i.e. some form of machine) or human (representing users). Activities performed by automatic agents are completely autonomous. Activities involving human agents can either be executed by an application with user interaction, or without any computer support. Activity nets (an extension of Petri nets) are used to specify control flow between tasks, defining execution order. Each agent has a worklist, and data flow between agents is modelled using orders, which insert data into an agent's worklist. Finally, the system can apply rules to activity ordering, agent selection and worklist management. This allows dynamic selection of activities and agents, and the automatic selection of items on worklists to be processed by automatic agents. This type of system could be used to implement inspection processes, however it is still very much a research prototype and hence not widely available. The combination of both activity nets and rules also makes process definition more complex than it might otherwise be.

A recent trend in workflow systems, in common with the newer inspection support tools described in Chapter 3 has concerned use of the World Wide Web [92]. The use of any standard WWW browser to access the system allows any user to make use of a workflow tool without installing dedicated software, essentially providing platform independence. The browser is also an interface which most users are already familiar with.

One Web-based research prototype is DartFlow [18]. It uses Java applets for the interface and agents to carry data and control information. When a user logs on to the system, their worklist is displayed. Clicking on a worklist item displays an HTML form, completion of which results in the generation of an agent to process the form. An example system demonstrating a system for opening bank accounts is presented, although it is not clear how easily DartFlow can be modified to implement other processes.

Another example is WebWork [91], a Web-based enactment system for the METEOR₂ workflow management system. Workflows are modelled as a set of tasks and the dependencies between them. The enactment system consists of task managers, application tasks and

a run-time monitor. Essentially, each task in the workflow is mapped to a task manager, an application task and a verifier. The task manager reads data from the previous task, prepares it for the current task, then invokes the application. Data is then collected and passed to the verifier, which ensures the task has proceeded correctly and selects the next task to execute. Tasks can be various types, including fully automated and those requiring human intervention. It would therefore be possible to use this system to model inspection processes, invoking the required tools at the appropriate time. Unfortunately, this type of WWW technology was not available when this research was being performed.

Fundamentally, workflow systems have been concerned with routing information and informing users of events, rather than supporting teamwork. They are usually used by many users company-wide [18], rather than the small teams which are fundamental to inspection. Steps within a workflow tend to be more finely-grained than those within an inspection process, usually consisting of simple decisions. Also, there is no standard for process definition, therefore a process defined in one workflow system cannot be used by another. Some tools allow supporting applications to be executed to help perform the task. Not every organisation has the same tools, however, so a process developed in one organisation may not be usable in another. Ideally, process definitions should be as widely applicable as possible. The technology used is becoming increasingly sophisticated, however, and it seems likely that the deployment of an inspection support tool based on such a system will become more feasible. A WWW-based system, such as WebWork, is certainly now a more viable platform. In the future, there may also be convergence on a standard for process definition.

As can be seen from the above descriptions, workflow systems have a process modelling element. Hence, the next section explores the possibilities for defining inspection processes presented by general-purpose process modelling languages.

Process Modelling Languages

The software development process is a key factor in the quality of delivered software [67]. Hence, there has been much research on modelling the process. Process models can be built to understand the nature of the process under scrutiny (and therefore improve it), and they can be used as a basis for automating the process. Since software inspection is a subprocess of software development, it is natural to investigate the applicability of process modelling, with a particular interest in the modelling languages available. Such a modelling language could be adopted for describing inspection processes and used as input to an inspection support tool.

A number of process modelling languages (PMLs) have been proposed, utilising various paradigms and approaches. McChesney [86] has developed a classification scheme for

process modelling approaches, a subset of which concerns the PML associated with the approach. Eight PML paradigms are identified: rule-based, imperative programming, object-oriented, AI/knowledge-based, Petri net-based, functional programming, formal specification and mathematical (or quantitative) modelling. Some of these are more widely used than others. Example languages from each paradigm are now briefly described.

Marvel [60, 61] is a process-centred environment based around a rule-based process engine. A particular environment consists of an object-oriented database containing process and product data, rules defining the behaviour of the environment and a set of tool envelopes. Each rule consists of three components. Preconditions are boolean expression which must be true before an activity is performed. The second component is the activity itself, which may invoke a tool, invoke another rule, or describe a task to be carried out by a human. Finally, postconditions are logical assertions which become true when the activity has been completed. There may be multiple postconditions reflecting the possible results of the activity. Forward and backward chaining of rules are used by Marvel to determine activities which can be performed automatically. Tool envelopes allow Marvel to invoke tools used in the process model. These envelopes specify the manner in which the tool is to be invoked and to return values indicating success or failure. Note that the integration of Marvel and the Scrutiny software inspection tool is discussed in Section 5.1.1.

APPL/A [109] is an imperative modelling language based on Ada. A traditional programming language was chosen as the basis for APPL/A as it provides basic control mechanisms, data definition facilities, executability and other features which are necessary for modelling processes. APPL/A extends Ada in a number of ways. It provides relation units, which represent relationships between objects in the process and provide data structures for representing process data. This data is shared and persistent. Trigger units represent logical threads of control which can react to events. These are generally used to propagate updates between relations, to send notifications of changes in data, and so on. Predicate units specify conditions on relations. They can be explicitly invoked or automatically invoked and are used to maintain consistency. A consistent state exists when all predicates on a relationship are satisfied. If one or more predicates are not satisfied then inconsistency occurs. Consistency management is further supported by a set of specific constructs.

EPOS [54, 23], on the other hand, implements SPELL, a persistent, object-oriented language which also makes use of rules. Processes are modelled as a typed network of tasks. EPOS types (including tasks and data entities) exist in a hierarchy. Types can be subclassed and augmented via single inheritance. A task type represents a step in the process and contains a script to be executed. This script is surrounded by preconditions and postconditions, which

can be either static or dynamic. Static conditions allow forwards and backwards reasoning without executing the script, while dynamic conditions allow the dynamic triggering of tasks. The system provides an execution manager which enacts the process. It uses the preconditions to determine when to enact a given task and then interprets the script associated with the activity.

GRAPPLE [49] makes use of the AI planning paradigm. In this paradigm, the system is expressed as a set of operators and a state schema (consisting of a set of predicates). These define possible actions within the system and the state of the system, respectively. A set of goals are also defined, represented by logical expressions involving the state predicates. A plan is a set of actions which achieve a goal given an initial system state. When applied to process modelling, operators allow the definition of processes and plans become data structures representing instantiations of the process. Each operator has a precondition and a primary goal, along with a set of side-effects. A set of sub-goals may also be defined, allowing the decomposition of complex processes. These operators are dynamically instantiated to generate plans which can be executed. The paradigm is aimed at emphasising the goals which must be achieved, rather than the actions which must be taken to achieve them.

The SPADE (Software Process Analysis, Design and Enactment) environment [4, 5] provides the SLANG (SPADE Language) process modelling language, based on Petri nets. A SLANG process model consists of a set of type definitions and a set of activity definitions. Types are defined in an object-oriented style, consisting of a hierarchy of subtype relationships. Activities are modelled by Petri nets with process data being represented by typed tokens of the net. Each activity is specified by a set of places, a set of transitions and a set of arcs. State is represented by an assignment of tokens to places. The occurrence of an event is modelled by a transition, where tokens are removed from the input places of the transition and added to the output places. Each activity is divided into an interface, which interacts with other parts of the process, and an implementation, which is hidden. Each activity can be composed of a number of sub-activities. Activities can be executed in parallel.

HFSP (Hierarchical and Functional Software Process) [110] is an example of a functional modelling system. A process is defined as a set of mathematical functions, each of which represents a process element with inputs and outputs. Each function defines the relationships between inputs and outputs, and can in turn be decomposed into subelements (with matching inputs and outputs). Decomposition continues until each process element maps to a single tool invocation or human operation. HFSP implements sequencing, iteration and concurrency of process elements and also allows communication between elements.

The formal specification language LOTOS [53] has been another avenue of research. A

specification consists of a number of processes, each of which is described by a behaviour expression. Behaviour is defined in terms of the sequence of events in which the process can participate. An event defines a synchronisation between processes. LOTOS specifications can describe the temporal ordering of these events, and can also model non-determinism and concurrency. Yasumoto *et al.* [123] described the use of LOTOS to model software development processes. The process is described in terms of the actions and order of primitive activities.

Finally, an example of quantitative modelling is the system dynamics approach of Abdel-Hamid and Madnick [1]. This technique simulates the system being modelled as a set of equations. When these equations interact, they provide feedback loops which simulate the dynamic nature of the system. The accuracy of the model depends on the form and parameters of the equations. The approach can be used to investigate the effect of altering process parameters, e.g. the number of developers involved.

When considering how they may be applied to inspection processes, it is worth considering how such a language will be used. Potential users of the language will include inspectors, moderators and inspection support tool developers, each of whom will have a different viewpoint and different requirements. Inspectors will have to follow processes defined. Moderators will use it to describe existing processes and to develop new processes. Tool developers will be concerned with implementing support for the language, providing adequate facilities for each phase in the process.

A list of requirements can now be formed. The language must be simple to allow processes to be easily written and modified. The language should be readily accessible and not require a significant learning effort. For example, industrial inspection training courses last a maximum of three or four days. It should be reasonable to teach the language within that timeframe alongside the myriad of other inspection issues. Furthermore, to be amenable to tool support, the language must be unambiguous, machine readable and executable. To achieve widespread support (i.e. be adopted by a number of tool developers) simplicity is also important. While the main purpose of this language would be to describe inspection processes for the application of tool support, it should also be simple enough to allow processes to be followed manually. For this reason, process descriptions should also be compact and readable, perhaps using an English-like syntax or a simple graphical notation. In terms of inspection elements, the language must primarily be able to represent people and documents. It must also represent the phases of the inspection and their execution order, with scope for concurrency.

Quantitative modelling can be immediately discarded, since the model only describes numerical aspects of the process. While the use of preconditions in AI and rule-based paradigms allow automatic transitions between states, it is not clear how useful this is for inspection

processes, since the moderator generally makes the decision as to when to move from one phase to the next, and the grounds for making the decision vary from inspection to inspection. In a non-computer supported environment this feature is obviously irrelevant. In fact, the functional, AI, and rule-based paradigms rely on computer support. Hence, their value in a non-computer supported environment is debatable.

Another concern is ease of understanding. Heimbigner [48] compares rule-based and procedural languages. He concludes that while rule-based languages are flexible and can handle unexpected events, it is difficult to understand the process flow. Similar comments can be made concerning functional languages. On the other hand, procedural languages are good for describing normal execution but are less flexible and cannot easily handle unanticipated events. This is not a disadvantage where inspection is concerned, because the process is linear with no optional paths. The only non-normal event is an early termination of the process, which can be catered for easily.

Complexity is another issue. Formal specification languages have a reputation for being difficult to use in general. Implementing such a language as part of an inspection support tool is a non-trivial task. Similar arguments can be applied to functional, rule-based and Petri net-based languages. Object-oriented languages have their benefits in terms of reuse and the grouping of related object by inheritance. Again, however, there are problems in terms of understanding such languages [79] and the proper implementation of object-oriented features is non-trivial.

Extensive type definition facilities, like those available in SLANG, are not required. From the review of processes in Chapter 2 it is apparent that inspection has well-defined types, in terms of documents and participants. Hence, type definition facilities are a complication which can be avoided.

It clear from the descriptions above that most PMLs are highly expressive, allowing a multitude of processes to be defined. For the desired application, however, such flexibility is not required. Although a flexible notation is required to allow all inspection variations to be described, there is also a well-defined outline process which needs to enforced. There are other disadvantages in choosing an existing PML. Most PMLs appear to have steep learning curves and require extensive programming skills from their users. It is preferable to minimise the amount of skill required by the user, to allow widespread use of the language. Choosing an existing PML also requires either writing a process engine afresh or obtaining from the authors of the language. The first would be a difficult and time-consuming task for many of the complex languages available, while the second would unnecessarily restrict the choice of development environment.

It was decided, therefore, to define a new language specific to software inspection. This avoids taking an existing language and applying artificial constraints. Furthermore, a language with such constraints should be easier to parse, with reduced complexity. A language designed specifically to model inspection processes should also simplify the task of writing processes, since information about the basic form of an inspection process is already encoded in the language. As simplicity and ease of understanding are paramount, the procedural paradigm was chosen.

One decision which remains to be made is the choice between a textual and graphical notation. A graphical notation can be more user-friendly and understandable. The supporting environment is more complex to implement, however. Furthermore, a mapping between the graphical semantics and an executable representation is required to allow process enactment [22]. Hence it was decided to initially implement a textual notation. A graphical representation could then be derived if required.

4.1.2 Derivation of Generic Process

To design a language capable of describing inspection processes, the first step is to derive a generic inspection process. This process should describe the essence of inspection, yet provide flexibility in that each inspection variation can be adequately expressed. The process was derived by examining the eight inspection types described in Chapter 2.

Initial Process Derivation

The derivation of a generic inspection process begins with the observation that every inspection has three major stages: **Organisation** (deciding on participants, timing and other details), **Detection** (performing the actual inspection and finding defects) and **Completion** (fixing the defects and checking the work done). These three broad stages are present, to a greater or lesser extent, in all the inspection methods described in Chapter 2. The first and last stages vary only slightly between various inspection methods, while the major variations appear during the detection stages.

Organisation Activities

The earliest inspection phase described is the *entry* phase proposed by Gilb and Graham [41]. This phase ensures that specific criteria are met *before* the inspection starts, reducing the chances of a wasted inspection. This becomes the first inspection phase. This phase is optional depending on the actual inspection type.

The next phase is some form of *planning*, sometimes known as set-up. At this point the moderator has to organise the numerous details of an inspection. This ranges from choosing and inviting participants, preparing documents and support material, and deciding how to split the target material. Other typical activities include distributing inspection material and assigning roles. This phase is an explicit part of some inspection processes, particularly that of Humphrey and of Gilb and Graham, but not of others. Hence, the phase is defined to be optional.

The final activity during organisation is *overview*, present in numerous methods. This phase is multi-purpose, with many possible activities. A major event is usually a presentation on the document by the author. This phase is optional.

Detection Activities

The next step is to examine the detection stage in more detail. A traditional inspection generally has two phases here: an individual phase and a group phase. The processes defined by Gilb and Graham, Fagan and Humphrey all have this type of arrangement, and so do the multiple inspector phases of phased inspections. An obvious first step is to assume that the detection may have two types of phases: individual and group. This can be generalised further, however, by saying that detection will consist of one or more meetings. Meetings can then be categorised according to their timing (synchronous or asynchronous), objective (examination, detection or collection), the number of participants and whether data is shared between participants during or after the meeting (public and private visibility, respectively). For example, individual preparation in Fagan inspection is simply an asynchronous meeting with four or five participants, where the data created is kept private to each participant. On the other hand, the inspection meeting itself is a synchronous meeting where data is available to all participants. These two meetings represent the detection activities of Fagan's inspection and it can easily be seen that the approach is similar for both Humphrey and Gilb and Graham-type inspections.

Now consider phased inspections. A single inspector phase is simply a meeting with only one person in attendance. A multiple inspector phase is similar to a Fagan or Humphrey inspection. So the entire phased inspection can then be described as a series of generic meetings. Active Design Reviews consist of a single individual phase followed by multiple group phases, each with different participants. This can also be modelled as a series of meetings.

Next, consider the FTArm asynchronous process. Private and public review can be described by two generic meetings, however, there is also a consolidation meeting where the moderator must decide if a synchronous group meeting should be held. This *consolidation* step can be generalised to be a decision step where the need for a further meeting (of any

type) can be decided. This allows the modelling of the optional group review meeting in FTArm, and also provides further flexibility to allow the specification of optional meetings in any inspection process.

Finally, as was shown in Chapter 2, one of the most radical ideas for performing inspections involves the use of two or more independent teams. These teams will inspect the same material, possibly using independent inspection methods, at the same time. A requirement of this technique is the presence of a *collation* phase to assimilate the results from the independent inspections. To increase flexibility, multiple collation phases are allowed.

Completion Activities

The last stage is perhaps the most well-defined. Essentially, there are two objectives here: defects found during the inspection must be corrected, and the changes which have been carried out must be checked. These two objectives are achieved in the *rework* and *follow-up* phases respectively.

During the rework phase, the author of the document tackles each defect found during the inspection. This phase is called the *edit* phase by Gilb and Graham and they also use this phase to assign a final classification to each defect, unlike Fagan inspection where this is performed at the inspection meeting. This phase is defined to be optional since defects in the product do not have to be repaired. Instead, it may be more cost-effective to apply the results of the inspection to process improvement. Rework is also not required when the inspection is used for training or education.

With the changes being made to the document, it is now usually the moderator's duty to ensure that the changes have been carried out satisfactorily, by means of a follow-up activity. At this point the moderator must also decide whether or not the document should be reinspected. This will depend on the extent and type of defects found. The average number of defects found per page of document should also be compared with a historical figure for this type of document. An unusually high or low value may indicate an extremely defective document or an ineffective inspection, respectively. In either case the document should probably be reinspected. The follow-up is not defined for all inspections, and is therefore optional.

The next possible phase is *exit*, proposed by Gilb and Graham. This phase is intended to validate the inspection by ensuring that criteria such as checking rate and defect density have been met. As with the entry phase, it is assumed that some form of exit criteria are set and met, although these may be implicit.

The final phase which must be considered is metrics collection and analysis. Although not a distinct phase in any of the processes described, collection and analysis of such data is

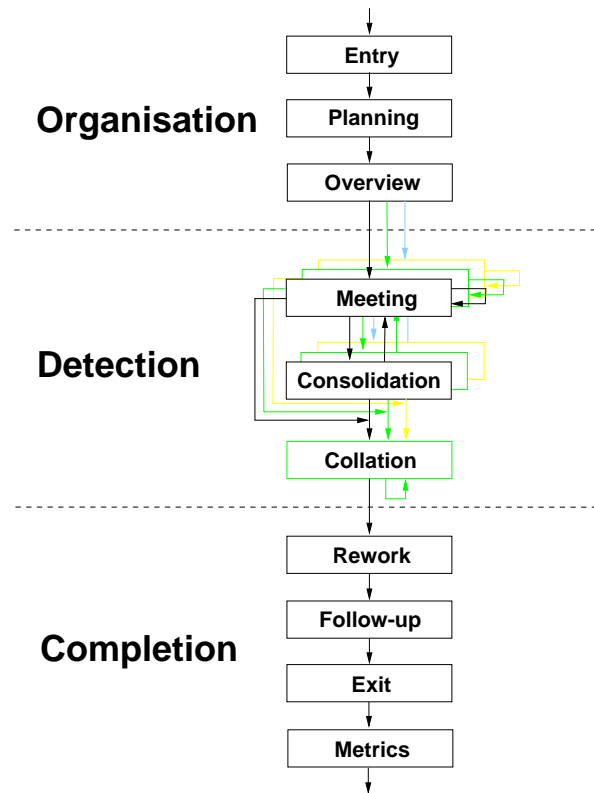


Figure 4.1: The generic inspection process.

deemed to be an important part of inspection. For this reason, it is proposed as a final explicit phase. The phase consists of preparing a report containing the appropriate measures of the inspection, and is optional. Figure 4.1 shows the complete generic inspection process.

Inspection Participants and Documents

Having derived the process, possible participants can now be decided, along with the possible resources required at each phase and the products generated by the inspection. Starting with the people involved, the key participant is the *moderator*, whose task is to plan and coordinate the entire inspection. The moderator will select and invite other participants, ensure that the required documentation is available and up-to-date, and moderate any group meeting that may be held. The moderator is sometimes referred to as the *leader*, and is required in every inspection.

The next participant to be considered is the *author*. As the person responsible for the document under inspection, the author usually has two main tasks: to brief other inspection participants on the document and to fix any defects found during the inspection. The author

can also be known as the *producer*.

Most participants in an inspection are those whose main responsibility is finding defects: the *inspectors* (known variously as *reviewers* or sometimes *checkers*). A typical inspection will require the services of two or more inspectors. It is therefore required that the number of possible inspectors is not constrained in any way. Given the assignment of reviewers to specific defect types that is practiced in ADRs, and a similar scheme using scenarios proposed by Porter [99], the ability to assign *responsibilities* to inspectors is also required. The responsibilities will usually indicate a set of inspection aids which would assist inspectors in finding their assigned types of defects.

Further roles can be defined for two of the inspectors if a traditional synchronous group meeting is held. The *recorder* (or *scribe*) is tasked with making a note of defects raised at the meeting. This master defect list is passed to the author for rework. The optional role of *reader* involves guiding the pace of the meeting and possibly paraphrasing the document.

The final participant is only relevant to an N-Fold inspection. The *coordinator* is tasked with coordinating the entire inspection, much like a moderator, but dealing with multiple inspections rather than just one. The coordinator will typically only interact with the moderator of each inspection team involved, collating the results from each inspection. Therefore the coordinator is only involved in the entry, planning and overview phases of the organisation stage, the collation phase of the detection stage, and the follow-up and exit phases of the completion stage.

From the process descriptions in Chapter 2, it can be seen that documents used and produced during each phase can be divided into several generic types:

- **Product** The document undergoing inspection.
- **Report** A report simply details the outcome of a phase, or of an entire inspection. It is usually completed by the moderator.
- **Inspection Plan** This is created during the planning phase and is the definitive prescription of the inspection process and the people who will follow it.
- **Source** A document used to produce the document undergoing inspection, for example, the design document for a section of code.
- **Detection aid** A document which assists the inspector with finding defects. This includes checklists which help to ensure adequate coverage of both the document and of common defect types, items such as scenarios which assign responsibilities to each inspector and questionnaires.

Phase	Timing	Participants	Documents available	Documents produced
Organisation				
<i>Entry</i>	-	Coordinator Moderator	Entry criteria	
<i>Planning</i>	-	Coordinator Moderator		Inspection plan
<i>Overview</i>	S	Coordinator Moderator Author Inspector	Product	
Detection				
<i>Meeting</i>	S,A	Moderator Author Inspector	Product Sources Checklists Standards Inspection plan Individual lists	Individual lists Master list Report
<i>Consolidation</i>	-	Moderator	Individual lists Master list	Report
<i>Collation</i>	S,A	Coordinator Moderator	Master lists	Collated list
Completion				
<i>Rework</i>	-	Author	Product Collated list Master list	Report
<i>Follow-up</i>	S	Coordinator Moderator Author	Product Collated list Master list	Report
<i>Exit</i>	-	Coordinator Moderator	Exit criteria	

Table 4.1: Summary of generic inspection phases and the possible timings, participants, resources and products. Timing is either synchronous (S) or asynchronous (A). Possible documents available at each phase, along with documents that may be produced during each phase, are also listed

- **Standard** The product will usually have to conform to a set of standards. These standards can be used for compliance checking during an inspection. This type of document describes the process to be followed at each phase in the inspection.
- **List** This is a generic document type for lists of comments produced by participants. These comments may concern defects in the product, process improvement suggestions, change requests for non-product documents and so on.
- **Criteria** Inspection entry and exit criteria may be required. Entry criteria ensure the inspection is not wasted on unsuitable material while exit criteria are used to ensure the inspection has been carried out correctly. Such criteria include inspection rates and estimated percentage of defects found.

Of course, some of these are only useful during certain phases, therefore the notation should only provide relevant alternatives at each phase. At the same time, the flexibility of the notation should not be unnecessarily limited. Finally, although inspection is usually concerned with finding defects in the product, the process should also allow defects in supporting materials, such as standards and procedures, to be noted, as required by Gilb and Graham [41].

Table 4.1 summarises the roles, resources and products relevant to each inspection phase. Note that the coordinator is only present in an N-Fold inspection, and that the roles of reader and recorder may only be assigned during a synchronous group meeting.

4.1.3 IPDL Definition

Having developed a generic inspection process, a notation to adequately describe existing inspection processes was derived. This notation is the result of giving due consideration to the issues described in Section 4.1.1. The following sections describe the grammar of the language using a Backus-Naur style of notation. In this notation, a phrase in *italics* is non-terminal, while words in `typewriter` style indicate language keywords. The “`::=`” operator is used to show expansion of non-terminal clauses. A plus sign (“+”) indicates one or more instances of a given clause, while *opt* indicates that the clause is optional. Finally, square brackets indicate alternatives, with the alternatives separated by vertical bars.

Structure of Process Description

The description of a software inspection consists of two parts. The first part contains declarations listing the participants and their roles, along with the documents which will be used and created during the process. The second part describes the process itself, split into the

```

software_inspection ::= inspection inspection_name
                        declarations
                        process
                        end

inspection_name ::= string
declarations ::= declarations
                  document_declarations
                  responsibility_declarationsopt
                  participant_declarations
                  classification_declarationopt
                  end

process ::= process
             organisation_process
             detection_process
             completion_process
             end

organisation_process ::= entryopt planningopt overviewopt
detection_process ::= [detection|n_fold]
completion_process ::= reworkopt follow_upopt exitopt metricsopt
string ::= “’ character+ “’”
character ::= Any printable character or white space.

```

Figure 4.2: Initial process definitions.

```

document_declarations ::= documents document_definition+ end
document_definition ::= document_name document_type
document_name ::= identifier
identifier ::= non_whitespace_character+
non_whitespace_character ::= Any printable character which is not white space.
document_type ::= [product|report|source|standard|
                    list|criteria|plan|detection_aid]

```

Figure 4.3: Document definitions.

three stages derived in Section 4.1.2. There is also a facility for naming the inspection. The initial definition of the inspection is that given in Figure 4.2. The keywords *inspection* and *end* are used to delimit the description. *inspection name* is simply an arbitrary string surrounded by quotes. The declaration section consists of inspection documents, responsibilities, participants and classification scheme. Each of these is described in the following sections. The inspection process itself mirrors the process described earlier, consisting of the three major phases of organisation, detection and completion. The initial definitions of these are also presented and each will be described in more detail later.

```

targets ::= targets document_name+
inputs  ::= inputs document_name+
outputs ::= outputs document_name+

```

Figure 4.4: Document inclusion clauses.

```

responsibility_declarations ::= responsibilities responsibility_definition+ end
responsibility_definition  ::= responsibility_name requires {document_name+}opt end
responsibility_name       ::= identifier

```

Figure 4.5: Responsibility definitions.

Inspection Document, Participant and Responsibility Declarations

The first section within the declaration part of the description describes all documents which are available and created during the entire inspection, and is defined in Figure 4.3. This section simply defines names for each of the documents to be used within the inspection. When the inspection is instantiated and run, part of the planning task is to associate the real inspection documents with each document defined. The format of such documents as criteria lists and reports is therefore left to the implementation. Note that the language words are reserved and are not available for use as document identifiers, or any other user-defined objects.

The definition of each phase of the inspection will require the documents present and created during that phase to be defined. This will be achieved by the use of several clauses, defined in Figure 4.4. Any document name appearing in these clauses must be declared within the document declaration section. The first of these introduces *target* documents, which may be of any type described above and are the actual documents being inspected. There is no constraint on the type since it is not unreasonable to allow standards, checklists and other supporting material to be inspected at the same time as the product. The *inputs* keyword indicates documents which are made available to this phase, and may also include documents of any type. The *outputs* keyword indicates the documents created or edited during the phase, which may either be reports, plans, criteria lists, detection aids or lists. During each phase, participants will only have access to documents defined for that phase using these clauses.

The next part of the description is concerned with describing the participants involved with the process and the responsibilities which they may be assigned. A common inspection practice is to assign reviewers responsibility for certain defect types, thus hopefully improving the coverage and effectiveness of the inspection. This responsibility usually comes in the form of a checklist or other defect finding aid. Figure 4.5 shows how a responsibility may be defined in terms of documents. Each responsibility has a name and a list of documents associated with

```

participant_declarations ::= participants participant_definition+ end
participant_definition ::= participant_name is
                             role
                             participant_listsopt
                             responsibility_assignmentopt
                             end
participant_name       ::= identifier
participant_lists     ::= lists document_name+
responsibility_assignment ::= responsibility responsibility_name+
role                  ::= [coordinator|moderator|author|inspector]

```

Figure 4.6: Participant definitions.

it. These will usually be checklists or other detection aids, but may also include standards or any other document type. These documents should be made available to the appropriate participant by the support tool.

The definitions for inspection participants are shown in Figure 4.6. There will usually be several constraints on the selection of participants. Typically, there must either be one moderator, *or* one coordinator and several moderators, depending on the type of inspection. This constraint is not part of the language because it may unnecessarily limit its flexibility. Instead, it is left to the implementation to enforce such restrictions as required. Zero or more authors may be declared, to allow maximum flexibility. Any number of inspectors may also be declared. The `lists` subclause indicates the document which this participant will use to record defects, change requests or other such items previously discussed when considering the list document type. Finally, the responsibility names used must be previously declared in the responsibility declaration section above. Note that any single person may have more than one role or responsibility. During each phase, any participant with no defined responsibility will only be given access to documents defined in that phase, i.e. those generally available. One such possible document is a general checklist used by everyone.

The participants description is simply a list of people involved and the names of their roles. Note that the participant list is *not* a list of the real people involved; it simply lists the names of the “characters” in the inspection and the roles they will play. For example, a Fagan inspection may have:

```
Moderator is moderator
```

indicating that the person called Moderator is executing the moderator's duties. Contrast this with a Gilb and Graham-type inspection, where the person carrying out the task of the moderator is known as the leader:

```
Leader is moderator
```

```

classification_declaration ::= classification classification_name
classification_name      ::= string

```

Figure 4.7: Item classification clauses.

```

participant    ::= participant participant_name
participants  ::= participants participant_name+

```

Figure 4.8: Participant inclusion clauses.

This convention allows the naming of roles in any way required, allowing us to use terms which coincide with any inspection practice. This also allows many people to take the same role, for example to have multiple inspectors, each with a unique responsibility:

```

Inspector_MF is inspector
    lists MF_defects
    responsibility Missing_Functionality
end
Inspector_AM is inspector
    lists AM_defects
    responsibility Ambiguity
end

```

Figure 4.7 shows the item classification clauses. These are used to optionally specify the classification scheme to be used for list items. The name of the classification scheme used must be known to the implementation (e.g. “Fagan”). Classification names are not part of the language definition. The implementation is also responsible for the manner in which items are classified. For example, the number of classification levels may vary between implementations.

Finally, for each phase of the inspection the participants required to be present must be indicated. This is achieved with the two definitions shown in Figure 4.8. The first definition indicates that only one participant should be present, while the second indicates the possibility of more than one person taking part. The use of these definitions will be shown along with each phase, but any participant name used within these clauses must have previously been declared in the participants declaration section.

The Organisation Process

As seen in Section 4.1.2, the organisation stage may have three phases: entry, planning and overview. Figure 4.2 shows the order in which these phases must occur, and indicates that they

```

entry ::= entry phase_name
         participant
         targetsopt
         inputsopt
         outputs
         end
phase_name ::= string

```

Figure 4.9: Entry phase definition.

```

planning ::= planning phase_name
              participants
              targetsopt
              inputsopt
              outputs
              end

```

Figure 4.10: Planning phase definition.

are optional. Each of these phases is defined in turn, starting with the entry phase, shown in Figure 4.9. This defines a name for the entry phase and indicates that only a single participant is required during this phase, usually either the moderator or the coordinator, depending on the type of inspection. At least one output document must be defined, usually a criteria list. A report detailing the outcome of the phase is also usually defined. Other documents may also be present using the targets and inputs keywords

The next phase defined is planning, shown in Figure 4.10. Again, this phase may be named according to the method being described. Although planning will generally involve a single moderator, multiple participants must be allowed for, especially in the case of an N-Fold inspection, where the cooperation of several moderators and a coordinator may be required to form the inspection plan. In this case, the coordinator should have overall control over the planning stage, while the other participants can provide input. With multiple participants there must be either a single moderator or a single coordinator. Again, this constraint is left to the implementation. At least one output must be defined (usually a plan), but others outputs, along with targets and inputs, may be defined.

The final organisation activity is overview, shown in Figure 4.11. This phase requires the definition of the participants involved, the format of the meeting, either local (same place) or distributed (different place). The presenter is the person who carries out the briefing; this is usually the author. The overview phase is optional.


```

overview ::= overview phase_name
            location [local|distributed]
            participants
            presenter participant_name
            targets
            inputsopt
            outputsopt
            end

```

Figure 4.11: Overview phase definition.

```

detection ::= [meeting_phase consolidation_stepopt]+
meeting_phase ::= [multi_meeting|single_meeting]
multi_meeting ::= parallel phase_name
                  single_meeting
                  single_meeting+
                  end
consolidation_step ::= consolidation meeting_phase

```

Figure 4.12: Detection stage definition

The Detection Process

When defining the detection activities in Section 4.1.2, it was asserted that there would either be a single detection activity, or an N-Fold activity. This is shown in Figure 4.2. A single detection activity was defined to consist of at least one meeting phase, possibly interspersed with consolidation steps. At this point the possibility of having several parallel meetings is also introduced to provide extra flexibility. This allows subsets of the team to meet separately. A consolidation step begins with a consolidation meeting, where it is decided if a further meeting is required. This is followed by the definition of the optional meeting. The definition of *detection* is shown in Figure 4.12.

A meeting is defined to be a phase with one or more participants who may meet synchronously or asynchronously, and whose discussion may be private or public. The meeting may have one of three objectives: examination, defect detection, or defect collection. The assignment of roles during the meeting must also be allowed. Finally, the documents produced and used in the meeting must be defined. The definition of a meeting is shown in Figure 4.13.

The definition starts with the keyword *meeting*, followed by the meeting name. The objective, timing, location and visibility are then set, along with the maximum duration of the meeting in minutes. The implementation should use the duration to help guide the moderator during the meeting. This is followed by a list of all meeting participants, as defined earlier. The roles of reader and scribe may be assigned. If no reader is specified, then it is assumed that any participant can guide the meeting (such as in a Humphrey-type inspection where the

```

single_meeting ::= meeting_phase_name
                   objective [examination|detection|collection]
                   timing [synchronous|asynchronous]
                   location [local|distributed]
                   visibility [public|private]
                   durationopt
                   participants
                   rolesopt
                   targets
                   inputsopt
                   outputsopt
                   end
duration       ::= duration_integer

```

Figure 4.13: Meeting phase definition.

```

roles           ::= roles_role_assignment+
role_assignment ::= participant_name is meeting_role
meeting_role   ::= [reader|scribe]

```

Figure 4.14: Role definition.

document is not paraphrased). If the scribe is not specified then the moderator should be given that role by default. The roles are followed by target documents, inputs from previous phases (such as lists) and outputs generated during this meeting (such as reports). All documents are optional except for target documents. The role assignment section is defined in Figure 4.14. Only the roles of Reader and Scribe are defined.

The consolidation phase may follow any meeting, and is used to decide on the need for a further meeting to resolve any remaining issues. The definition is shown in Figure 4.15. Again, the phase may be named, and this is followed by the single participant who will perform the consolidation (usually the moderator). The target documents and input documents to this phase are then specified, which generally consist of the product and one or more lists, respectively. Finally, at least one output must be defined: this is usually a report.

The alternative to a single detection activity is to have multiple, parallel detection activities with a collation stage, i.e. N-Fold inspection. To increase flexibility, there is the possibility

```

consolidation ::= consolidation_phase_name
                   participant
                   targets
                   inputs
                   outputs
                   end

```

Figure 4.15: Consolidation phase definition.

```

n_fold ::= n_fold phase_name
           fold
           fold+
           collation+
           end
fold    ::= fold phase_name
           detection
           end

```

Figure 4.16: N-Fold stage definition.

```

collation ::= collation phase_name
              timing [synchronous|asynchronous]
              location [local|distributed]
              participants
              roles
              targets
              inputs
              outputs
              end

```

Figure 4.17: Collation meeting definition.

of holding more than one collation meeting. The definition is given in Figure 4.16. As usual, the phase may be named. The definition will then consist of two or more detection activity definitions, as described above, surrounded by the keywords *fold* and *end*, along with one or more collation meeting definitions.

The collation meeting definition is shown in Figure 4.17. For each collation, a number of participants can be listed, usually several moderators along with the coordinator, one of whom must be nominated scribe with a role definition, another of whom may be nominated reader. Inputs will generally consist of a collected list of defects from each inspection. The output will usually be a single master list of defects for the entire inspection, but reports may also form outputs from this phase. Several collation meetings may take place, to allow for the possibility of the coordinator meeting with a subgroup of moderators. In this case, an input to subsequent meetings should be the collated lists of defects from previous meetings.

The Completion Process

The completion process consists of four activities, as shown in Figure 4.2: rework, follow-up, exit and metrics, all of which are optional.

The rework phase is defined in Figure 4.18. Although rework is generally carried out by the author, the possibility of another participant performing rework is catered for. This may occur if the author is not part of the inspection team, or is otherwise unavailable. Various

```

rework ::= rework phase_name
           participant
           targets
           inputs
           outputsopt
           end

```

Figure 4.18: Rework phase definition.

```

follow_up ::= follow_up phase_name
               participant
               targets
               inputs
               outputs
               end

```

Figure 4.19: Follow-up phase definition.

documents may be made available during this phase. Target documents are always required, with the implementation having to provide some means of editing these documents. Input documents will typically consist of one or more lists. The output of the phase may consist of one or more reports, or other documents as required.

The next phase is follow-up, involving checking the work performed in rework, and is defined in Figure 4.19. Only one person should perform follow-up: this is usually the moderator (or coordinator), but there is the possibility of another participant performing this task. A target document is always required, and other input documents (usually a list of defects) must also be present. Finally, the defined output is one or more reports.

Next is the optional exit phase, defined in Figure 4.20. This is similar to the entry phase in that it defines one or more output documents, usually lists of criteria which must be met. A report detailing the outcome of the phase may also be appropriate. Input and target documents may also be defined. One single participant is involved in this phase: this is either the moderator or the coordinator, depending on the inspection type.

Finally, the metrics collection and analysis phase is shown in Figure 4.21. This follows the format of other phases. The main difference is the `data` subclause. This is used to indicate the measures which must be supplied by the tool for this phase. Each measure consists of its name, an optional participant name for whom this measure applies, and an optional phase name which states which phase that particular measure is to be taken from. For example, to collect the number of list items produced by the participant `Moderator` during the phase `'Preparation'`, the following might be used:

```

data

```

```

exit ::= exit phase_name
      participant
      targets_opt
      inputs_opt
      outputs
      end

```

Figure 4.20: Exit phase definition.

```

metrics ::= metrics phase_name
         participant
         data
         targets_opt
         inputs_opt
         outputs
         end
data     ::= data measure+
measure  ::= identifier participant_name_opt phase_name_opt

```

Figure 4.21: Metrics collection phase definition.

```
list_items Moderator 'Preparation'
```

Other metrics, such as the length of the product, are not specific to a single phase or a single participant and do not require to be specified. No measures are defined in IPDL. It is assumed that the measures available will depend on the implementation, or the organisation performing the process.

4.1.4 IPDL Example - Fagan Inspection

An IPDL description of the Fagan code inspection process is shown in Figure 4.22. The description starts by titling the inspection “Fagan Code Inspection”. The declaration section first of all lists all documents used and created in the inspection. The `Master_plan` is the definitive guide to the inspection. Although not explicitly mentioned by Fagan, it is assumed that such a plan must be prepared for the inspection. `Code` is the document under inspection. `Design` is the source document from which the code is derived. This is followed by the declaration of six defect lists, one for each participant and a master list which will contain the defects logged by the entire team. Finally, two reports are declared. One is used to detail the outcome of the inspection meeting, while the other will contain the moderator's findings from the follow-up phase. The declarations section also defines the inspection participants. Five participants are declared: the moderator, author and three inspectors.

The process section defines the six phases of a Fagan inspection. The `Planning` phase simply involves the moderator creating the master plan for the inspection, which details the

```

inspection 'Fagan Code Inspection'
  declarations
    documents
      Code          product
      Design        source
      Defects1      list
      Defects2      list
      Defects3      list
      Defects4      list
      Defects5      list
      Master_defects list
      Meeting_report report
      Follow_up_Report report
      Master_Plan   plan
    end
  participants
    Inspector1 is inspector
      lists Defects1 end
    Inspector2 is inspector
      lists Defects2 end
    Inspector3 is inspector
      lists Defects3 end
    Moderator is moderator
      lists Defects4 end
    Author is author
      lists Defects5 end
  end
end
process
  planning 'Planning'
    participants Moderator
    outputs Master_Plan
  end
  overview 'Overview'
    location local
    participants
      Moderator
      Author
      Inspector1
      Inspector2
      Inspector3
    presenter Author
    targets Code
  end
  meeting 'Preparation'
    objective examination
    timing asynchronous
    location local
  end
end
visibility private
participants
  Moderator
  Author
  Inspector1
  Inspector2
  Inspector3
targets Code
inputs Design
end
meeting 'Inspection'
  objective collection
  timing synchronous
  location local
  visibility public
participants
  Moderator
  Author
  Inspector1
  Inspector2
  Inspector3
roles
  Inspector1 is reader
  Inspector2 is scribe
targets Code
inputs Design
outputs
  Master_defects
  Meeting_report
end
rework 'Rework'
  participant Author
  targets Code
  inputs
    Master_defects
    Design
  end
follow_up 'Follow-up'
  participant Moderator
  targets Code
  inputs
    Master_defects
    Design
  outputs Follow_up_Report
end
end
end

```

Figure 4.22: An IPDL description of the Fagan inspection process.

actual participants involved in the inspection and the documents to be used. The Overview is defined to involve all participants and involves the author presenting the code to be inspected. The process then moves into the first of two detection phases. The Preparation phase involves all participants individually examining the document, hence the objective of the phase is examination, the work is carried out asynchronously, and all data created by participants remains private. The target of the phase is the product (i.e. Code) and the source is the design document. Individual defect lists are implicitly available. The next phase is the inspection meeting, again involving all participants. This time the meeting occurs synchronously, and all data is made public. The objective of the phase is the collection of defects into a master defect list, which is an output of the meeting, along with a meeting report. Two roles are defined

during the meeting: the reader and the scribe, which are assigned to Inspector1 and Inspector2 respectively. The Rework phase involves the author taking the code and the master list of defects and performing the required fixes. This work is checked in the Follow-up phase by the moderator, again using the code and the master list of defects, who also produces a report on the follow-up.

4.1.5 Conclusions

IPDL has been derived with the intention of allowing inspection processes to be easily communicated, and to be used as input to an inspection support tool. Its simplicity should also allow its use in non-tool supported environments IPDL was designed to be simple to use when defining new processes, and to provide relatively short definitions. This is achieved by using a language with a much higher level of abstraction, where the emphasis is on the implementation providing much more knowledge about the type of phases involved, hiding these details from the user. On the other hand, this does mean that IPDL is more restricted in the processes which can be defined. As ever, there is a trade-off between flexibility and simplicity, and it was decided that a simple, easy to use language would be more generally accepted.

It should be borne in mind that, with the exception of asynchronous inspection (which is inherently tool-based), IPDL represents paper-based processes. The act of introducing tool support may alter the way in which such processes are performed. The facilities available will vary from tool to tool, also having an effect.

4.2 Introduction to ASSIST

The first version of ASSIST [69] was implemented with three goals in mind: to demonstrate IPDL, to investigate mechanisms for providing support of any document type, and to provide a means for comparing basic tool-supported inspection with paper-based inspection. The first version was capable of executing any process written in IPDL, ensuring that the process is followed precisely and that the inspection participants are provided with the correct materials and tools at each stage of the inspection. This allows ASSIST to perform inspections with any number of people on any number and type of documents.

ASSIST is implemented in Python [68], an interpreted, object-oriented language. It is based on a client/server architecture, with a central server storing all inspection data, documents and personnel information. The client provides the user interface to the system, allowing users to modify, store and compile processes, enter personnel and document data, and to actually perform an inspection. This architecture allows distributed inspections to be easily

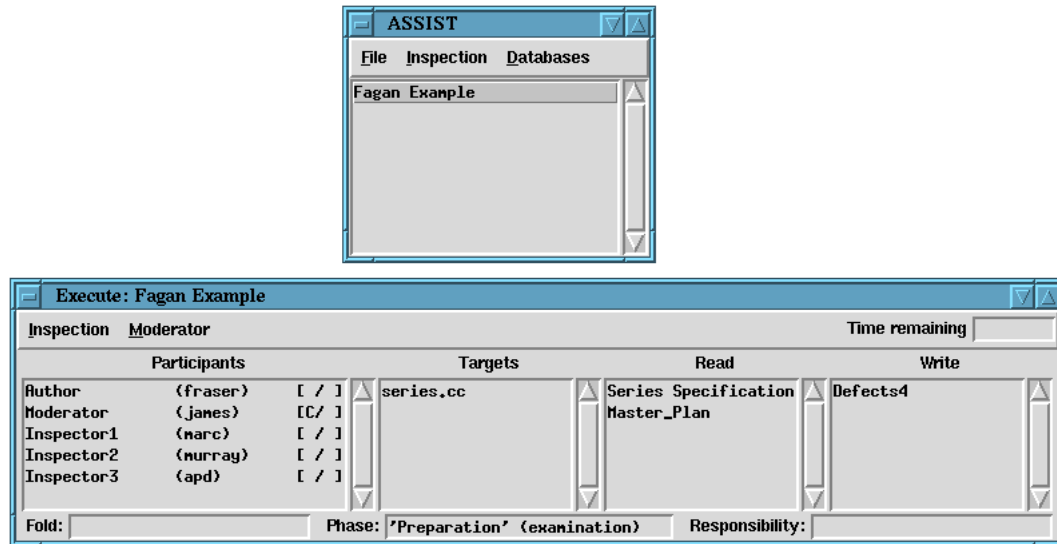


Figure 4.23: Joining the inspection.

held. Currently, a World Wide Web-based implementation is a more generic solution, allowing anyone with a standard WWW browser to access the system. At the time of the initial design of ASSIST, however, WWW technology was not mature enough to support such a system.

4.2.1 Using ASSIST to Execute the Process

To use an IPDL process with ASSIST, the process definition is first entered into ASSIST where it is compiled ready for use. When a new inspection is started, ASSIST loads a copy of the compiled inspection and allows the instigator to finalise the details of the personnel and documents which are used in the inspection. For each participant and document in the declaration section, the instigator may enter a person or document from ASSIST's databases. New personnel and documents can be added to these databases at any time. For example, consider the Fagan process shown in Figure 4.22. For the document known as Code during this inspection, the real document `series.cc` is associated with it. Similarly, for the participant Author, the real person `fraser` is chosen. When all details have been finalised, the inspection can be started.

At this point, each participant is informed of their participation by email, and when they start ASSIST the inspection name will appear in their list of pending inspections, as shown in the top window in Figure 4.23. Double clicking on the inspection name allows the inspector to participate in that inspection, bringing up an **Execute** window, like the bottom window in Figure 4.23. This window shows the `Preparation` phase of a Fagan inspection from the moderator's perspective. It can be seen that all five inspection participants are required

to be present, shown by the list of names in the top half of the window. This corresponds with the IPDL description given earlier. Each participant has the name of the person performing that role in brackets. Similarly, the IPDL description for this phase required the documents `Code` and `Design` to be present. These are represented by `series.cc` and `Series Specification` respectively, since those documents were chosen when the inspection details were finalised. Finally, the definition of `Moderator` specified that `Defects4` would be used as a defect list for this participant. This document also appears in the document list.

Next to each participant's name is a status indicator, with the letter 'C' used to indicate that the person is currently participating in this inspection, while an 'F' is used to indicate that the participant has finished their work in this phase, controlled by an item in the **Inspection** menu. The **Inspection** menu also allows each participant to leave the inspection. The **Moderator** menu is only available to the moderator and contains controls to advance or abort the inspection. If the current phase has a time limit it is shown in the **Time remaining** box at the top right of the window. Finally, the bottom line of the window shows the current fold (for an N-Fold inspection), the current phase, and any responsibility which this participant may have.

4.2.2 Inspection Facilities

The facilities available to the participants depend on the inspection phase. Figure 4.24 shows a typical view of the Fagan inspection described in Figure 4.22. This view is of the Preparation phase.

Document Handling In contrast with all existing inspection tools, ASSIST has a flexible document type system, allowing new document types and their associated browsers to be added as required. This addresses one of the fundamental weaknesses identified in existing tools. ASSIST has an open architecture with a well-defined interface which browsers must follow. This interface allows the use of standard ASSIST features such as annotation. Several browsers were implemented for the first version.

The **list browser** allows the user to manipulate lists of items, typically document annotations describing defects. Lists can be either read only or read-write. Each item within a list consists of a title, the name of the document which the item refers to, a position and a textual description. Classification of defects was deemed to be an essential feature of an inspection support tool, therefore ASSIST allows items to be classified according to a user-definable classification scheme (specified by IPDL), with up to three levels. The list browser allows items to be added, removed, edited and copied between lists. ASSIST implements a flexible mechanism for describing the positions of annotations, allowing annotation at virtually any

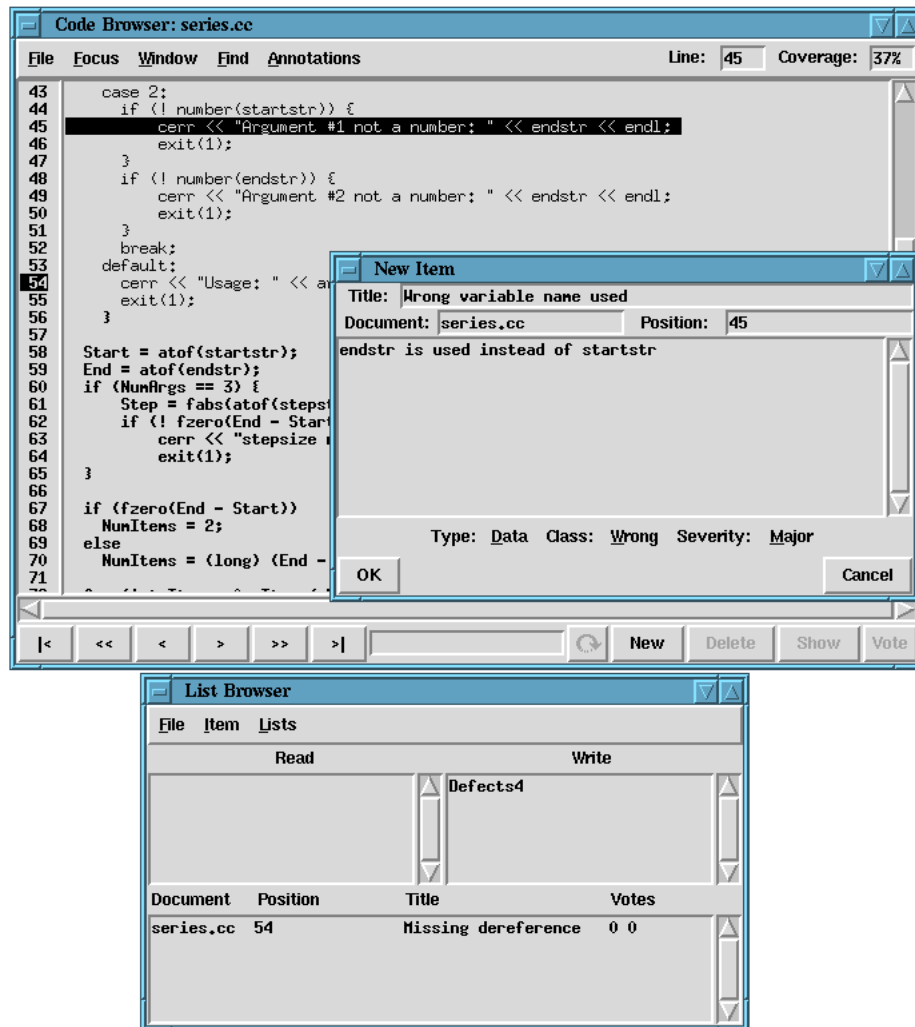


Figure 4.24: Using ASSIST to inspect some C++ code.

scale, from individual letters and words up to paragraphs, sections, and whole documents. The system automatically links annotations and the area of the document to which they refer.

The **code browser** allows documents to be viewed and annotated via the list browser. The browser is based on the concept of a current focus, i.e. a line of code which is currently under scrutiny. The current focus can be annotated, or existing annotations read. The browser indicates the current line number and the percentage of the document inspected. The view of the document can be split horizontally or vertically, allowing two separate areas of the document to be viewed simultaneously. Finally, a find facility is available, allowing the user to search for specific strings in the document. The code browser and the list browser are shown in Figure 4.24.

Individual Preparation This consists of each inspector studying the product and adding defects to their private lists. ASSIST provides a **simple browser**, which is similar to the code browser but without annotation facilities. This browser is used for all supporting documents, such as checklists and specifications, which are themselves not being inspected. All relevant documents are therefore available on-line, satisfying another of the criteria identified in Section 3.3. When preparation has finished, the moderator moves the inspection on to the Inspection stage. The facilities of ASSIST then change to allow a synchronous meeting to be held.

Meeting Support During a group meeting, the focus for the whole group is controlled by the reader: when the reader moves the focus to a new line, the browsers of the other participants are automatically updated. The list browser allows participants to propose items from their personal lists to the whole group, allowing the item to be discussed and voted on. If the item is accepted, it is copied to a master list of defects, representing the output of the group meeting. The scribe may also edit proposed items to reflect refinements suggested at the meeting. ASSIST also makes use of the Mbone video tool `nv`, audio tool `vat` and whiteboard tool `wb`, along with its own textual discussion mechanism `discourse`, to allow distributed meetings. The list browser allows threads of discussion to be created, supporting asynchronous meetings.

Data Collection ASSIST collects the time spent in inspection, the order in which lines of code are inspected, and the time each defect was found.

Apart from the process modelling facilities, this implementation of ASSIST provides the level of functionality required to compare tool-based inspection with paper-based, a level which is similar to that provided by the tools described in Chapter 3. The next chapter describes a controlled experiment to compare paper-based and tool-based inspection. Not only would this give information on the comparative effectiveness, it would provide useful feedback on the usability of ASSIST and help shape the remaining research, in terms of advanced facilities to support defect detection. The next chapter also qualitatively evaluates IPDL.

Chapter 5

Evaluation of Basic Tool Support

This chapter presents an evaluation of IPDL, comparing it with other means of describing inspection processes. It also presents an evaluation of the basic version of ASSIST. This takes the form of a controlled experiment comparing it with paper-based inspection.

5.1 IPDL Evaluation

The evaluation of IPDL takes two forms. To begin with, it must be able to describe all popular inspection processes. Section 4.1.4 showed the implementation of Fagan inspection, while IPDL descriptions of the other seven processes described in Chapter 2 can be found in Appendix C. IPDL must also be compared with other methods of defining inspection processes. There are currently three alternative methods for representing inspection processes: general process modelling languages, the technology underlying the Scrutiny inspection support tool, and the modelling facilities provided by CSRS. General purpose languages have already been described. This section compares IPDL with Scrutiny and CSRS.

5.1.1 Scrutiny

The Scrutiny tool [15], described more fully in Section 3.2.3, is an inspection support tool based on ConversationBuilder [63], which in turn is a generic tool designed to support collaborative work activities. ConversationBuilder (CB) provides a structured conversation model which is suitable for modelling the process of software inspection.

The CB architecture consists of three components. The Message Bus is a multi-cast messaging system which allows communication between all components of CB and its applications. Next is the User Interface Suite which provides each client with the means to manage the user interface, along with other housekeeping tasks such as file manipulation. Finally, there is the Conversation Engine which manages collaboration and the actual application activities. Each application is known as a protocol and is a CLOS (Common Lisp Object System) description of the conversation to be enacted by the conversation engine.

Scrutiny implements a basic four stage inspection process. *Initiation* allows the moderator to organise the inspection by obtaining the product to be inspected and inviting the participants. *Preparation* is an individual activity where each inspector prepares comments and questions about the product. *Resolution* is a synchronous group meeting where the annotations created in preparation are shared and discussed, with the results being noted by the recorder. Finally, *Completion* allows the results of the inspection to be processed and propagated to other stages of the development process.

The fundamental inspection process used by Scrutiny is described with Petri nets, which define the states a role can reach at each stage of the inspection. The role of the user determines the interface presented (and, hence, the facilities available) to that user. CB allows two types of data to be displayed: private data (which can be seen only by its owner) and shared data (which can be accessed by any participant in the review). Shared information can be either synchronous or asynchronous. Synchronous information has any updates propagated to all appropriate users immediately, while asynchronous data is only presented when requested by the user.

To generalise the process, work was carried out to alter the operations available to each role and during each stage. A major change came from splitting Preparation into two sections: private preparation and shared preparation. To model different review methods, Scrutiny allows a number of policies concerning when and how participants move from private preparation to shared preparation. While this change of policy undoubtedly improves the flexibility of Scrutiny, it is still limited to a basic four stage process. There is no ability to move between individual and group stages at will, as is required by some methods. There is also no provision for holding meetings concurrently, or for supporting an N-Fold type inspection. IPDL, on the other hand, was designed with these features as a fundamental part of the language.

The use of CB provides much scope for complete process flexibility, since it is designed as a generic CSCW system. However, this flexibility is balanced by the effort required to implement a full system, which is a non-trivial task. IPDL is designed to obviate such effort. The difficulties in extending and modifying CB-based applications have been noted: ACME [2]

was produced as an attempt to extend Scrutiny with process evolution capabilities, and integrates the Marvel process engine [60] with CB. Marvel itself has already been described in Section 4.1.1.

The extensions planned for Scrutiny consisted of adding a number of schemes to improve the effectiveness of reading annotations and allowing Scrutiny to be customised for specific inspection types. This customisation comes in the form of allowing specific tools to be run for each inspection type, and is, in essence, adding a new step to the inspection process. For example, inspection of C code may require the static analysis tool *lint* to be run on the product. Integration with Marvel was chosen as the implementation vehicle as it allowed other work on Scrutiny to proceed unhindered. This integration consists of the Marvel system monitoring the actions of CB and using this information to form an external view of the Scrutiny process. Marvel's envelope facility is then used, in conjunction with a message sending program, to send messages to Scrutiny.

Arguably the most interesting part of this research concerns the addition of new process steps. It can be seen how this work may be further generalised by adding other steps to the inspection. However, it is not clear how flexible such an approach may be, or how easy it is to implement. Some of the implementation described involves programming at a low level, and it is exactly this type of effort that IPDL is aimed at avoiding.

Finally, an improved version of Scrutiny with much more flexibility was designed, but the project was ended before implementation could take place [42], hence any further evaluation is impossible.

5.1.2 CSRS

CSRS (Collaborative Software Review System) [117], described in Section 3.2.8, is a computer based review system developed to implement a software inspection framework derived by its author. The framework breaks inspection into a series of phases. Each phase can be classified by three characteristics: objective, interaction mode and technique. The objective describes the goal of each phase, and can be one of comprehension, examination or consolidation. A comprehension phase is concerned with becoming familiar with and understanding the document. An examination phase is intended to be used for finding defects. A consolidation phase is where items found by individual reviewers are collated into an agreed form. The interaction mode is concerned with the degree and type of collaboration. A group interaction involves all participants together, while an individual interaction involves each reviewer working alone. A selective subgroup interaction involves only a subset of the inspection team. With a group or subgroup interaction, the type of collaboration can also be defined, either as

synchronous or asynchronous. A synchronous interaction involves all participants performing their task at the same time. With asynchronous interaction the participants can perform their task at any time convenient to them. The final characteristic is the technique used by participants to achieve the stated objective, of which there are several types. Inspection techniques vary from straightforward free review to checklist assisted inspection. Consolidation techniques include discussion and voting. Techniques may also involve the use of tools. Each phase may also have a set of entry and exit criteria associated with it. The criteria usually define properties that can be expected of both review artifacts and participants before and after each phase. Using this framework, each major inspection method is modelled as a sequence of phases, along with the appropriate characteristics.

This framework was used to develop the process modelling facilities in CSRS. These facilities are based around a set of languages, the most important of which are the data and process modelling languages. The data modelling language is used to describe the documents manipulated by the inspection, including their type and relationships with other documents. Each document is represented by one or more nodes, which in turn contain fields of various types. Nodes may also have attributes and status associated with them. Relationships between nodes are represented by typed links, implementing a hypertext network. Four base nodes types are defined. Source nodes contain documents to be reviewed. Commentary nodes are used to contain items generated during the inspection, such as defects. Checklist nodes contain verification aids such as checklists. Administrative nodes contain information about the review process itself, including details of participants. These node types can be inherited and specialised.

The process modelling language defines the phases that compose each inspection. It is based around a single multi-purpose phase which can be given the attributes, such as objective and interaction mode, which have been defined within the framework. A process description consists of a set of ordered phase definitions. CSRS allows these phases to be invoked either manually or automatically. Manual invocation involves the administrator periodically checking the status of a phase and activating the next phase as required. Automatic invocation is supported by the use of exit and entry criteria.

In comparison with IPDL, the facilities of CSRS are far more generic, providing a less rigid underlying process model, consisting of a sequence of generic phases. On the other hand, CSRS cannot be used to model concurrent process steps. For example, in N-Fold inspection a potentially important document is inspected multiple times using one or more inspection methods. While this may be modelled serially, it is important that the inspections can occur concurrently, otherwise the expense of the inspection will exceed the benefits. Also, the

generic phase cannot be used to model tasks such as planning and rework. These tasks are an integral part of the inspection process and must be modelled in some way, especially if it is intended to make use of computer assistance. Describing the inspection as a series of generic phases may restrict the support that can be provided for each phase. Since, as has been shown, inspections consist of a limited number of well-defined phases, it is preferable to explicitly model as many of these phases as possible, while a similar approach of describing such characteristics as objective can then be used for the more variable phases of defect detection and collection. This is the approach used in IPDL.

Other questions arise with respect to computer assistance. For example, the framework introduces the idea of roles, but it is not clear how responsibilities are associated with these roles. There appears to be no methods for providing each reviewer with specific checklists for their responsibilities, such as is required for Active Design Reviews [95]. This is a feature of IPDL.

A major difference between CSRS and IPDL is the size of process definitions. The CSRS definition of FTArm is around 2700 lines of code. The IPDL version, presented in Appendix C.5, is less than 200 lines. The difference in effort required to produce these is obvious. This difficulty is perhaps illustrated by the fact that no attempt has been made to write descriptions of existing processes. In contrast, IPDL descriptions of the eight processes used in its derivation have already been shown. The CSRS languages are also tightly coupled to the underlying system, consisting of a set of Emacs-Lisp functions and macros. This makes it difficult to use in another inspection tool, especially one which is not Lisp-based. IPDL is language and environment independent, and its syntax is such that it can also be used in a non-computer supported environment as a means of precisely communicating inspection processes.

5.1.3 Conclusions

IPDL has successfully been used to implement the eight inspection processes from which it is derived. The diversity of these processes helps ensure that IPDL is capable of describing any existing inspection process. It should also be capable of describing many future processes, provided they conform to the organisation-detection-completion model which underlies IPDL. Although Scrutiny is based on a generic CSCW system which would allow support of multiple inspection processes, tailoring such a system is non-trivial, especially compared to the ease with which processes can be defined in IPDL. The process modelling facilities of CSRS provide much flexibility but once again this flexibility comes with the price of increased complexity. IPDL avoids this complexity by providing high-level, application specific constructs,

resulting in concise, readable process descriptions which can be used both as input to an inspection support tool and to document a paper-based process.

5.2 Comparing Basic Tool-based and Paper-based Software Inspection

Tool-supported inspection will only become an accepted practice if it can be demonstrated that it does not detract from the main goal of software inspections: finding defects. Furthermore, if it can be shown that the simplest level of support does not alter the efficiency of inspection, other, more advanced, support can be explored secure in the knowledge that the overall concept is not fundamentally flawed.

While a number of advantages of tool-based inspection have already been identified, there are a number of possible disadvantages. The time required to train an inspector in the use of the tool may become an issue. If the tool is too complex, it may detract from the effectiveness of the inspection, even in experienced inspectors. Also, many people are far slower at typing than handwriting, slowing down the process of noting down defects, especially if the tool is heavily mouse-based.

There may also be two problems concerning the move from paper to screen. One concern is the limited amount of screen space. Examination of the product, a source document and a checklist requires three windows to be on-screen simultaneously. However, most common displays are not capable of simultaneously showing three such windows of sufficient size to be useful. The screen may also be cluttered with other windows necessary for operation of the tool, such as has been described in *Scrutiny* [44]. Contrast this with paper-based inspection, where inspectors are free to find as large a workspace as required and to spread all the documents around in a manner comfortable to their working method. The second problem concerns reading text from a screen. There have been a number of studies comparing reading from screen versus reading from paper, and Dillon [29] provides a good review of these. Evidence points to a 20–30% reduction in speed when reading from screen compared to reading from paper, while reading accuracy may suffer for visually- or cognitively-demanding tasks. On the other hand, comprehension appears not to be affected, and may even be improved. Ergonomic issues may also have a part to play in reading text from a screen. These include the fixed orientation of the screen, differing width to height ratios, refresh rates, image polarity, display quality and so on. Reading from screen has also traditionally been thought to be more tiring than reading from paper, while there appears to be a natural tendency for users to prefer paper. Several caveats apply when considering how these types of study apply to tool-based

inspection, including their length, the type of text used, and the task being performed. Lastly, it should be noted that the quality of an individual display will affect the usability of such a tool, and that the ideal presentation will vary depending on the individual user, implying that the ability to customise the tool is an advantage.

This section describes a controlled subject-based experiment designed to investigate any possible reduction in inspection efficiency when moving to a tool-based inspection process. An appropriate measure of inspection efficiency is the number of defects detected in a constant period of time. When comparing tool-based and paper-based inspection the following null hypothesis, H_0 , can be formed:

There is no significant difference in performance between individuals performing tool-based inspection and those performing paper-based inspection, measured by the total number of defects found during a given time period.

The alternative hypothesis, H_1 is simply:

There is a significant difference in performance between individuals performing tool-based inspection and those performing paper-based inspection, measured by the total number of defects found during a given time period.

Similar hypotheses can also be formed when considering the performance of inspection teams as a whole.

5.2.1 Evaluations of Existing Inspection Support Tools

While there have been a number of attempts at implementing tool support for software inspection, the quality of evaluation of each tool varies enormously. In the case of ICICLE [12], the only published evaluation comes in the form of lessons learned. In the case of Scrutiny, in addition to lessons learned [44], the authors also claim that tool-based inspection is as effective as paper-based, but there is no quantifiable evidence to support this claim [43].

Knight and Myers [66] describe two experiments involving their InspeQ inspection tool, designed to support their phased inspection method. The first simply provided information on the feasibility of their method, and on the usability of the associated toolset. The second experiment involved inspection of C code, but provided no comparison with paper-based inspection. Mashayekhi [83] reports on the implementation of three prototype tools, with the aim of investigating distribution and asynchrony in software engineering. Again, no comparisons with paper-based inspection are made, except in the case of group meetings, where comparable meeting losses are found using both the tool and paper-based methods.

Finally, CSRS (Collaborative Software Review System) has been used to compare the cost effectiveness of a group-based review method with that of an individual-based review method [116]. Again, since both methods are tool-based, there is no indication of the relative merits of tool-based and paper-based inspection.

Although the evaluations described above attempt to measure, in various ways, the effectiveness of tool support, the fundamental question “Is tool-based software inspection as effective as paper-based inspection?” remains unanswered. Practically all existing support tools provide the level of support required to test the hypotheses stated above. Given that favourable comparison with paper-based inspection is the means by which tool-supported inspection will become acceptable, it is surprising that there is little investigation of this nature.

5.2.2 Experiment Design

The testing of the hypotheses required two groups of subjects to inspect a single document, one using a tool-based approach and the other using a paper-based approach. To ensure that any effect was not simply due to one group of subjects being of higher ability, the subjects must also inspect a second document, this time using the alternative approach. The inspection process used was an abbreviated Gilb and Graham type. It consisted of two phases: an individual detection phase, where each subject inspected the document for defects, and a group collection meeting, where individual lists were consolidated into a single master list for the group. The group phase also provided an additional defect detection opportunity. An IPDL process implementing this inspection was written and used as input to ASSIST for the tool-based parts of the experiment. The appropriate inspection involving the correct participants and documents was initiated for each group as required.

Subjects

The experiment was carried out during late 1996 as part of a team-based Software Engineering course run by Strathclyde University's Department of Computer Science for third year undergraduates. The students already had a firm grounding in many aspects of Computer Science. In particular, they had been taught programming in Scheme, C++ and Eiffel, and had also completed a course in the fundamentals of Software Engineering. Subject motivation was high, since the practical aspects of the experiment formed part of their continual assessment for this course, the final mark of which contributes to their overall degree class.

A total of 43 subjects participated in the class, split into two approximately equal sections. Section 1 had 22 subjects and Section 2 had 21 subjects. The split was achieved by ordering

subjects according to their mark in a C++ programming class (C++ was chosen as the type of code to be used in the experiment). Adjacent subjects were then blocked into sets of four, with two randomly chosen subjects assigned to one section, with the remaining two subjects assigned to the other. Within the two sections the subjects were organised into groups of three (and a single group of four). This was done in such a way as to create equal ability groups, based on their C++ programming marks. Section 1 therefore consisted of six groups of three subjects and one group of four subjects, while Section 2 contained seven groups of three subjects.

Materials

Having previously assisted with the running of several defect detection experiments [90, 122], it was decided that the most appropriate material to inspect would be C++ code. A number of factors influenced this decision. Initially, source code was chosen as the appropriate material due to ease with which defects in code can be defined. This is in contrast with, say, English language specifications, which provide many problems of ambiguity. It is also easy for inspection of such material to degenerate into arguments over English style and usage. Intelligent seeding of defects in code avoids these problems and provides a well-defined target against which performance can be judged. It is also easier for the experimenter to classify defects and hence subject performance is more accurately measured.

Subject experience was also taken into account. Inspection must be performed by personnel with experience in the type of document being inspected. It was therefore important to choose material in a form which the subjects had experience in. This also avoids teaching a new notation or language which subjects may spend much of their time trying to understand and become familiar with, instead of finding defects. Since the subjects were competent in C++, material in that language was chosen for the experiment. The decision was further ratified by the availability of high quality material from a local replication of Kamsties and Lott's defect detection experiment [62]. Of course, the act of choosing a single material type affects the extent to which the results can be generalised. It is felt, however, that the results are applicable to inspection of source code in general.

For the training materials, a selection of programs originally used in Kamsties and Lott's experiment were used, since each program had an appropriate specification, a list of library functions used by the program and a comprehensive defect list. These programs were originally written in non-ANSI C and were translated into standard C++ for the experiment. The programs used were: `count.cc` (58 lines, 8 defects), `tokens.cc` (128 lines, 11 defects) and `series.cc` (145 lines, 12 defects). A further example, `simple_sort.cc` (43 lines,

4 defects) was created for use in the tool tutorial.

Since the Kamsties and Lott material had already been used in the same class the previous year, the two programs to be used for the experiment (and, hence, the assessment), were specifically written afresh. One program (`analyse.cc`, 147 lines, 12 defects) was based on the idea of a simple statistical analysis program given in [28]. The second program (`graph.cc`, 141 lines, 12 defects) was written from a specification for a Fortran graph plotting program, originally found in [8]. For each program, a specification written in a similar style to that of the Kamsties and Lott material was also prepared, along with appropriate lists of library functions.

There is no clear consensus on the optimal inspection rate. For example, both Barnard and Price [7] and Russell [103] quote a recommendation of 150 lines of code per hour. On the other hand, Gilb and Graham [41], recommend inspecting between 0.5 and 1.5 pages per hour, translating to between 30 and 90 lines of code. All conclude that lower rates improve defect detection. Each practical session lasted two hours, giving an inspection rate of around 70 lines per hour. This figure represents a compromise since subjects were not professional inspectors and could not be expected to perform at the highest recommended rates. At the same time, there was enough time pressure to make the task realistic. Two hours is also a standard inspection meeting length.

The actual inspection task was to use the program specification and list of library functions to inspect the source code for functionality defects, making use of a checklist. Use of a checklist is standard inspection procedure, and subjects were supplied with the checklist described below. Subjects were specifically discouraged from finding defects relating to other qualities, such as efficiency. Each program was seeded with functionality defects and with checklist violations. These were based on those naturally occurring when the programs were written, and those found in the Kamsties and Lott material. For the two experimental programs, defects in one program were matched, in terms of type and perceived difficulty, with defects in the other program, in an effort to match the overall difficulty of the programs. All programs used compiled with no errors or warnings using `CC` under SunOS 4.1.3.

The checklist used was derived from a C code checklist by Marick [81], a C++ checklist by Baldwin [3] (derived from the aforementioned C checklist), the C++ checklist from [51] and a generic code checklist from [33]. Items which were considered to be irrelevant were removed from the C and C++ checklists (for example, no programs made extensive use of macros), along with esoteric items, such as those dealing with threaded programming and signals. From the generic checklist, items not relevant to C++ were removed. Duplicates were then removed and the remaining items grouped into a number of categories. An additional category was

added concerning differences between the specification and behaviour of the program. Finally, another edit was performed on the checklist to reduce the number of categories, allowing the checklist to fit on two sides of paper. It was felt that a short checklist covering the major points to consider would be more effective than a much longer, more detailed checklist which the subjects would struggle to cover in the time allowed. This follows practice recommended by Gilb and Graham [41]. The checklist and all other materials used in the experiment are presented in Appendix D.

Instrumentation

For paper-based inspection, each subject was given an individual defect report form, like that in Appendix D.3, containing blank entries to be filled with each defect found. For group meetings, the scribe was given a similar form to prepare the master list, presented in Appendix D.4. During tool-based inspection, ASSIST was used to keep both individual lists and the master list. Each practical session was limited to a maximum of 2 hours. Almost all participants made use of the full two hours.

For each subject, data collected were the total number of correct defects found, along with the number of false positives submitted (i.e. defects which subjects incorrectly identify), and similarly for each group. Also calculated were meeting loss (number of defects found by at least one individual in the group, but not reported by the group as a whole), and meeting gain (number of defects reported by the group, but not reported by any individual) for each group. Finally, for each defect in each program, the frequency of occurrence was obtained, both in tool-based and paper-based inspection.

Experiment execution

The practical element of the course ran over a period of ten weeks. The first six weeks were devoted to providing the subjects with training in software inspection and using ASSIST, as well as refreshing their C++ knowledge. These practical sessions were interspersed with lectures introducing each new topic where appropriate. After inspection of each program was complete, the subjects were presented with a list of defects in that program. The remaining four weeks were used to run the actual experiment. Appendix D.1 details the exact timetable used. Each practical session was run twice, once for each section of the class, thus ensuring their separation when using different methods on the same program. Both practicals occurred consecutively on the same afternoon of each week.

Threats to Validity

Any empirical study may be distorted by influences which affect dependent variables without the researcher's knowledge. These are threats to internal validity, and the possibility should be minimised. The following such threats were considered:

- Selection effects may occur due to variations in the natural performance of individual subjects. This threat was minimised by creating equal ability groups.
- Maturation (learning) effects concern improvement in the performance of subjects during the experiment. The data was analysed for this and no effect was found. Section 5.2.3 describes this analysis in more detail.
- Instrumentation effects may occur due to differences in the experimental materials used. To help counteract the main source of this effect in the experiment, both groups of subjects inspect both programs.
- Presentation effects may occur since both sets of subjects inspect the programs in the same order. It is believed that if such an effect exists, it is symmetric between both sets of subjects, and that the effect presents less risk than the plagiarism effect possible when the order of presentation is reversed for one set of subjects.
- Plagiarism was a concern in our experiment since the group phase of each experimental run took place one week after the individual session, hence providing an opportunity for undesired collaboration among subjects. This was mitigated by retaining all paper materials between phases. With the same purpose in mind, data from the tool regarding the individual phases was extracted immediately after each session. Furthermore, access to the tool and any on-line material was also denied. Any plagiarism effect between groups would be noticeable by any group presenting an above average meeting gain. No such groups were detected.
- Boredom may have affected the results. Subjects were asked to perform a total of six inspections. Although these were spread over ten weeks, it is likely that subject enthusiasm was waning towards the end of the experiment. On the other hand, the use of the tool may have provided some novelty to mediate this factor.

Threats to external validity can limit the ability to generalise the results of the experiment to a wider population, in this case actual software engineering practice. The following were considered:

Program	analyse.cc		graph.cc	
Section	1	2	1	2
Method	Tool	Paper	Paper	Tool
Subjects	22	21	22	21
Mean	7.68	7.76	6.64	6.00
St. Dev.	1.55	1.92	1.43	2.05
St. Error	0.33	0.42	0.30	0.45
F Ratio	0.02		1.40	
F Prob.	0.88		0.24	

Table 5.1: Analysis of variance of individual defect scores.

- The subjects involved in the experiment may not be representative of software engineering professionals. This was unavoidable since the choice of subjects was limited by available resources.
- The programs and defects used may not be representative of the length and complexity of those found in an industrial setting. The programs used were chosen for their length, allowing them to be inspected within the time available. However, the amount of time given to inspect each program was representative of industrial practice quoted in popular inspection literature.
- The inspection process used may not correspond to that used in industry, in terms of process steps and number of participants. For example, the process used did not involve the author presenting an overview of the product, giving no context for the inspection. A rework phase was also not used. However, the detection/collection approach used in the experiment is a standard process [41].

These threats are typical of many empirical studies, e.g. [62, 99]. They can be reduced by internal and external replication of the experiment, with other subjects, programs and processes.

5.2.3 Results and Analysis

Defect Detection

The raw data from the experiment can be found in Appendix E.1.1. Table 5.1 presents a summary of the data and the analysis of variance of the individual phases of both inspections. For each program, the method used by each section of subjects is shown, along with the number of subjects in that section and the mean number of defects found in the program.

Program	analyse.cc		graph.cc	
Section	1	2	1	2
Method	Tool	Paper	Paper	Tool
Groups	7	7	7	7
Mean	10.86	10.71	9.57	8.86
St. Dev.	0.69	0.95	1.27	1.07
St. Error	0.26	0.36	0.48	0.40
F Ratio	0.10		1.29	
F Prob.	0.75		0.28	

Table 5.2: Analysis of variance of group defect scores.

The standard deviations, standard errors and F ratios and probabilities are also shown. For `analyse.cc`, it is obvious that there is very little difference in performance, and this is confirmed by the analysis of variance. For `graph.cc`, the section using paper-based inspection appear to outperform that using the tool, although this difference is not significant. In both cases the null hypothesis concerning individuals cannot be rejected.

Table 5.2 presents a summary of the data and the analysis of variance of the group phases of both inspections. These results follow the same pattern as for individual: `analyse.cc` provides very similar results between methods, while `graph.cc` provides a larger difference, but which is not statistically significant. Again, the null hypothesis as applied to groups cannot be rejected.

Under further investigation, the data from the individual phase of the `graph.cc` inspection failed the Levene test for homogeneity of variances. This is probably due to the increased difficulty of `graph.cc` compared with `analyse.cc`, discussed next. However, the robustness of the F test is well documented. For example, Boneau [10] has studied the effects of violating the assumptions which underlie the t test, and generalised these results to the F test. Provided samples sizes are sufficient (around 15) and virtually equal, and a two-tailed test is used, non-homogeneity of variances should not cause difficulties. A similar conclusion is presented by Edwards [34]. As a safeguard, the Kruskal-Wallis non-parametric test was applied to all four sets of data, and gave results similar to those for the parametric tests, with no significance.

The data was analysed for any effect stemming from the order in which the methods were used and for any difference caused by the two programs. The results are shown in Table 5.3. There proved to be no significant difference between subjects who used the tool first and those who used paper first. However, the results indicated a significant difference in

Effect	F Ratio	F Prob.
Order	0.34	0.56
Program	33.78	$\ll 0.01$
Order \times Program	2.20	0.15

Table 5.3: Analysis of variance of method order and program.

the difficulty of `analyse.cc` compared with `graph.cc`. This was also supported by one of the post-experiment questionnaires (described in more detail later in this section). The greater difference in performance with `graph.cc` may imply that tool-based inspection becomes less efficient as the material under inspection becomes more complex. This could be due to the use of the tool interfering with the cognitive task of defect detection. Finally, the data was analysed for any effect from the order in which the methods were used combined with the two different programs. No significant result was found.

The next set of data to be analysed concerned the detection frequencies of individual defects. Comparing the frequencies achieved by each method would allow the discovery of defects which tool users found particularly easy or difficult in comparison with paper-based inspection. This might then indicate benefits of the tool, or ways in which the tool could be improved. It should be noted, however, that due to the nature of the data in some cases these differences will be due to natural variation, and hence do not correspond to any underlying effect. Therefore, although examining the data at this level is useful, caution must be exercised to avoid hypothesising about effects which do not exist.

Figure 5.1 summarises the frequency with which each defect was found in `analyse.cc` during the individual phase. In most cases there is no great difference between the scores achieved with the tool compared to those using paper. Considering the four defects with the largest differences (1, 4, 7, and 8), there is no clear indication why such differences exist. Defect 1 is an array indexing error, defect 4 concerns output appearing at the wrong point in the program, defect 7 concerns missing functionality and defect 8 is a failure to initialise an array. In theory, defect 8 should be found by all subjects, since it is explicitly covered by a checklist item. The difference may indicate that it is more difficult to use the on-line checklist (lack of screen space). On the other hand the low scores point to an overall lack of checklist use. Hence, the promotion of checklist usage may be a means through which tool support can increase inspection efficiency.

Figure 5.2 shows the frequency of detection for the group phase of the `analyse.cc` inspection. The largest difference is for defect 3 with only one tool-based group finding it,

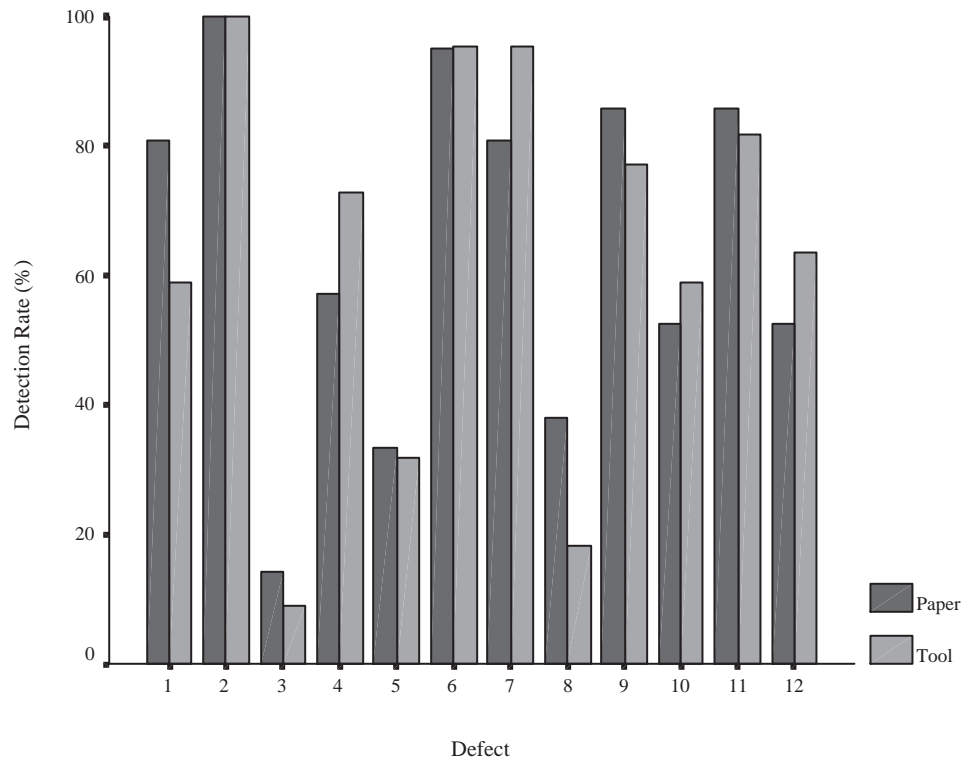


Figure 5.1: Summary defect detection data for `analyse.cc` (individuals).

compared to three paper-based groups. The reason for this becomes apparent by considering the individual results. Only two tool users found this defect, each belonging to distinct groups. One group reported the defect, while the other did not. In contrast, three paper-based subjects found the defect, each of whom belonged to a different group. All three groups managed to report this defect. Given a different makeup of groups, this difference could have been reduced to zero.

Figure 5.3 summarises the defect detection frequency for the individual phase of the inspection of `graph.cc`. The largest difference appears for the third defect, which was found by 90.9% of the paper-based inspectors, yet only 61.9% of the tool users. This alone represents 35% of the overall difference between tool and paper. This defect concerns a missing function call, which means the program does not print some output specified by the specification. In fact, this is a very easy defect to detect using the search facility of the tool. By entering the function name as the target of the search, the inspector can find calls to that function, almost guaranteeing that the defect will be found. However, although the mechanics of the find facility were explained, the use of the tool to detect such defects was not explicitly taught to the

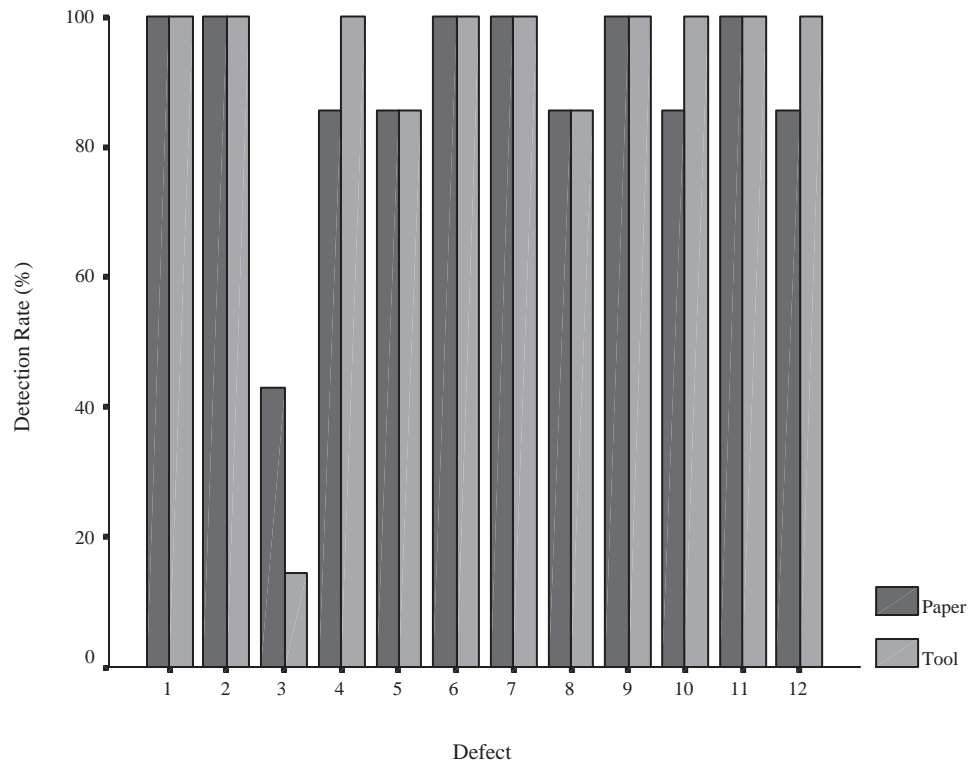


Figure 5.2: Summary defect detection data for `analyse.cc` (groups).

subjects. The reason for subjects performing paper-based inspection finding this defect more easily is not clear. A difference of just over 20% in favour of paper-based inspection occurs for defect 5 (variable names X and Y are transposed), but there is no obvious reason for this difference. The same applies to defect 11 (an incorrect calculation), which has a difference of just over 20% in favour of the tool.

The defect detection data for the group phase of the inspection of `graph.cc` is shown in Figure 5.4. This time defect number 8 has the largest difference between methods. The data for the individual phase already shows that this is the most difficult defect to find. The relatively poor performance of the tool can be explained by the fact that every individual who found the defect belonged to a separate group, giving the paper-based groups a 3 to 1 advantage. Another paper-based group also managed to find the defect at the meeting, giving 4 to 1. Finally, the single tool user who actually found the defect either failed to mention it at the group meeting or was talked out of it, since that group did not report it. A similar trend is apparent with defect 9, with 4 individual tool users mapping into 4 groups, and 7 paper users mapping into 6 groups. Again, these results show group makeup has an effect on the group

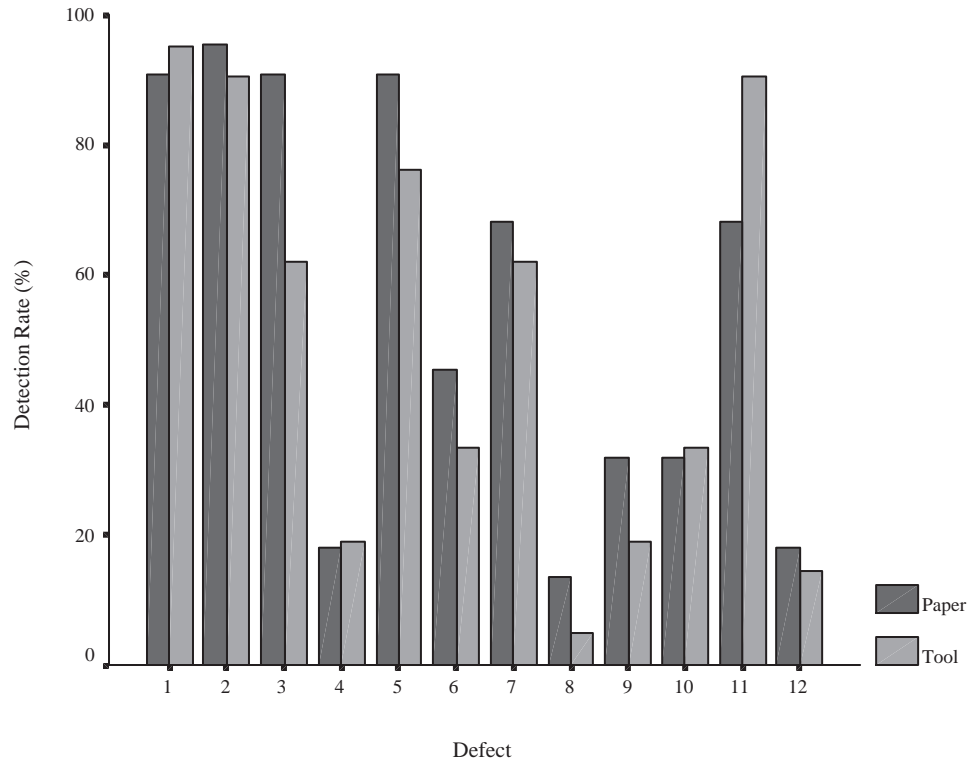


Figure 5.3: Summary defect detection data for `graph.cc` (individuals).

result. Different group makeups could have reduced both these differences to 1, or it may have increased the difference.

The overlap between group members' defect lists can have a fairly significant effect on the group score. If there is a large overlap (i.e. individuals have many defects in common) the group score is likely to be lower than that of a group with a small overlap (individuals have few defects in common, hence the total score is higher). Groups with individuals whose defect lists have a greater overlap are disadvantaged, even though individual scores may be very respectable. On the other hand, if more than one participant reports an individual defect, the risk of that defect becoming a meeting loss may be reduced.

False Positives

In addition to the number of defects found by each subject, the number of false positives reported was also measured. False positives are items reported by subjects as defects, when in fact no defect exists. It is desirable to investigate whether tool-supported inspection alters the number of false positives reported, since an increase would reduce the effectiveness of

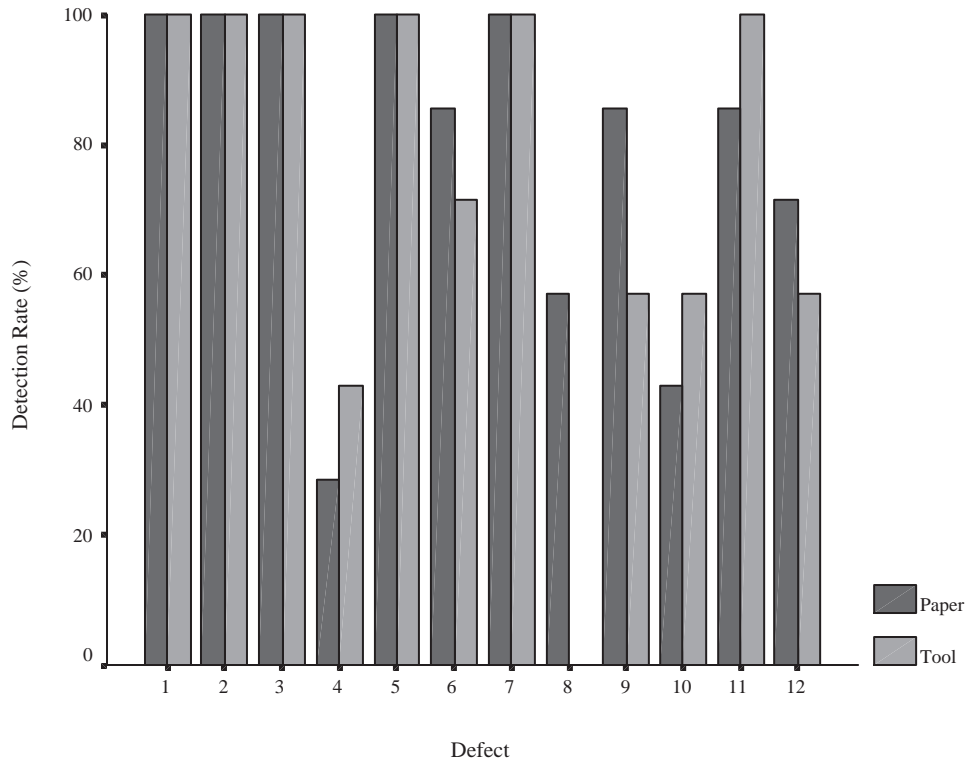


Figure 5.4: Summary defect detection data for `graph.cc` (groups).

the inspection. On the other hand, if use of the tool in some way suppressed false positives, the efficiency of the inspection would be increased, with less time wasted on discussing these issues.

Table 5.4 presents the analysis of variance for the false positive data from the individual phases of each inspection. While the tool appears to provide an improvement over paper for `analyse.cc`, this difference is not significant. On the other hand, the means for `graph.cc` are almost identical, and this is confirmed by the ANOVA test. The analysis of variance for the false positive data of the group phases is shown in Table 5.5. For both programs, there is no difference between the tool-based and paper-based approaches.

Examination of the false positives revealed no discernible difference in those produced by tool-based inspection as compared with paper-based. The only point of interest was that some of the false positives which occurred were defects which had occurred in training material. Presumably the subjects had memorised these as defect types and submitted them as defects without checking if they actually occurred, in the hope of increasing their score.

Program	analyse.cc		graph.cc	
Section	1	2	1	2
Method	Tool	Paper	Paper	Tool
Subjects	22	21	22	21
Mean	3.59	4.19	3.23	3.24
St. Dev.	1.943	1.94	1.9	2.14
St. Error	0.41	0.42	0.4	0.47
F Ratio	1.02		≈0.00	
F Prob.	0.32		0.99	

Table 5.4: Analysis of variance of individual false positives.

Program	analyse.cc		graph.cc	
Section	1	2	1	2
Method	Tool	Paper	Paper	Tool
Groups	7	7	7	7
Mean	3.57	3.14	2.14	2.43
St. Dev.	1.72	1.46	1.34	2.22
St. Error	0.65	0.55	0.51	0.84
F Ratio	0.25		0.08	
F Prob.	0.62		0.78	

Table 5.5: Analysis of variance of group false positives.

Program	analyse.cc		graph.cc	
Section	1	2	1	2
Method	Tool	Paper	Paper	Tool
Groups	7	7	7	7
Mean	0.29	0.14	0.43	0.57
St. Dev.	0.49	0.38	0.79	0.53
St. Error	0.18	0.14	0.30	0.20
F Ratio	0.38		0.158	
F Prob.	0.55		0.698	

Table 5.6: Analysis of variance of meeting gains.

Program	analyse.cc		graph.cc	
Section	1	2	1	2
Method	Tool	Paper	Paper	Tool
Groups	7	7	7	7
Mean	0.14	0.14	0.71	0.71
St. Dev.	0.38	0.38	0.49	0.95
St. Error	0.14	0.14	0.18	0.36
F Ratio	0		0	
F Prob.	1.00		1.00	

Table 5.7: Analysis of variance of meeting losses.

Meeting Gains and Losses

The final set of data to be analysed concerns meeting gains and losses. Synchronous meetings are frequently cited as necessary because it is believed that factors such as group synergy can improve the output of the meeting, manifested in defects being found at the meeting which have not been found during the individual phase (providing defect detection, and not just defect logging, is an objective of the meeting). On the other hand, defects may be lost when a participant fails to raise a defect found during the individual phase. It was hypothesised there would be no difference between tool and paper-based methods for both gains and losses. Table 5.6 shows the analysis of variance of meeting gains, while Table 5.7 shows the analysis of variance of meeting losses for group meetings in both inspections. It is clear there is no significant difference between methods.

Overall, gains and losses are insignificant. The small meeting gain indicates that subjects were not performing extra defect detection at the group meeting, despite being told that this was an opportunity which could be exploited. The small meeting loss perhaps indicates a lack

of discussion of each defect, or (more likely) subjects were “playing safe” by putting every reasonable defect into the group list. On a per-defect basis, there are two possibly significant occurrences. In `graph.cc` defect 4 appears as a meeting loss for three groups and defect 10 appears as a meeting loss for four groups. Both are split evenly between methods. Why these defects have more significant meeting losses is not clear, other than they appear to be two of the hardest defects to find. Hence, subjects may be easily persuaded they are not correct defects.

Debriefing Questionnaires

During the course, subjects were asked to complete four questionnaires. The first two were given after the full practice using ASSIST, one after the individual inspection (Questionnaire 1, 67.4% response rate) and one after the group meeting (Questionnaire 2, 48.8% response rate). These questionnaires focussed on eliciting qualitative feedback on ASSIST. Two further questionnaires were then presented, one after the first phase of the experiment was complete (Questionnaire 3, 93% response rate), the other after the second phase of the experiment was complete (Questionnaire 4, 93% response rate). These questionnaires concentrated on such topics as the overall difficulty of the task and the relative merits of paper-based and tool-based inspection. The full text of the questionnaires can be found in Appendix D.11. This section presents some results from these questionnaires.

Questionnaire 1 One concern when moving to tool-based inspection is the ease with which the user can navigate around the document, since doing so on paper is very natural. When asked to rate the difficulty of moving around the document 3.4% of the subjects found it very difficult, 20.7% found it difficult, 24.1% found it average, 37.9% found it easy and 13.8% found it very easy. Overall, most people seemed to have no difficulty, but around 25% of subjects finding some difficulty is concerning.

A second factor, discussed previously, is the number of windows which each user has to have open at one time. When asked how the number of open windows affected their inspection efficiency, 13.8% of subjects said it improved, 31% said it had no effect and 55.2% thought it reduced their efficiency. The conclusion here is obvious: the number of open windows is definitely a problem. One possible solution is the use of a “virtual window manager”, which gives users a number of workspaces which they can move between with single keystrokes. This allows all windows to be open full-sized without overlaying each other. The use of a virtual window manager was not common among the subjects.

Finally, the question “Compared with manual (paper-based) inspection, do you feel that

computer-based inspection is more efficient, less efficient, or about equally efficient?” was asked, and subjects were also asked to give reasons. 55.2% of respondents deemed it less efficient, 26.6% thought it equally efficient, and 17.2% found it more efficient. The majority of negative responses here concern the number of windows used and, to a lesser extent, the speed of the tool. Positive responses were generally given by those who found it easier to manipulate their defect lists.

Questionnaire 2 Subjects were asked to rate the ease-of-use of the defect proposal system. 14.3% of subjects found it difficult to use, 38.1% found it average, 28.6% found it easy to use, while 19% found it very easy to use. The voting mechanism provided by ASSIST was deemed to be useful. 52.4% of subjects found it helped resolve issues, 33.3% said it had no effect, and only 14.3% thought it hindered issue resolution.

Subjects were asked to rate the effect ASSIST had on their group meeting. 4.8% said it had a large negative effect, 19% said it had a negative effect, 38.1% said it had no effect, and 38.1% said it had a positive effect (no subjects said it had a large positive effect). Overall, the use of a tool would seem to have enhanced the meeting. When asked whether a tool-based meeting is more efficient than a paper-based meeting, 38.1% of subjects said it was less efficient, 47.6% said it was equally efficient, while only 14.3% said it was more efficient.

Questionnaire 3 Subjects were first questioned about their understanding of the code. 5% of subjects understood 41-60% of the program, 25% understood 61-80% of the program, with the remaining 70% claiming to understand 81-100% of the program. Only 5% of subjects thought there was not enough time to inspect the code, 87.5% thought they were given the correct amount of time, and 7.5% thought there was too much time.

The group meeting was found to be a useful part of the inspection for most subjects. When asked if their understanding of the program was changed at the meeting, 2.5% said their understanding was confounded, 35% said their understanding was unchanged, but 62.5% said their understanding was increased. This finding supports the common view that meetings are a useful education mechanism. When asked to guess the meeting gain for their inspection, 7.5% said that no gains had been made, 40% said that one or two extra defects had been found, 47.5% said that 3-5 extra defects had been discovered, and 5% estimated that more than five had been found. Subjects obviously believed that group discussion of the program helped find more errors, although these estimates seem very optimistic. Subjects perhaps misunderstood the phrase “meeting gains”, thinking it included defects found by other inspectors which they themselves had not found. Subjects were also asked to estimate their meeting loss. 27.5%

said that no losses had occurred, 35% believed that one or two defects had been lost, 27.5% estimated that 3-5 defects had been lost and 10% believed that more than five defects had gone astray. In general, these estimates seem high (especially those who said more than five). Again, there may be misunderstanding of terminology – some subjects may have assumed “meeting loss” included defects discovered during individual preparation but discarded at the meeting because they were genuinely incorrect.

Subjects were asked to rate their overall group performance in terms of the percentage of defects which they believed their group had found. 2.5% believed they had only found 21-40% of the defects, 30% thought they had found 61-80%, and 67.5% believed they had found 81-100% of the total defects. In general, subjects seemed to slightly underestimate their group performance, but not by much.

Finally, subjects who used ASSIST for this phase of the experiment were asked to rate the usability of ASSIST, with 90.9% of the tool users responding. 10% found it extremely usable, 50% found it fairly usable, 35% found it average, and 5% thought it fairly unusable (no subjects found it totally unusable).

Questionnaire 4 As with Questionnaire 3, subjects were first asked about their understanding of the program. 2.5% understood only 21-40% of `graph.cc`, 20% understood 41-60% of the program, 37.5% understood 61-80% of the program, and only 40% understood 81-100% of the code. Comparing this result with that from the `analyse.cc` inspection there is an overall reduction in understanding of the program by the subjects. When asked if sufficient time was given to inspect the program, 35% of subjects thought that insufficient time was given, 62.5% thought that the time given was about right, and 2.5% thought there was too much time. Comparing this with the result from the `analyse.cc` inspection, it is obvious that more people found it difficult to inspect the program in the time given. Finally, the subjects were asked to rate the complexity of `graph.cc` compared to `analyse.cc`. 49% of subjects believed `graph.cc` to be much more complex, while 44% believed it to be slightly more complex, and only 7% considered it to be of similar complexity to `analyse.cc`. These results all support the earlier statistical analysis concerning the difference between programs.

Once again, the group meeting was found to be useful. 25% of subjects reported no change in their understanding of the program, while 75% said their understanding was increased. In terms of meeting gains, 5% of subjects thought no gains had been made, 60% thought one or two extra defects had been found, 22.5% reported a gain of 3-5 defects, and 12.5% thought more than five extra defects had been found at the meeting. In terms of meeting loss, 72.5% of subjects believed no losses had occurred, 17.5% believed one or two losses had occurred,

10% reported losses of 3-5 defects, with no-one reporting more than five losses. These figures are more realistic than their counterparts from Questionnaire 3, although still higher than the true figures.

Tool users were asked to rate the usability of ASSIST, with 100% of users responding. 4.8% found it extremely usable, 33.3% found it fairly usable 23.8% found it average, 33.3% found it fairly unusable and 4.8% found it totally unusable. This result is poorer than that for those who used ASSIST to inspect `analyse.cc`, and may indicate that the difficulty of `graph.cc` affected users perception of the tool.

Questionnaire 4 ended with some general questions about subjects' overall performance. When asked to rate their understanding of software inspection, 7.5% of subjects believed they understood it completely, 32.5% understood it well, 55% understood it reasonably well, while 5% were slightly unsure. Concerning their knowledge of C++, 80% of subjects believed it to be adequate for the tasks set and 20% believed it to be inadequate.

Subjects were then asked several questions comparing tool-based and paper-based inspection. When asked to compare their use of the checklist during each inspection, 37.5% stated they used it more during paper-based inspection, 52.5% used it about the same amount with both methods and only 10% used it more with ASSIST. This is probably due to the lack of screen space during a tool-based inspection. Of the three large documents which inspectors must use (product, specification and checklist), the checklist was probably regarded as the least important by subjects, and hence its window would be left closed most of the time.

Subjects were asked to indicate their preference for screen-based and paper-based documents. 15% stated a preference for screen-based, 20% had no preference and 65% preferred paper. This overwhelming preference for paper is undoubtedly due to its familiarity and perceived flexibility (e.g. ability to write comments on code, ability to spread documents out as required, etc.).

The question "Overall did you feel you performed better during individual inspection using manual (paper-based) inspection or ASSIST, or were you equally effective with both methods?" was asked. 39% of respondents claimed to have performed better using paper-based inspection, 39% had no preference, while 22% claimed to have performed better with ASSIST. The low preference for ASSIST probably stems from the familiarity of paper, which people are comfortable with. It is possible that extended training with ASSIST would increase its user acceptance. When the same question was asked with regard to the group meeting, the number of people preferring paper-based inspection dropped to 19.5%, the number with no preference increased to 61% and the number preferring ASSIST dropped to 19.5%. This change is probably due to the perception of the group meeting being an easier task than the

individual phase. Collating defect lists is presumably easier than finding the defects in the first place, therefore the method used to perform the collation task is less important. There is no clear correspondence between preferences for individual and preferences for group: some people who expressed a preference for paper-based individual inspection then went on to select ASSIST for the group meeting, others always preferred paper-based or always preferred ASSIST, yet others moved from preferring paper-based to no preference or from preferring tool-based to no preference.

The qualitative statements indicating preference made interesting reading. People who indicated a preference for paper-based inspection generally liked the tactile nature of paper, allowing them to scribble notes on the code itself. Others simply preferred reading code on paper instead of on-screen. A number of people found it awkward moving between the code, specification and checklist windows of ASSIST. While the scribe's burden is reduced in ASSIST, one subject commented that it was *too* easy to propose defects and put them into the master defect list, and therefore the phrasing of the defect was not usually considered as much as when the scribe had to manually write it down.

People who preferred ASSIST pointed out the following advantages. It was easy for the group as a whole to see exactly where individuals' defects were, and it was also considered easier to compile the master defect list, giving more time for the group to search for further defects. Others found it easier to traverse the code, and a number of people preferred the defect creation/editing facilities. The voting method for resolving defects was also considered to be useful.

People who expressed no preference also made interesting points. For example: “[It] was easy to look through code when it was on paper, but ASSIST has its advantages such as searching through the document for keywords...[During group meetings] paper-based inspection provoked more discussion, [but] ASSIST made it easier for [the] reader”.

5.2.4 Conclusions

The results from this experiment show that a straightforward computer-based approach to inspection does not degrade the effectiveness of the inspection in any significant way. There is no significant difference in the number of defects found. The number of false positives reported and the amount of meeting gains and losses were also measured, and again no significant difference was found. Although the experiment made use of student subjects and inspected short pieces of code, the inspection process used was realistic and the rate of inspection was typical of industry.

Some lessons about the facilities provided by ASSIST were learned. The defect proposal

and voting system was liked by subjects, and hence could be left unaltered. The speed of the tool appeared to be a concern among users, and methods of improving performance had to be investigated. Some facilities were obviously not being used to their full extent, such as the find facility. Hence, proper training is vital to ensure subjects are competent with the tool.

Suggestions for features to be added to ASSIST were also produced by the experiment. The first concerns checklist usage. The questionnaire results indicated subjects made more use of the checklist during paper-based inspection. An aspect of the tool which could therefore be improved is the ease of use of the checklist. It was also noticed that subjects using paper-based checklists marked items on the checklist as they completed them. Devising an equivalent of this for on-line checklists is therefore another line of research. Navigation around the document was deemed to be easier on paper. Again, this is a feature which could be improved in the tool. The number of windows used by the tool was a concern for most subjects, since all the windows could not be open on the screen at one time. This could be improved by minimising the number of windows used, making the windows more compact, providing resizable windows and providing facilities for moving between windows. In fact, the user interface may have had a biasing effect on the experiment if subjects disliked particular aspects of the tool. It may be that the effect of the interface of the tool obscures the comparison of methods.

If the inspection efficiency cannot be increased by finding more defects, perhaps the number of false positives can be reduced. This would increase the efficiency of the inspection meeting, as well as reducing the amount of time required by the author to tackle the defect list. Many false positives found in the experiment were due to subjects' uncertainty concerning aspects of C++. A standard C++ reference as part of the tool may help reduce this type of false positive.

Having established that the concept of tool supported inspection is not fundamentally flawed, various methods of increasing the efficiency of inspection, based on the results of this experiment, were investigated. The next chapter describes their implementation.

Chapter 6

Enhancing the Software Inspection Process

Having compared basic tool-supported inspection with paper-based inspection and found no significant difference between the two, the next stage was to investigate methods of enhancing the efficiency of tool-support. This investigation was based on feedback from the comparison experiment and requirements alluded to in various areas of the literature.

To begin with, navigation within documents and the number of windows used by the tool were found to be problematic during the first experiment. Automatic cross-referencing was thought to be a solution to these problems. Essentially, this facility would provide links between related areas of documents. Instead of scrolling a window to find the correct place, or finding the correct window, the link would allow the user to move directly to the relevant part.

Checklist usage during the first experiment appeared to be poor. Two features were designed to address this issue. Active checklists would allow users to mark off items as they were used, something which subjects were observed to perform with paper versions of the checklist. Active checklists could also make use of the cross-referencing facility to provide links between document features and appropriate checklist items.

As some of the false positives produced by subjects in the experiment appeared to stem from weaknesses in C++ knowledge, an on-line reference guide was an obvious addition. Combining this with an active C++ checklist and the cross-referencing mechanism could provide a complete C++ inspection environment. In this environment, features of C++ code could be automatically linked to relevant checklist items and reference entries.

Finally, a major task at an inspection group meeting is collating multiple defect lists into a single master list. Automating this task could reduce the length of the meeting, or even obviate the need for such a meeting. This was the final area examined.

The above features were implemented in a new version of ASSIST, with the intention of comparing the enhanced version with paper-based inspection. This chapter describes these features in detail.

6.1 Automatic Cross-referencing

The major difference between computer-based and paper-based inspection is that documents are now presented on-screen instead of being distributed as paper copies. Available screen space is a limitation to the usability of an inspection tool. Inspection requires the participant to frequently move between several documents, yet most displays in common use are not capable of simultaneously displaying several windows of sufficient size to be useful. This means that windows have to be frequently closed and reopened, or restacked, hindering the inspectors performance. This type of “window overload” was reported in questionnaire responses from subjects involved in the comparison experiment.

A feature which may help mitigate this problem is a cross-referencing system. Essentially, this would link related parts of each document together. If designed properly, it could provide a means of navigating within documents, as well as cross-referencing between documents. The major obstacle to such a system is the variety of document types that have to be dealt with: the strategy applied to English language documents is obviously different to that applied to source code, and even different source languages will have different strategies.

The open architecture implemented in ASSIST has already been described. It allows document-type specific browsers to be easily added, provided they comply with a well-defined interface. This feature provides the means to solve the cross-referencing problem: each browser can provide its own means of deriving references, and the standard interface can be used by the system to integrate and access these references.

The scheme developed consists of two stages. The first stage occurs when a new inspection is started. At this point each document to be used in the inspection has cross-references generated for it by the appropriate browser. These references are stored as part of the inspection data. The second stage occurs when an inspector uses the tool to perform part of the inspection. At that point, the references for all the documents used during the current phase are combined to form a cross-reference table. This means that inspectors are not given references to documents which are not in use during this phase. This cross-reference table is

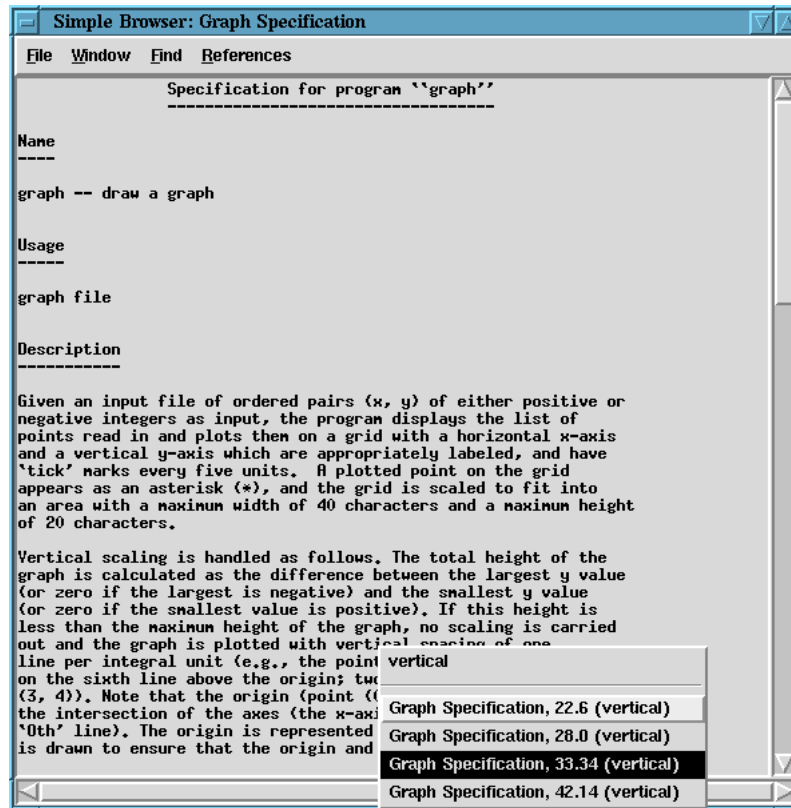


Figure 6.1: The ASSIST simple browser, showing a list of cross-references.

available to any browser through the standard interface, allowing lookups to be performed. How this information is used is entirely dependent on the browser in question, but the browser interface provides a call which, when given a document name and position, brings the appropriate window to the front of the screen (opening it if required), and scrolls the document to that position.

As an example, consider the simple browser provided in ASSIST. This is intended to be used for displaying ASCII source documents, more specifically English language documents. The cross-reference generator for this browser starts by dividing the text into words. Each word is then checked against a stop list of information-free words (such as “a” and “the”), and discounted if it appears. Many stop lists have been described, ranging from a few tens of words to hundreds of words [47]. The stop list used was [39]. If the word does not appear in the stop list it is stemmed to find the root of the word. Stemming allows words which are related but not identical to be linked [47]. For example, *calculation* would also reference *calculates* and *calculated*. The stemming algorithm used was the popular Porter algorithm [100]. The root, the whole word and its position are then added to the list of cross-references.

In use, selecting a word in the document brings up a list of references to similar words in this and other documents. Selecting a reference moves the document to the appropriate position or opens another browser, as required. Figure 6.1 shows the simple browser and a list of cross-references.

6.2 Active Checklists

Most software inspections make use of checklists. In its simplest form, a checklist consists of a list of items which the inspector must investigate. Each item is usually a simple statement or question designed to highlight typical ways in which defects may manifest themselves in the document under inspection. Checklists can be general, or they can be specific to a single document type, notation or language.

In their traditional guise, there is no requirement to actually answer the question or indicate completion of the item. The checklist used for the experiment described Section 5.2 is an example of this. In fact, many of the paper checklists returned by subjects after the experiment had markings obviously showing which items had been applied. A natural extension of this idea is to present the checklist in such a way as to require each item to be completed. This type of checklist has been implemented to a limited extent in InspeQ [66]. Details are sparse, however, and items can only be marked as completed.

Checklist items can obviously be written which require more than binary answers. Active Design Reviews [95] make use of questionnaires which inspectors have to fill in and discuss with the designer. It is obvious that answers may include numbers, sentences, dates, etc. In general, each checklist item would require the inspector to perform a task to answer it, the performance of which may lead to finding defects. Presenting these items on-line and storing the answers provided by each inspector is an obvious area which can be supported by an inspection tool.

A final possibility when checklists are presented on-line is to link them directly to areas of the document to which they may apply. This would promote checklist usage, since applicable checklist items can be easily found by users. For example, in a C++ checklist there may be items which must be applied to all `for` loops in the code under inspection. Sections 6.1 discussed a generic mechanism for cross-referencing between arbitrary documents. This can be used as a means to achieve references between checklist items and areas of the document. It can be implemented by allowing a keyword to be associated with each checklist item. The browser which displays the document can then “plant” corresponding keywords in appropriate places within the document. To continue the example, the C++ checklist items concerning

```

checklist ::= checklist_section+
checklist_section ::= heading checklist_item+
checklist_item ::= [subheading|multi|open|numeric|date|check]

```

Figure 6.2: The format of a checklist.

`for` loops would have a unique keyword. The browser displaying the code would then associate this keyword with each occurrence of the `for` loop, with the result that the system would automatically provide links between the checklist and code. The system would work in both directions: for each checklist item, all appropriate areas of the document could be listed, and for each area of the document all appropriate checklist items could be accessed.

To allow checklists to be easily created, a specific definition language was derived. A checklist is considered to consist of a number of sections, each of which consists of a heading plus one or more individual checklist items. A checklist item can be one of the following:

- **Subheading:** A subheading simply allows the checklist to be grouped in logically related sections.
- **Multi:** A multi is a multiple-choice question, where two or more predefined responses are supplied for the inspector to select from.
- **Open:** An open item allows textual answers to be supplied.
- **Numeric:** A numeric item is used when the answer is expected to be an integer.
- **Date:** Requires the answer to be in the form of a date.
- **Check:** A check item can only be marked complete.

The basic format of a checklist is defined in Figure 6.2. Each item type is defined in Figure 6.3, along with the definition of the checklist heading. The notation used is identical to that used in describing IPDL.

The checklist heading consists of the keyword `heading` followed by the heading itself. A subheading has a similar format. A multi consists of the keyword `multi` followed the question and two or more responses. An open question requires the keyword `open`, followed by the question and the maximum length of the answer required. In the same manner, a numeric question requires the keyword `numeric`, followed by the question itself and the maximum answer length. Furthermore, the units following the answer may be specified. A

<i>heading</i>	::= heading <i>string</i> <i>keyword</i> _{opt}
<i>subheading</i>	::= subheading <i>string</i> <i>keyword</i> _{opt}
<i>multi</i>	::= multi <i>question</i> <i>response</i> <i>response</i> ⁺ <i>text_answer</i> _{opt} <i>keyword</i> _{opt}
<i>open</i>	::= open <i>question</i> <i>length</i> <i>text_answer</i> _{opt} <i>keyword</i> _{opt}
<i>numeric</i>	::= numeric <i>question</i> <i>length</i> <i>unit</i> _{opt} <i>numeric_answer</i> _{opt} <i>keyword</i> _{opt}
<i>check</i>	::= check <i>question</i> <i>check_answer</i> _{opt} <i>keyword</i> _{opt}
<i>date</i>	::= date <i>question</i> <i>date_answer</i> _{opt} <i>keyword</i> _{opt}
<i>length</i>	::= length <i>integer</i>
<i>unit</i>	::= unit <i>string</i>
<i>text_answer</i>	::= answer <i>string</i>
<i>numeric_answer</i>	::= answer <i>integer</i>
<i>date_answer</i>	::= answer DD"/"MM"/"YYYY
<i>check_answer</i>	::= answer [yes no]
<i>question</i>	::= <i>string</i>
<i>response</i>	::= <i>string</i>
<i>string</i>	::= “'” <i>character</i> ⁺ “'”
<i>character</i>	::= Any printable character or white space.
<i>integer</i>	::= Any standard integer.

Figure 6.3: The definition of checklist items.

date question requires the keyword `date` followed by the question. Similarly, a check item only requires the keyword `check` followed by the question to be specified.

Each item may have a “correct” answer associated with it. This answer may be used for ensuring the checklist has been completed correctly. The answer is indicated by the keyword `answer` followed by the answer itself, in the appropriate format. Finally, each item may have a keyword associated with it. This keyword is designed to be used for cross-referencing with other documents, as described above.

This checklist language was implemented in ASSIST as the Checklist Browser. It is shown in Figure 6.4 displaying the C++ checklist used in the first comparison experiment. In this case, each item is a simple check, which can be marked as completed when the inspector has finished with that item. Items requiring more complex answers have associated spaces in which to type the answers. Clicking on an item gives a menu with cross-references for that item. Selecting a reference brings the appropriate window to the front, scrolls it to the correct position, and highlights the referenced item, just as described in Section 6.1.

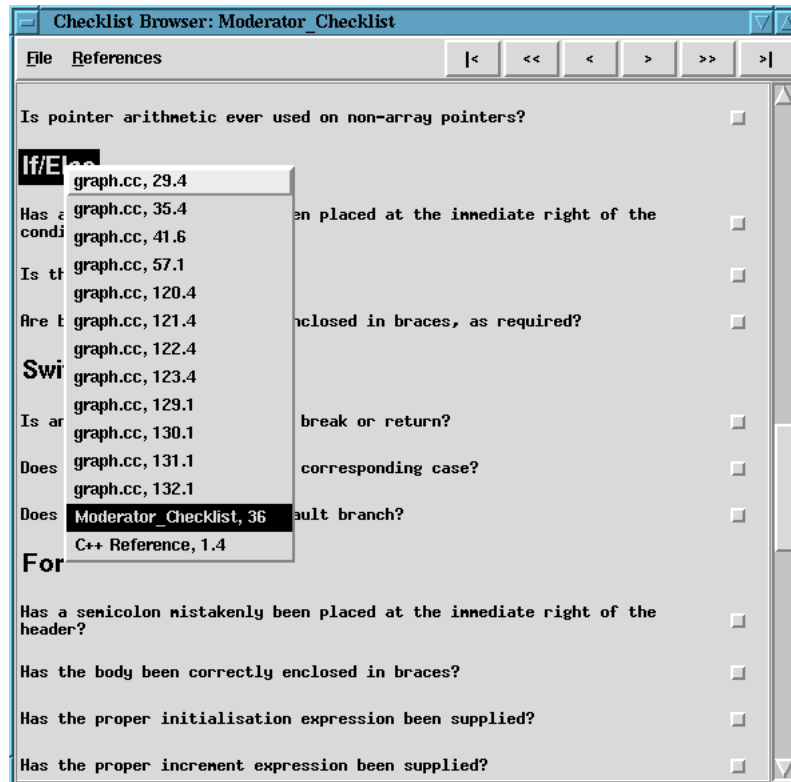


Figure 6.4: An example of an active checklist.

6.3 A C++ Inspection Environment

The cross-referencing system and active checklists described so far are applicable to inspection of any document type. With a view to repeating the paper vs. tool experiment, their use in inspecting C++ was investigated. The browser used for displaying the code under inspection in the original experiment was a simple generic browser with no specific facilities. A C++ specific browser, with knowledge of the language and syntax, was an obvious avenue to explore. Feedback from the first experiment also gave information on ways to improve the system in general. When designing this browser, the following features were deemed to be necessary:

- Links between variable declarations and usages. This would allow users to easily determine the type of variables, how they have been initialised, and where in the program they are used, hence supporting defect detection.
- Links between function declarations and usages. Users could determine whether function calls are being made correctly, and where in the program each function is used.

Again, this would help support defect detection.

- References to appropriate checklist items. This would promote checklist usage, a perceived weakness in the first version of ASSIST.
- An on-line guide to C++, preferably linked to the code. The type of false positive reported by subjects during the first experiment suggested a guide to C++ would be useful.
- Character-level annotation. The browser used in the first experiment could only annotate a single line. The ability to annotate any contiguous block of text would allow more precise positioning of defects.

To provide useful information about the code under inspection requires intelligent parsing of the code. Since writing a C++ parser is a non-trivial task, a public domain C++ parser, `cppp` [26], was used. This parser creates an abstract syntax graph of the program which allows information about the major constructs in the code to be more easily extracted.

For each variable and function in the source code, `cppp` assigns a unique identifier. This identifier is placed in the syntax graph whenever the variable or function is referred to in the source code. The C++ browser parses the tree looking for such identifiers and creates references to them for the appropriate place in the source code. When completed, this gives a complete cross-reference table for all functions and variables. To the user, this information is available by clicking on a function or variable name. Doing so brings up a list of references to all other uses of that object. Selecting a reference moves the browser to the appropriate place and highlights the occurrence. The C++ parser also provides information about the types of each variable, and this information is shown in the reference list. This allows inspectors to easily decide whether implicit type conversion is happening during an expression, and whether data is being lost as a result.

Creating references to checklist items is fairly straightforward. The checklist used in the first experiment was rewritten in the checklist format described above. Each item in the checklist was given a unique keyword. For each C++ feature in the syntax graph, the C++ browser inserts the appropriate keyword in the reference list for the corresponding location in the source code. For example, the C++ keywords `while` and `for` have references to the checklist items concerning the corresponding loop type, while all operators have links to the items concerning order of evaluation, implicit type conversion and so on. Some checklist items were also supplemented by supplying extra information concerning how to use the tool to apply that item. For example, the item concerning 'dead code' had an extension suggesting the use of the cross-referencing mechanism and the find facility.

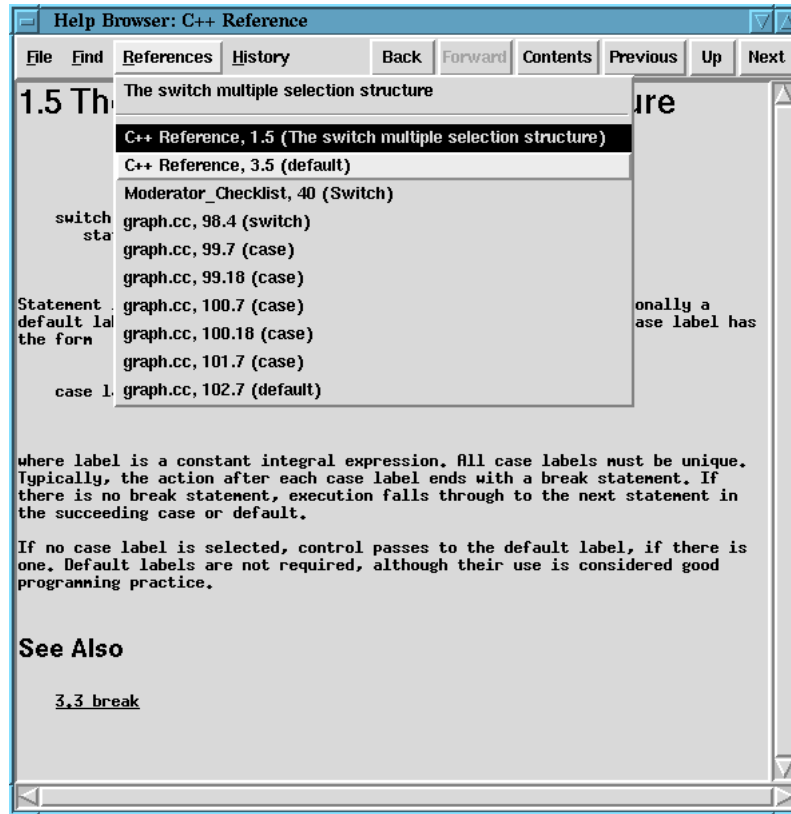


Figure 6.5: The C++ reference, displayed using the Help Browser.

Another goal was to have some form of on-line C++ reference. Rather than incorporate this directly into the C++ browser, a more generic form was chosen. Another browser was created, known as the Help Browser. This browser implements a subset of HTML tags and allows a document to be presented as a series of sections and subsections, each of which is displayed as single page. It also generates a contents page and allows keywords to be associated with each page. This allows references between areas of code and pages in the Help Browser. A short C++ guide was then written, with pages on major constructs, types and keywords, and including information on the standard library. Each page has a keyword associated with it. When the C++ browser parses the code under inspection, it inserts these keywords into the reference list for appropriate places in the code, in the same manner as for checklists. For example, if a `#include` directive refers to one of the standard libraries, the keyword associated with that library is generated at the appropriate position. When a `while` loop is detected, the keyword for the `while` construct page is inserted. Therefore, when the cross-reference data for the code and C++ guide are combined, pages in the guide provide links to parts of the code to which they apply. Similarly, C++ reserved words like `for` and

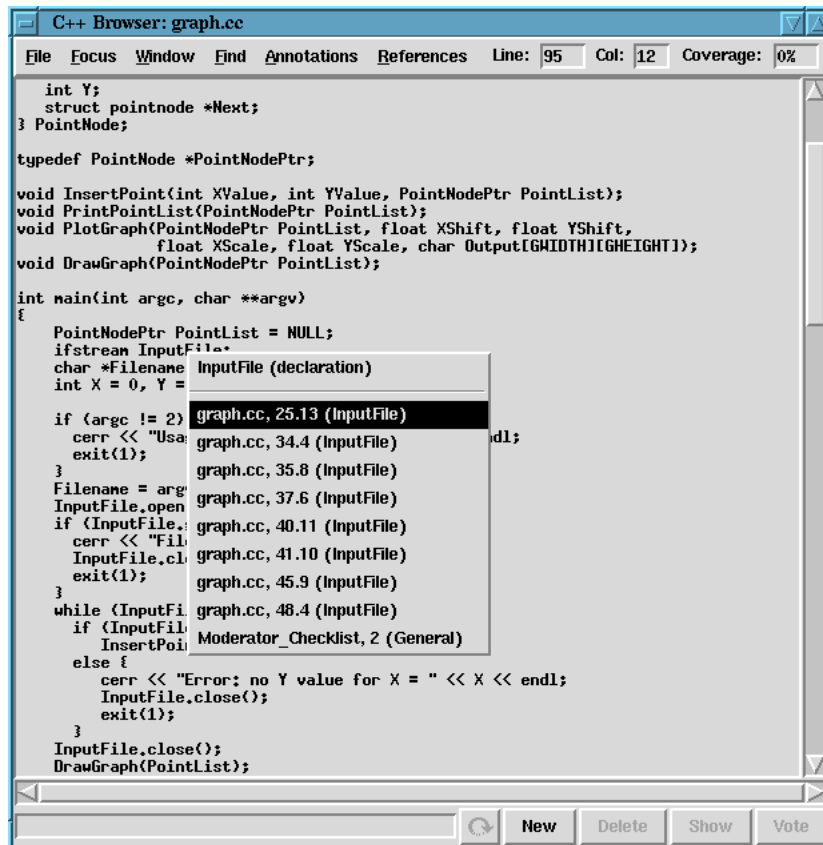


Figure 6.6: The C++ browser.

while have links to the corresponding pages of the guide. The Help Browser displaying the C++ guide is shown in Figure 6.5. This shows the page on the `switch` statement, with a list of references to other relevant pages in the reference, checklist items and appropriate sections of the code under inspection.

The C++ browser creates all internal references, links to the checklist and links to the C++ reference with one pass through the syntax graph produced by `cppp`. Unfortunately, `cppp` lacks some features to allow truly comprehensive cross-referencing. For example, when `cppp` encounters a named constant it substitutes the value of the constant rather than leaving the name. Hence, it is impossible to provide a link between the use of a constant and its declaration. For the purposes of tool evaluation, these links can be (and were) inserted manually. A more complete C++ parser would allow everything to be performed automatically.

Finally, several other enhancements were made. The browser used in the first experiment could only annotate at the line level, with defect positions being given as line numbers. To allow more accurate defect positioning, it was deemed necessary to allow annotation at the

character level. Feedback from the first experiment also showed that subjects became confused during the group meeting because the browser showed defects belonging to all participants, and it was not clear who had prepared which annotation. Hence, another facility implemented allowed users to choose between displaying all annotations on the code, only their annotations, or no annotations at all. The C++ browser is shown in Figure 6.6. A list of references for the declaration of the variable `InputFile` is shown. These include places in the code where the variable is used and a checklist item.

6.4 Automatic Defect List Collation

The group meeting present in almost all inspection processes is expensive to set up and run, requiring the simultaneous participation of three or more people. Some would argue that their cost is unjustified and they should be replaced altogether. For example, Votta [118] presents evidence that meeting costs outweigh their benefits and suggest their replacement by depositions. Asynchronous inspection via a supporting tool is another proposed method for replacing synchronous meetings, e.g. [83, 117]. Proponents of group meetings, on the other hand, contend that the benefits of group meetings are not easily quantified. Education of new inspectors is one quoted benefit, while synergy, found in many small group situations [105], is another.

A major component of the group meeting is collating individual defect lists into a single master list. To reduce the amount of effort required at the meeting, Humphrey [50] recommends that the moderator should perform the collation before the meeting. The collated defect list then becomes the agenda for the meeting. Although not particularly demanding, collating defect lists can be time consuming. This section describes automatic defect list collation (auto-collation), designed to allow multiple lists of issues or defects to be combined into one with duplicate entries automatically removed.

It is unlikely that duplicate items from different inspectors will be identical, hence some form of approximate matching must be used. A defect or issue in ASSIST has several components (Figure 6.7):

- The title of the item.
- The document in which this item occurs.
- The position within the document where the item occurs.
- A free-form text description of the item.

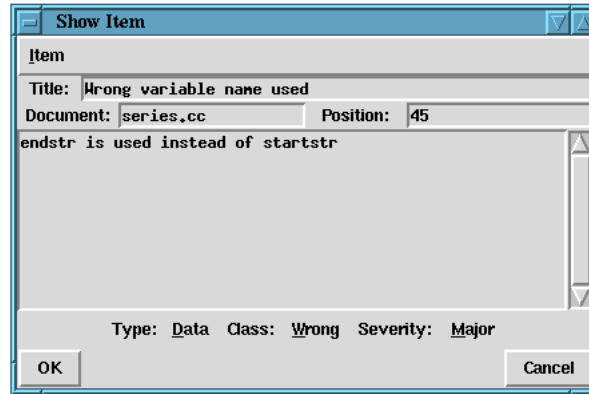


Figure 6.7: Editing an item in a list.

- Up to three levels of classification.

When comparing items for duplicates, document name and position are considered together as the position, the title and text description are considered together as the content of the issue, and the classification is considered on its own. The mechanism used is to score items on their similarity in each of these facets. If two items match with a score above a threshold, one of the items is discarded.

In terms of position, the closer the physical locations of the two items, the higher the score is given, with 0 representing no match and 1 representing identical positions. Items occurring in different documents are given a score of 0. Positions in ASSIST are given as one or more integers, separated by points if necessary. For example, a simple line number is given as 23. A specific character on that line may be 23.12. The scheme can be extended to as many levels as required, and its hierarchical nature allows it to be used for many document types. For example, chapter and section numbering in English documents follows this scheme. When performing a comparison, each component of the position is weighted according to its importance, with leading numbers being more significant.

When comparing the contents of two items, the first step is to generate a list of words occurring in each item. Any words appearing in the stop list (identical to the one described in Section 6.1) are removed, since these words contribute little to the meaning of the contents. The remaining words are stemmed, again using Porter's algorithm. The two word lists are compared and the number of common words found, expressed as a fraction of the total number of words in the smaller list, gives a score of between 0 and 1. The more words which are common to both items, the higher the score.

The three classification levels also contribute to the similarity score, with the levels scaled in the ratio 2:1:1. This allows one classification factor to be given importance over the other

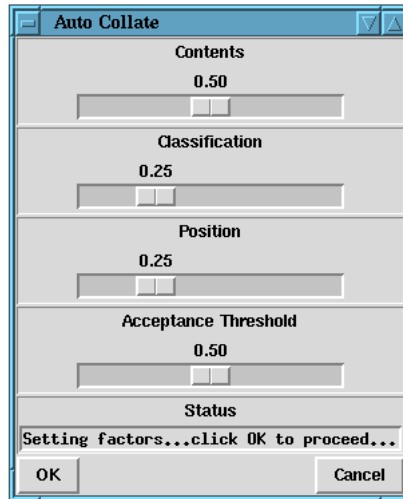


Figure 6.8: Auto collation of defect lists.

two. Each classification level in one item is checked with the corresponding level in the other, and a simple binary decision made whether they match or not. Checking all three levels gives a score between 0 and 1.

To allow more flexibility when performing the auto-collation, the relative importance of each facet can be altered. Figure 6.8 shows the control window for the implementation of auto-collation in ASSIST, which allows the various factors affecting the auto-collation to be set. The **Contents**, **Classification** and **Position** values indicate the relative importance of the each facet when calculating the similarity between two items. The total of all these factors must sum to 1, hence increasing (decreasing) one factor decreases (increases) the others. The similarity value is calculated by multiplying the individual values by the appropriate factor, adding them together, then scaling to give a value between 0 and 1.

Finally, the similarity value for two items must be compared with a value used to determine whether the items are sufficiently similar to be declared duplicates. This value is the **Acceptance Threshold**, and it can be defined by the user. The higher the threshold value, the more similar two items must be to be declared duplicates. Too high a threshold, however, will result in no matches being made.

An initial investigation into the use of auto-collation proved encouraging, and it appeared to be an efficient way of removing duplicate items. As can be imagined, however, the set of values used as factors has a huge effect on the result of the auto-collation, with the outcome ranging from discarding virtually all items to performing no removals. Hence, a more rigorous experiment was performed to determine the ranges within which these factors are most usefully set. This experiment is described in Section 7.2.

The resulting system also provides a new format for the inspection meeting. Each participant can prepare in their own time in the normal way. Prior to the meeting, one participant can use the auto-collation mechanism to combine defect lists. At the group meeting only one participant needs to access the tool, to show documents and edit the master defect list. This participant essentially takes the role of the reader and scribe, which is possible since both roles are less arduous with the help of computer support. The display from the single tool can be projected for all participants to see. This makes the best use of the tool both in individual preparation and as an administrative aid during the meeting, but also allows the meeting to proceed in a traditional, face-to-face way. Furthermore, there is no need to have multiple computers arranged in an appropriate way in a communal room.

6.5 Conclusions

The basic version of ASSIST used in the comparison experiment showed no significant difference between the effectiveness of tool-based and paper-based inspection. This system could then be used as a vehicle to explore facilities for enhancing the inspection process. Several such facilities were identified from the first experiment. A method of cross-referencing both within and between documents was identified as a means of assisting users in managing the large number of windows which have to be used in an inspection support tool. Active checklists are designed to promulgate checklist usage amongst users. Along with the cross-referencing system and a C++ code browser, they have been used to create a C++ inspection environment. This environment links checklist items with relevant areas of the code. It was hoped that this more streamlined environment would prove more user-friendly, allowing users to concentrate on the task of finding defects. At the same time, the new features such as variable tracing should help with the detection of defects. Finally, collating defect lists was identified as a time-consuming task. This has been addressed by developing a method for automatically collating defect lists while removing non-identical duplicate items.

Several miscellaneous improvements suggested by users were incorporated into the new version of ASSIST. Originally, many of the windows used by ASSIST did not resize properly, and some did not resize at all, hence users had to deal with a number of large windows cluttering the screen. All windows were now properly resizable. Several speed improvements were made to reduce the amount of time users spent waiting for the tool, which caused frustration in the first experiment. These, combined with an upgrade to the machines used to run the experiment, made a faster environment for the user.

To evaluate the new version of ASSIST, implementing the enhanced features described in

this chapter, it was decided to re-run the comparison experiment described in Chapter 5. This experiment would compare paper-based inspection with the enhanced version of the tool. To assess the efficiency of the auto-collation mechanism, it was decided to carry out a rigorous experiment using data from both comparison experiments. The next chapter presents these experiments in detail.

Chapter 7

Evaluation of Enhanced Tool Support

Having investigated various methods of improving the efficiency of inspection, the next step was to evaluate their impact. This chapter describes a controlled experiment comparing paper-based inspection with inspection using a version of ASSIST implementing the C++ inspection environment described in the previous chapter. The environment provides cross-referencing within and between documents, and active checklists with links to relevant features in the code. This chapter also describes an investigation into the effectiveness of the auto-collation system.

7.1 Comparing Enhanced Tool-based and Paper-based Software Inspection

7.1.1 Introduction

To investigate the effectiveness of enhanced tool support, the experiment comparing tool-based inspection with paper-based inspection was re-run. This second experiment was identical in all respects, except the version of ASSIST used featured the support detailed in the previous chapter, namely the C++ browser, C++ reference, active checklists, and cross-referencing [70]. Feedback from the first experiment also resulted in a number of miscellaneous bug fixes, interface updates and feature enhancements.

The null hypothesis for this experiment, H_0 , is identical to that of the first experiment:

There is no significant difference in performance between individuals performing tool-based inspection and those performing paper-based inspection, measured by the total number of defects found during a given time period.

The alternative hypothesis, H_1 is also identical:

There is a significant difference in performance between individuals performing tool-based inspection and those performing paper-based inspection, measured by the total number of defects found during a given time period.

Similar hypotheses can again be formed regarding the performance of groups.

7.1.2 Experiment Design

The design of this experiment was identical to the previous experiment. Testing of the hypotheses requires two groups of subjects to inspect a single document, one making use of the tool and the other using paper. Subjects then swap methods and perform an inspection on another document, ensuring that any effect observed is not due to one group of subjects being of greater ability. The same inspection process was used, consisting of an individual detection phase, followed by a group collection meeting.

This experiment was carried out during Autumn 1997 in the same setting as the previous experiment. The subjects used were participating in the next session of the same team-based Software Engineering course for third year undergraduates, and had identical experience to their counterparts in the first experiment. Again, the practical aspects of the experiment formed part of their continual assessment for this course, increasing subject motivation.

A total of 49 students participated this time, split into two approximately equal sections. Section 1 had 24 subjects and Section 2 had 25 subjects. The split was achieved in an identical manner to that for Experiment 1. Section 1 consisted of eight groups of three students, while Section 2 contained seven groups of three students and one group of four students.

The same materials used for the first experiment were used in this experiment, presented in an identical manner. Again, the practical element of the course ran over ten weeks, with six weeks devoted to training in software inspection and using ASSIST, also providing an opportunity for the subjects to refresh their C++ knowledge. When each training program had been inspected, subjects were given a list of defects in that program. An extended version of the tool tutorial used in the first experiment was created, giving more information on how best to use ASSIST to find defects, and tackling the use of the new features. Four weeks were then used to run the experiment. The exact timetable is given in Appendix D.1. Each practical session was run twice, once for each section of the class. Both sessions occurred consecutively on the same afternoon of each week.

Instrumentation of the experiment was also identical. Each subject used the individual defect report form (Appendix D.3) during paper-based inspection, with the scribe making use

Program	analyse.cc		graph.cc	
Section	1	2	1	2
Method	Tool	Paper	Paper	Tool
Subjects	23	25	24	25
Mean	6.52	6.68	5.92	5.88
St. Dev.	1.83	1.77	1.53	1.33
St. Error	0.38	0.35	0.31	0.27
F Ratio	0.09		0.01	
F Prob.	0.76		0.93	

Table 7.1: Analysis of variance of individual defect scores.

of the master defect report form (Appendix D.4) for the group meeting. ASSIST was used to keep both individual and master defect lists during tool-based inspection. The data collected was similar to that of the first experiment: the total number of correct defects found, the number of false positives submitted (both for each individual and for each group), meeting loss and meeting gain for each group, and frequency of detection for each defect. Since this experiment is identical in all respects to the previous experiment, the threats to validity remain the same and are not repeated here.

7.1.3 Results and Analysis

Defect Detection

The raw data from this experiment can be found in Appendix E.1.2. Table 7.1 summarises the data and the analysis of variance of the individual phases of both inspections. The table shows the method used by each section of subjects for each program. The number of subjects participating in each phase, the mean number of defects found, the standard deviations and standards errors are shown for each treatment. Note that one subject from Section 1 failed to participate in the individual inspection of `analyse.cc`. The F ratios and probabilities for the comparisons of paper and tool are also shown.

For `analyse.cc`, there is very little difference in performance, confirmed by the analysis of variance. For `graph.cc`, the difference in performance is even smaller. Again, this is confirmed by the analysis of variance. For both programs the null hypothesis relating to individual performance cannot be rejected. The advanced features provided by ASSIST give no increase in performance over paper-based inspection. However, compared to the previous experiment there is much less of a difference between methods when applied to `graph.cc`.

The defect detection data for groups is summarised in Table 7.2. Paper-based inspection

Program	analyse.cc		graph.cc	
Section	1	2	1	2
Method	Tool	Paper	Paper	Tool
Groups	8	8	8	8
Mean	9.50	10.25	8.88	8.63
St. Dev.	1.69	0.89	1.25	0.74
St. Error	0.60	0.31	0.44	0.26
F Ratio	1.24		0.24	
F Prob.	0.28		0.63	

Table 7.2: Analysis of variance of group defect scores.

Effect	F Ratio	F Prob.
Order	$\ll 0.01$	0.96
Program	4.97	0.03
Order \times Program	0.22	0.64

Table 7.3: Analysis of variance of method order and program.

appears to outperform tool-based inspection on `analyse.cc`, but this difference is not statistically significant. The difference is probably due to a less favourable allocation of subjects to groups, with group members having less distinct subsets of defects. Another factor is meeting gain, which is higher for those performing a paper-based meeting (see later). The results for `graph.cc` show very little difference, with no statistical significance. In both cases the null hypothesis concerning group performance cannot be rejected.

All defect detection data passed the Levene test for homogeneity of variances. As a safeguard, all four sets of data were subjected to the Kruskal-Wallis non-parametric test, giving results similar to the parametric tests.

Table 7.3 shows the results of the analysis for effects stemming from the order in which methods were used and differences in the two programs. There is no significant difference between subjects performing tool-based inspection first compared to those who performed paper-based inspection first. As with the first experiment, however, there is a significant difference in the difficulty of each program. The post-experiment questionnaires showed that subjects in this experiment also regarded `graph.cc` as more complex than `analyse.cc`. Unlike Experiment 1, however, there is less of a performance deficit between methods when inspecting `graph.cc`. This contradicts the previous suggestion that the tool may become less efficient as the material under inspection becomes more complex. This may be due to the

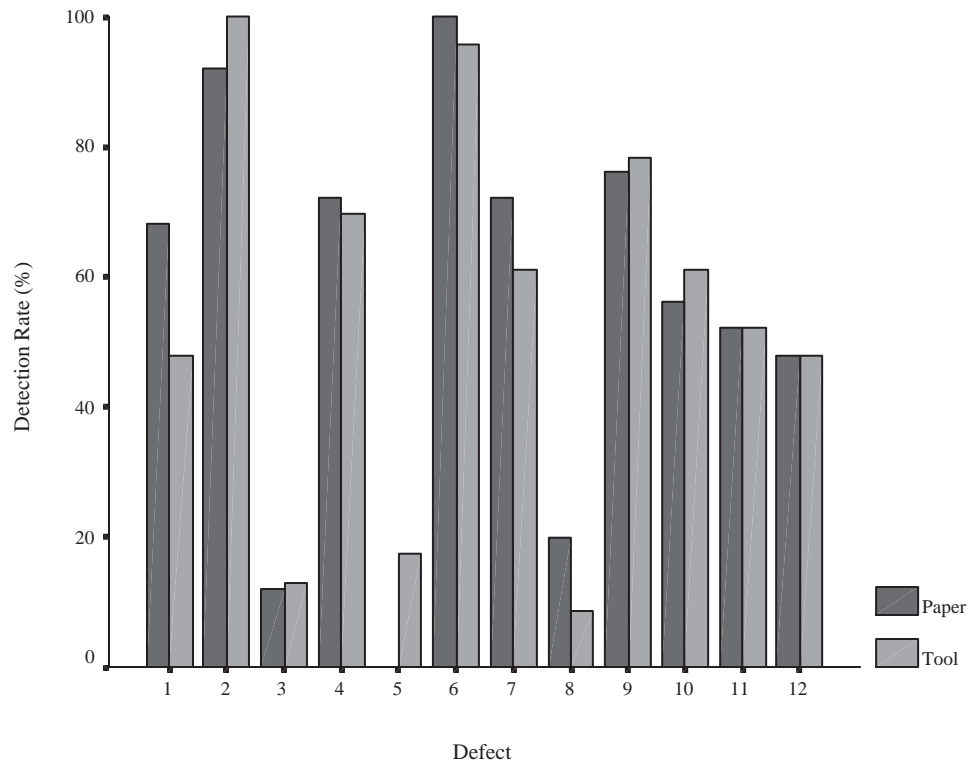


Figure 7.1: Summary defect detection data for `analyse.cc` (individuals).

extra facilities provided by the second version of ASSIST, but it is difficult to say for certain. An experiment comparing the basic version of ASSIST with the enhanced version may be able to determine if this is indeed the case. Finally, analysis for any effect due to the combination of order of method and program was performed, but no significant result was found.

The individual detection frequencies of each defect were analysed in the same way as the first experiment. Figure 7.1 summarises frequency of detection of each defect during the individual inspection of `analyse.cc`. The defects with the greatest differences in detection frequency are 1 and 5. Defect 1 (an array indexing error) was found by more subjects using paper-based inspection than by those performing tool-based, mirroring a similar result from Experiment 1. Defect 5 has a large difference in favour of the tool, which does not correspond to any similar result from the last experiment. This defect concerns a missing cast of a variable. A checklist item exists to cover both types of error, and in both cases ASSIST provides explicit cross-references between the checklist items and possible occurrences in the code. Hence, it is difficult to claim that the facilities of ASSIST are responsible for the increase in detection frequency for defect 5, when they should at least prevent the tool from performing worse

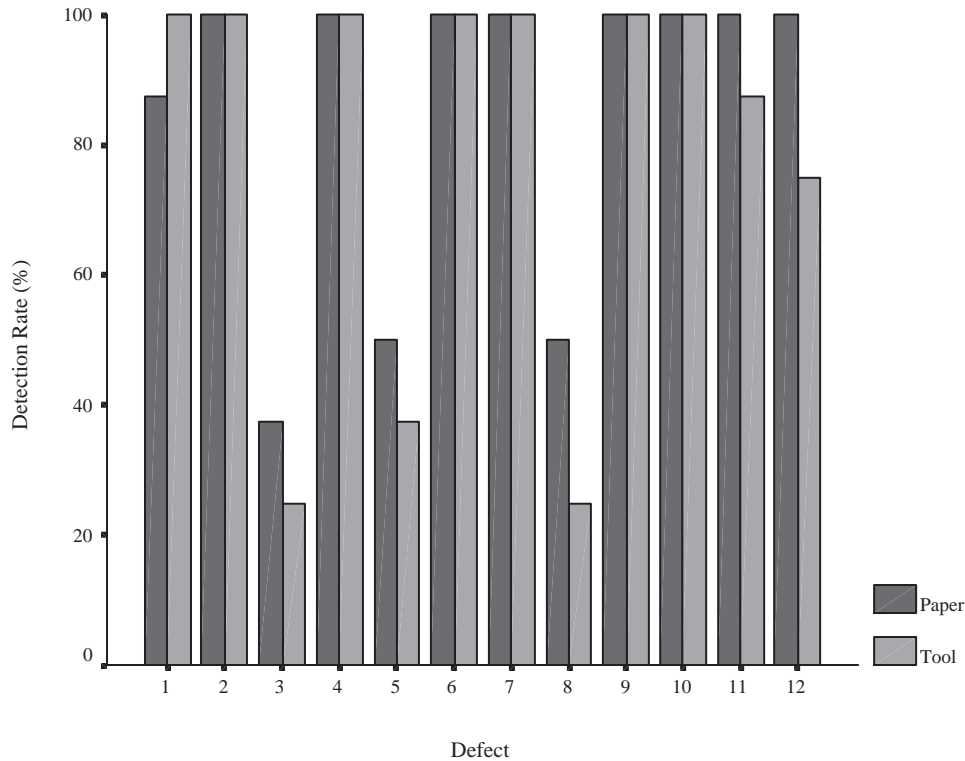


Figure 7.2: Summary defect detection data for `analyse.cc` (groups).

on defect 1. Defect 1, however, may be considered as requiring more thought: a thorough understanding of the code is necessary to detect it. The facilities of ASSIST, on the other hand, lead the inspector straight to defect 5 with little need for understanding.

Figure 7.2 shows the frequency of detection for the `analyse.cc` group meeting. Compared with the first experiment, there are more variations between methods, but these variations are smaller. There are no clear correspondences between the results of the two experiments. Differences for two defects stand out: 8 and 12. There are no parallels with Experiment 1. Defect 8 is a calculation error, while defect 12 is a loop error. Both were found by more paper-based groups than tool-based, but there is no correspondence with the results from the individual phase. Instead, these variations are due to the allocation of subjects to groups. A different group allocation could have reduced the disparity, and could also have increased it (meeting gains and losses have no effect on these particular defects).

The defect detection frequencies for the individual inspection of `graph.cc` are shown in Figure 7.3. Overall, the profiles for each method are very similar, with the exception of defect 10, which has an above average advantage in favour of the tool. This defect concerns

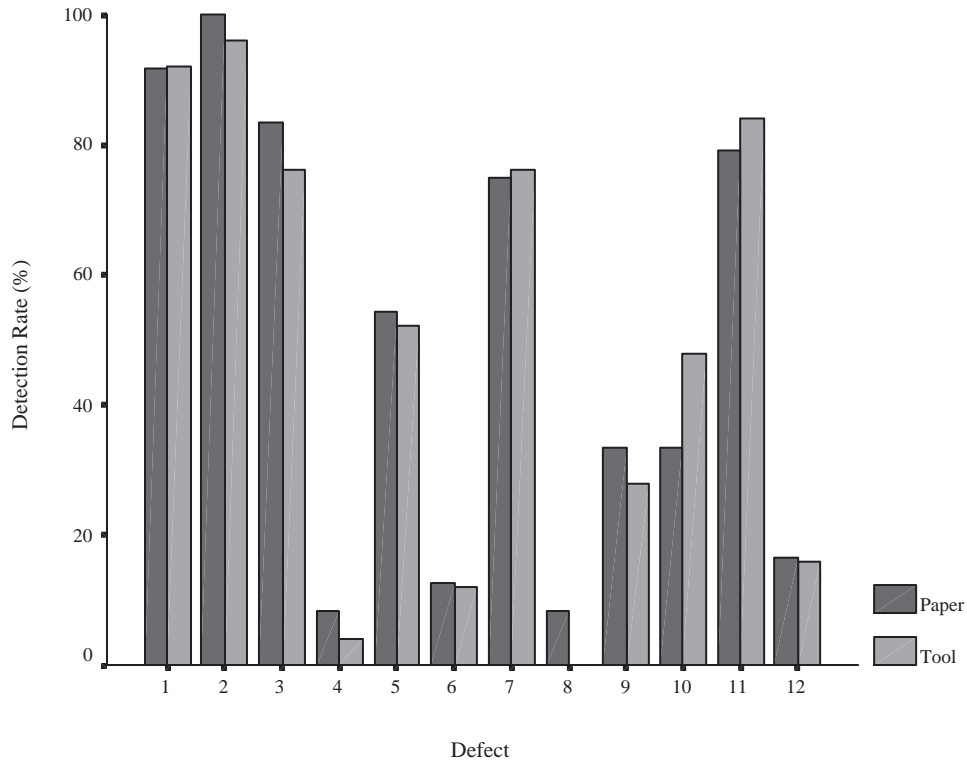


Figure 7.3: Summary defect detection data for `graph.cc` (individuals).

an incorrect initialisation, for which there is an explicit checklist item. In the case of the tool, each variable declaration has an explicit reference to the checklist item, which may account for the difference. This theory is supported by the fact that no such difference occurred in the previous experiment, where the tool did not support cross-references between the code and checklist. While the large difference observed in favour of paper-based inspection for defect 3 in the first experiment has been reduced, there is still a slight difference. Frustratingly, this defect is even easier to find with the second version of the tool: clicking on a function name gives a list of lines where that function is used. Non-use of a function results in an empty list. Finding this defect cannot be made any simpler! The obvious conclusion is that subjects need more time to become familiar with all aspects of the tool and how they are used to find defects.

Finally, the defect detection frequency of the group meeting with `graph.cc` is shown in Figure 7.4. Two large differences occur, one for defect 8 and another for defect 12. As with the group results for `analyse.cc`, there is no correspondence with the results from the individual phase, but defect 8 had a large difference in the group phase of the previous

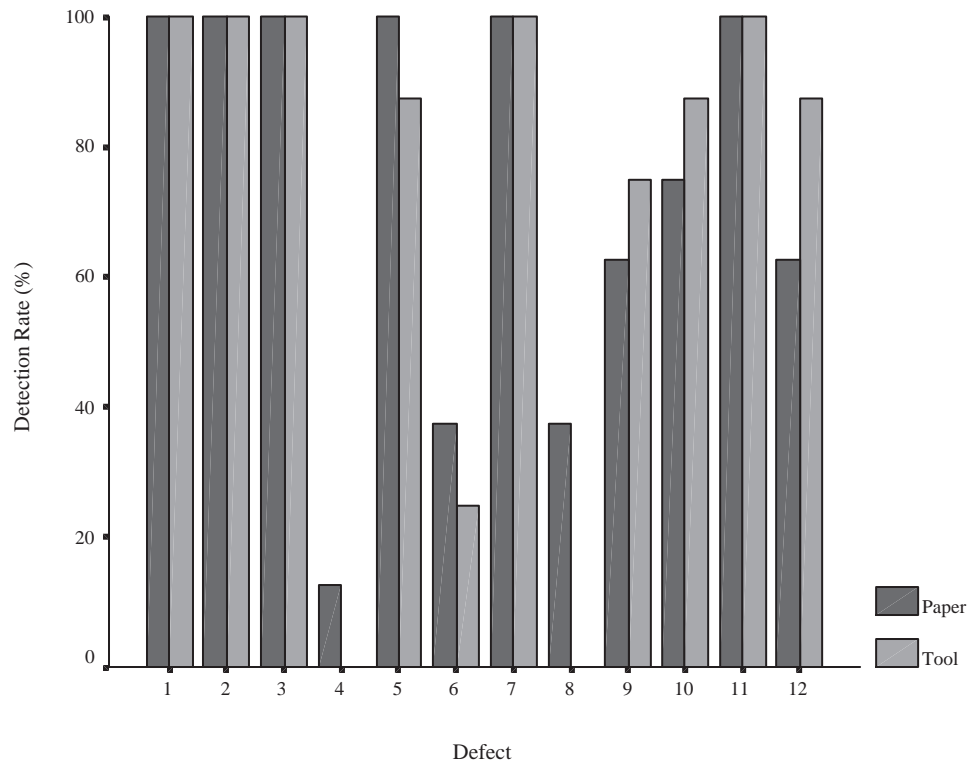


Figure 7.4: Summary defect detection data for `graph.cc` (groups).

experiment. As with the previous experiment, however, this difference is due to the allocation of subjects to groups. This is also true for defect 12.

False Positives

The number of false positives found by each subject was collected, to try to determine if tool-supported inspection reduces the amount reported. This might be the case if subjects used the on-line C++ reference to check features of C++ which they were unsure of. The analysis of variance of false positives from individual inspection is presented in Table 7.4. For `analyse.cc` there is a significant difference at the 5% level, with tool-supported inspection producing fewer false positives. This trend is not continued with `graph.cc`, indeed it is reversed, although the difference is not significant. From these results it could be hypothesised that subjects in Section 1 were less prone to false positives, although there is no clear reason why this should be so.

Table 7.5 shows the results for false positives reported by groups. For both programs there is no significant differences in the numbers reported, although groups in Section 1 consistently

Program	analyse.cc		graph.cc	
Section	1	2	1	2
Method	Tool	Paper	Paper	Tool
Subjects	23	25	24	25
Mean	4.13	6.20	2.79	3.36
St. Dev.	2.90	2.77	1.44	2.00
St. Error	0.60	0.55	0.29	0.40
F Ratio	6.40		1.29	
F Prob.	0.02		0.26	

Table 7.4: Analysis of variance of individual false positives.

Program	analyse.cc		graph.cc	
Section	1	2	1	2
Method	Tool	Paper	Paper	Tool
Groups	8	8	8	8
Mean	3.88	5.00	2.38	2.88
St. Dev.	1.55	2.14	0.92	1.46
St. Error	0.55	0.76	0.32	0.52
F Ratio	1.45		0.68	
F Prob.	0.25		0.42	

Table 7.5: Analysis of variance of group false positives.

produce fewer false positives than their counterparts, as could be predicted from the results for individuals.

As with the first experiment, there was no obvious patterns in the types of false positives produced by each method. Again, some were obviously defects which had occurred in training material which subjects had memorised and submitted because similar code was found in the experiment programs.

Meeting Gains and Losses

Finally, meeting gains and losses are examined. As with the previous experiment, it was hypothesised there would be no difference between tool and paper-based methods for both gains and losses. Even though the version of ASSIST used by the subjects had the list auto-collation facility, it was disabled for the experiment. Evaluation of this facility can be found in Section 7.2. Table 7.6 shows the analysis of variance for meeting gains, while Table 7.7 shows the analysis of variance for meeting losses for each inspection. As with the first experiment,

Program	analyse.cc		graph.cc	
Section	1	2	1	2
Method	Tool	Paper	Paper	Tool
Groups	8	8	8	8
Mean	0.25	0.63	1.25	0.38
St. Dev.	0.46	0.74	1.16	0.52
St. Error	0.16	0.26	0.41	0.18
F Ratio	1.46		3.78	
F Prob.	0.25		0.07	

Table 7.6: Analysis of variance of meeting gains.

Program	analyse.cc		graph.cc	
Section	1	2	1	2
Method	Tool	Paper	Paper	Tool
Groups	8	8	8	8
Mean	0.25	0.25	0.38	0.38
St. Dev.	0.46	0.46	0.52	0.52
St. Error	0.16	0.16	0.18	0.18
F Ratio	0		0	
F Prob.	1.00		1.00	

Table 7.7: Analysis of variance of meeting losses.

there was no statistically significant difference between methods, despite paper-based inspection of `graph.cc` appearing to produce larger meeting gains than tool-based. Although this difference is not significant, there is evidence in the questionnaires completed by subjects (presented next) that tool-based inspection provided less opportunity for the group to engage in additional defect detection.

When individual defects are considered, there are three points of interest. Defect 5 of `analyse.cc` provided the largest meeting gain for that program. Only one gain was due to tool-based inspection; the other four belonged to paper-based groups. This defect was found by four individual tool users, but no paper-based subjects. So despite the tool apparently enabling users to find this defect more easily, this is not translated into the meeting, indicating a lack of defect detection during tool-based meetings. Paper-based meetings, however, allowed subjects to find more instances of this defect, even though no individual found it. Considering `graph.cc`, one meeting loss (concerning defect 4) and one meeting gain (concerning defect 12) are of interest. Defect 4 was lost by three groups (two paper, one tool). This is one of the least well-reported defects, It is likely that subjects who found this defect on their own

were dissuaded from reporting it by other group members. Defect 12 was gained by three tool-based groups and three paper-based groups, far more than any other. This defect means that the program prints the graph upside-down. While quite a difficult defect to find, it is very memorable. Hence, it is suspected that collusion between subjects resulted in the spread of this defect among other groups. Unfortunately, since a week elapsed between the individual phase and the group phase, such collusion is practically inevitable, especially when subjects are motivated to gain the highest marks possible.

Debriefing Questionnaires

The four questionnaires distributed during the first experiment were also used during this experiment (with minor updates). Again, Questionnaire 1 (91.8% response rate) was given after the first full individual practice with ASSIST, while Questionnaire 2 (97.9% response rate) was given after the first full group practice session. Questionnaires 3 (93.9% response rate) and 4 (95.9%) were distributed after the first and second phases of the experiment respectively. The full text of the questionnaires can be found in Appendix D.11.

Questionnaire 1 With respect to ease of navigation around the document under inspection, no subjects found it very difficult, 2.3% of subjects said they found it difficult, 33.3% said they found it average, 55.5% found it easy and 8.9% said they found it very easy. These results compare favourably with those from the previous experiment (3.4%, 20.7%, 24.1%, 37.9% and 13.8% for the respective categories). When asked whether the number of windows used by ASSIST had any effect on their inspection efficiency, 20% of subjects said it increased their efficiency, 44.4% said it had no effect and 35.6% thought their efficiency was reduced. Again this compares well with the previous experiment (13.8%, 31% and 55.2%). Taken together, these results suggest an increase in the general usability of ASSIST compared to the previous version.

An increase in acceptance of tool-based inspection was seen when subjects were asked which mode of inspection they thought more efficient. 24.4% of subjects thought tool-based less efficient, 33.3% thought it equally efficient and 42.2% thought it more efficient. The corresponding results from the first experiment were 55.2%, 26.6% and 17.2% respectively, giving a large swing in opinion. Although there is no quantifiable increase in performance from subjects, it is clear that the improved version of the tool is more acceptable.

Questionnaire 2 When asked to rate the usability of the defect proposal system used at the meeting, 4.2% of respondents found it very difficult to use, 4.2% found it difficult, 37.5%

found it to be average, 35.4% found it easy to use and 18.8% found it very easy to use. These figures are roughly comparable with those from the previous experiment (0%, 14.3%, 38.1%, 28.6% and 19% respectively), which is unsurprising since the defect proposal mechanism was unchanged from the first version of ASSIST. More surprising was the response to the question concerning the usefulness of the voting mechanism. 12.5% of respondents thought it hindered the resolution of issues, 64.6% thought it had no effect and 22.9% thought it helped. The corresponding figures from Experiment 1 were 14.3%, 33.3% and 52.4%. Around the same proportion of respondents thought it hindered issue resolution, but far fewer people thought it helped resolve issues.

Subjects rated the effect ASSIST had on their group meeting. 14.6% of subjects thought it had a large negative effect, 22.9% felt it had a negative effect, 18.8% considered it to have no effect, 35.4% felt it had a positive effect and 8.3% thought it had a large positive effect. The corresponding figures from the previous experiment are 4.8%, 19%, 38.1%, 38.1% and 0%, showing no real trend in terms of change of opinion, other than fewer people having a neutral opinion. When asked to compare the efficiency of tool-based and paper-based meetings, 33.3% thought tool-based less efficient, 27.1% thought it equally efficient and 39.6% thought it more efficient. When compared to the figures from the first experiment (38.1% 47.6% and 14.3%), there is a definite positive swing in favour of tool-based meetings. The reason for this change in opinion is probably due to the facilities designed for individual inspection (the active checklist, cross-referencing, etc.) also being found to be useful during the group meeting.

Questionnaire 3 When questioned about their understanding of the `analyse.cc` code, 8.7% of respondents reported understanding 41-60%, 32.6% understood 61-80% of it, and 58.7% reported understanding 81-100% of it. The corresponding figures for the first experiment were 5%, 25% and 70%, showing a slight overall decrease in understanding of the program. 10.9% of subjects thought there was not enough time to inspect the code, 82.6% thought there was just enough time and 6.5% thought there was too much time. These are very similar to the figures from the first experiment (5%, 87.5% and 7.5%).

When asked if the group meeting changed their understanding of the code, no subjects said their understanding was confounded, 39.1% reported no change in understanding, and 60.9% reported an increase in understanding. Compared with the previous experiment (2.5%, 35% and 62.5%), these figures are very similar. This reinforces the view that the inspection meeting helps consolidate understanding of the product amongst group members. Meeting gains were reported to be slightly lower than those in Experiment 1. 17.4% reported no gains, 39.1% reported one or two gains, 32.6% reported gains of three to five defects and 10.9%

reported a gain of more than five defects. The respective figures for the first experiment were 7.5%, 40%, 47.5% and 5%. Meeting losses, on the other hand, were thought by subjects to be greater. 23.9% of respondents thought no losses had occurred, 23.9% thought one or two losses had occurred, 43.5% thought three to five defects had been lost and 8.7% thought more than five losses had occurred. The figures for Experiment 1 were 27.5%, 35%, 27.5% and 10% respectively.

In terms of their group's performance, 2.2% of respondents thought their group had found 41-60% of all the defects in the code, 34.8% thought 61-80% of the defects had been found and 63% thought 81-100% had been found. These figures are very similar to those from the first experiment (2.5%, 30% and 67.5%).

88% of tool users responded to the question concerning the overall usability of ASSIST. 18.2% thought it extremely usable, 50% thought it fairly usable, 18.2% thought it average, 13.6% thought it fairly unusable (no respondents classed it totally unusable). Compared with the first experiment, where the figures were 10%, 50%, 35% and 5% respectively, there are less subjects with a neutral opinion. The changes in opinion are evenly split between finding ASSIST extremely usable and finding it fairly unusable.

Questionnaire 4 Asked about their understanding of `graph.cc`, 4.3% of subjects understood 41-60% of the program, 57.4% understood 61-80% of the program and 38.3% understood 81-100% of it. Subjects seemed to think they understood the program more than the subjects in the previous experiment, where the corresponding figures were 20%, 37.5% and 40% (with 2.5% of subjects only understanding 21-40% of the program). Despite this, subjects in the last experiment slightly outperformed subjects in the current experiment. Concerning the time given to inspect the program, 23.4% of respondents thought insufficient time was given, 74.5% thought that just enough time was given, and only 2.1% thought that too much time was available. Compared with Experiment 1, where the corresponding figures were 35%, 62.5% and 2.5%, slightly more people thought the time given was sufficient to complete the inspection task. Comparing the complexity of the two programs, 53.2% of subjects thought `graph.cc` much more complex than `analyse.cc`, 34.1% thought it slightly more complex, 2.1% thought it to be of similar complexity and 10.6% perceived it slightly less complex. This is similar to the results for Experiment 1 (49%, 44% and 7% respectively, with no subjects finding `graph.cc` less complex), although a number of subjects in the second experiment found `graph.cc` less complex than their counterparts in the first experiment.

Concerning the group meeting, 6.4% of respondents indicated their understanding of the program was confounded, 34% reported no change in understanding and 59.6% indicated an

increase in understanding. Compared to Experiment 1, fewer people reported an increase in understanding (0%, 25% and 75%). Subjects reported more meeting gains than their counterparts in Experiment 1. A gain of zero was reported by 12.8% of subjects, 44.6% reported a gain of one or two defects, 29.8% reported a gain of three to five defects and 12.8% reported a gain of more than five defects. The corresponding figures for the previous experiment were 5%, 60%, 22.5% and 12.5%. On the other hand, meeting losses were perceived to be virtually identical. Zero losses were reported by 70.3% of respondents, 19.1% reported a loss of one or two, 8.5% reported a loss of three to five and 2.1% reported a loss of more than five, compared to 72.5%, 17.5%, 10%, and 0% from the previous experiment.

100% of tool users responded to the question concerning the usability of ASSIST for inspection. 16.6% thought it extremely usable 45.9% thought it fairly usable, 33.3% thought it average and 4.2% found it fairly unusable (no subjects thought it totally unusable). This compares favourably with the results of the previous experiment, where the corresponding figures were 4.8%, 33.3%, 23.8%, 33.3% and 4.8%. The advanced features in the second version of ASSIST appear to have increased its acceptance here.

Subjects were asked to rate their understanding of software inspection. No subjects rated their understanding as complete, 51.1% thought they understood it well and 48.9% believed they understood it reasonably well. Compared with Experiment 1, where the corresponding figures were 7.5%, 32.5% and 55% (with 5% of subjects slightly unsure), there is less of a spread of answers, but no real trend. Asked if their knowledge of C++ was adequate for the tasks set, 17% thought it inadequate and 83% thought it adequate, being almost identical to the first experiment, with 20% and 80% respectively.

When subjects were asked to compare their use of the checklist with each method, 38.3% thought they used it more with paper-based inspection, 46.8% used it the same with both methods and 14.9% used it more with ASSIST. Although more subjects made use of their checklist with ASSIST compared with Experiment 1 (where the corresponding figures are 37.5%, 52.5% and 10%), it is not a great increase. This is slightly disappointing, since one of the major features of the second version of ASSIST was the active checklist mechanism, designed to increase the use of the checklist. As with the first experiment, the lack of screen space probably forces users to discard the checklist window in preference to the code and specification.

In terms of preference of screen-based documents versus paper, 17% of subjects preferred screen-based documents, 42.6% expressed no preference and 40.4% preferred paper documents. Compared with the first experiment (with figures of 15%, 20% and 65%), more people have no preference, at the expense of paper documents. This could indicate that the second

version of ASSIST is more usable, with more people happy to use either type of document rather than being averse to screen-based.

Finally, subjects were asked to compare their performance using tool-based and paper-based inspection. With individual inspection, 14.9% of subjects thought they performed better with paper, 57.5% thought they performed equally well with both methods, and 27.6% felt they performed better using the tool. Like the preference for document type, more people are equally amenable to both methods compared to the previous experiment (where the corresponding figures were 39%, 39% and 22%). When considering the group meeting, 21.3% of subjects felt they performed better with paper, 63.8% felt they performed equally well with both methods, and 14.9% felt they performed better using ASSIST. There is a slight drop in preference for ASSIST compared to the previous experiment (19.5%, 61% and 19.5%), although not by much, and there is no clear trend.

The qualitative statements of preference were generally similar to those from the first experiment. Those who preferred paper-based inspection liked to be able to write their own notes directly on the code and found it easier to organise a number of sheets of paper on the desk than a number of windows on-screen. In general, they thought paper to be more natural (although one user who preferred ASSIST noted that reading code on-screen was more natural!). Some subjects indicated they performed more additional defect detection at a paper-based group meeting than when using the tool, and that sitting at a computer inhibited group discussion.

Subjects with a preference for ASSIST found the cross-referencing facilities to be useful when finding dead code and for tracing function and variable usage. It also helped find appropriate checklist items. The on-line C++ reference was thought to help save time, obviating the need to find and search a C++ textbook. Defect list manipulation was regarded as easier, especially during the group meeting where the scribe does not have to write out each defect again. The voting mechanism also had advocates.

Subjects with no preference pointed out advantages and disadvantages of both methods. Some felt that while the facilities of ASSIST were useful during individual inspection, the use of ASSIST hindered the group meeting, since group members were spaced further apart. This was accentuated by the layout of the computer laboratory: long rows of machines side-by-side. Users found it difficult to look at the screen and at their colleagues. A circular layout for each group would be more appropriate. Alternatively, a process consisting of a tool-based individual phase followed by a face-to-face meeting could be used.

7.1.4 Conclusions

The results from this experiment fail to show an increase in the performance of subjects using enhanced tool-based inspection compared with those performing paper-based inspection. Although statistically significant results were not found, responses to questionnaires indicated greater user acceptance of the enhanced version of the tool over the basic version. Compared with Experiment 1, more subjects perceived tool support to be more efficient than paper-based inspection, both for individual preparation and the group meeting. Document navigation was found to be easier, and the number of windows used by ASSIST was less of an issue, perhaps indicating that the cross-referencing mechanism was helping users. Some individual defects were found by more tool users than paper-based inspectors, showing the usefulness of the checklist linked to document features. Other defects were still found more easily on paper. More research has to be carried out to discover which particular defect types can be found more easily on paper, and hence explore mechanisms for supporting their detection on-line. Regarding the inspection meeting, less defect detection seemed to occur at tool-based meetings than paper-based meetings. Subjects appeared to use the tool simply as a mechanism for combining defect lists. This certainly needs to be addressed, either by training or by improving the facilities offered by the tool.

As with the Experiment 1, this experiment was limited by available resources. The use of student subjects and short pieces of code limit the extent to which these results can be generalised, although a realistic inspection process and inspection rate were used. Further work is required in replicating this experiment both in similar environments and in an industrial setting. For example, although the training tutorial was modified to help subjects find the best way of using the tool, the training itself was still brief, which undoubtedly had an adverse effect on their performance with the tool. Prospective subjects must preferably be experienced in inspection and the use of whichever tool is to be investigated. Some features of the tool may have had a positive effect, while other factors produced a negative effect. Isolating features and experimenting with them individually may provide more information. Only such repeated experimentation will provide a definitive answer to the question of the relative effectiveness of paper-based and tool-based inspection.

7.2 Automatic Defect List Collation

7.2.1 Introduction

Automatic defect list collation, introduced in Section 6.4, is designed to increase the efficiency of inspection by merging defect lists and removing duplicates without human intervention. Several factors can be set by the user to influence the results of the collation. These are the relative importance of the three facets used to compare defects (position, content and classification) and the threshold value above which items are considered to be duplicates. To find the range within which these factors are most usefully set, the defect lists produced by subjects in both comparison experiments were used as source material.

7.2.2 Method

The individual defect lists produced by subjects using ASSIST when inspecting the two experiment programs `analyse.cc` and `graph.cc` were used to perform this evaluation. 22 lists were available from the first experiment with `analyse.cc`, while 21 lists were available from the first experiment with `graph.cc`. The second experiment provided 23 and 25 lists respectively. The items in each individual list were tagged with a number indicating the defect (according to the defect lists in Appendix D), or removed if the defect was a false positive. Auto-collation was then applied to groups of three or four lists, as determined by the group allocation from the appropriate experiment, using a number of factor settings. The factor settings used were

- Content (importance of the contents when checking for duplicate items): 0.05 - 0.95, in steps of 0.05.
- Position (importance of the position when checking for duplicate items): 1 minus the current Content setting.
- Classification (importance of the classification when considering duplicates): always set to 0, since subjects were not asked to classify their defects.
- Threshold (value which the similarity score must reach for two items to be considered duplicates): 0.05 – 0.95, in steps of 0.05.

For each setting of the content factor, the position factor was set appropriately and auto-collation applied for each threshold value.

For each group of lists the optimal defect list which could be produced was calculated, consisting of the set of all defects found in all lists minus duplicates. The quality of the output

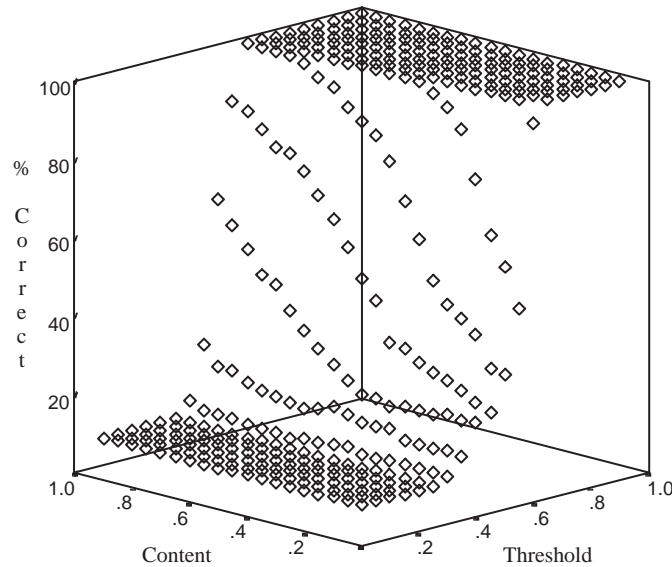


Figure 7.5: Average percentage of correct defects in collated lists for each value of content and threshold (Experiment 1 data).

from auto-collation can then be measured as the percentage of the optimal defect list retained (hereafter known as the percentage of correct defects). The worst case list for each group was calculated as the concatenation of all lists. Performance of duplicate removal can then be measured as the number of duplicates in each auto-collated list, expressed as a percentage of the total possible duplicates in the list. The data for both programs was grouped together, however the data for both experiments was treated separately. This was because the defect positions for the first experiment consist only of line numbers, while those in the second experiment consist of line numbers and character positions, due to the different browsers used to display the code.

7.2.3 Results

Figure 7.5 shows the performance of auto-collation in terms of the percentage of correct defects in the generated list, applied to defect lists from Experiment 1. Overall, the performance is very stable, with the graph having three distinctive areas. With low thresholds (≤ 0.25) most items are discarded, as expected, since items easily pass the similarity test. With high

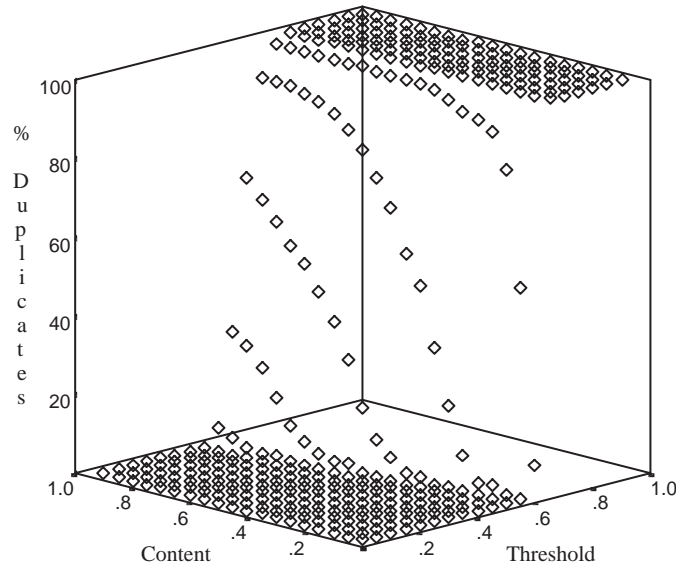


Figure 7.6: Average percentage of duplicates in collated lists for each value of content and threshold (Experiment 1 data).

threshold values (≥ 0.6), most items are kept, since the similarity test is harder to pass. In between these threshold values, there is steady increase in the percentage of correct defects, with the rate of this increase rising with larger content factors. Note that the lowest value for each setting of the content factor is always just above zero, since one defect must be put in each list to begin the auto-collation.

The average percentage of duplicates left in each list for various values of content and threshold is shown Figure 7.6. This graph shows a similar trend to Figure 7.5, although the threshold values corresponding to the three distinctive areas are higher than their counterparts, at around 0.45 and 0.65. The content factor setting has an effect here similar to that in Figure 7.5: as it increases so too does the rate of increase in the percentage of duplicates remaining.

To find the optimal settings for the content and threshold factors, the percentage of correct defects in the list must be compared with the percentage of duplicates remaining. The optimal settings occur when all the correct defects are present, while as many duplicates as possible are removed. Figures 7.7, 7.8 and 7.9 show the performance in terms of correct defects and

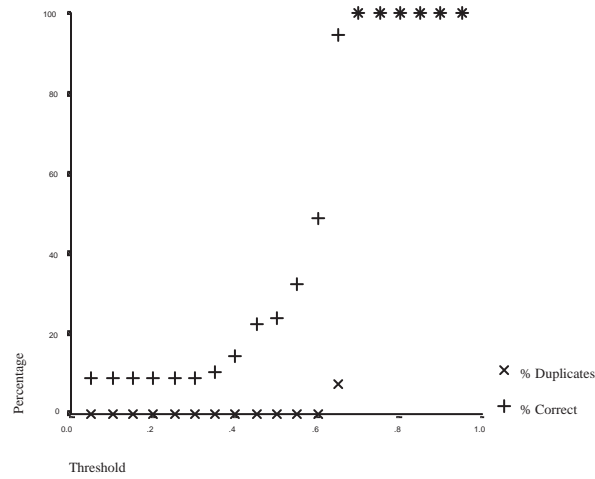


Figure 7.7: Average percentage of defects remaining and average percentage of duplicates in collated defect lists for a contents factor of 0.05 (Experiment 1 data).

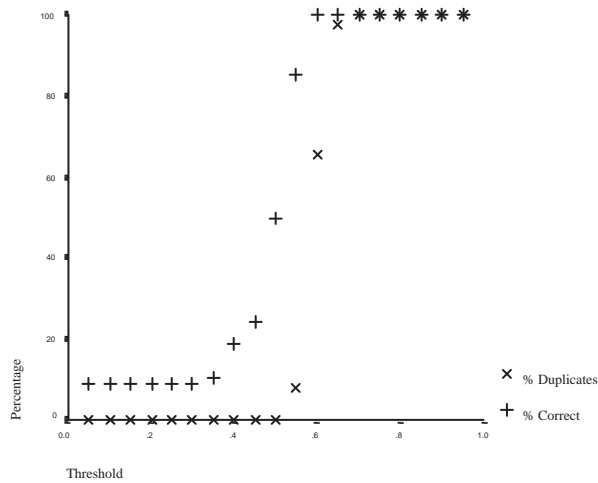


Figure 7.8: Average percentage of defects remaining and average percentage of duplicates in collated defect lists for a contents factor of 0.5 (Experiment 1 data).

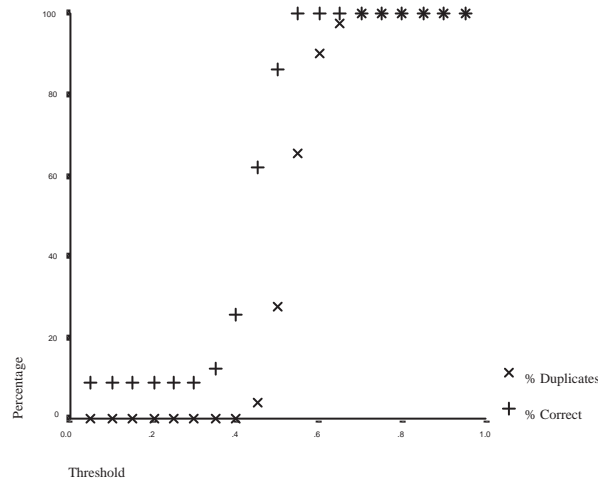


Figure 7.9: Average percentage of defects remaining and average percentage of duplicates in collated defect lists for a contents factor of 0.95 (Experiment 1 data).

duplicates for varying threshold factors for content factors of 0.05, 0.5 and 0.95, respectively. For a content factor of 0.05 (Figure 7.7), 100% of defects occur at a threshold of 0.7, but so also do 100% of duplicates. The threshold value of 0.65 is interesting, since less than 10% of duplicates occur, but only around 95% of correct defects occur. This presents an interesting dilemma: is it more efficient to remove 90% of duplicates, even if it means losing one or two real defects? When the content factor is set to 0.5 (Figure 7.8), the 100% region for defects occurs at a threshold of 0.6. At this threshold around 65% of duplicates occur. At a content factor of 0.95 (Figure 7.9), 100% of defects occur at a threshold of 0.55. At the same threshold only 65% of duplicates remain. Overall, setting the threshold to around 0.6 along with a contents factor of at least 0.5 appear to give the best results, removing about a third of duplicates, with no loss of defects. If the loss of one or two defects is acceptable then the contents factor can be reduced and the number of duplicates removed increased significantly.

The graphs of performance for the data from Experiment 2 are very similar to their Experiment 1 counterparts. Figure 7.10 shows the average percentage of correct defects in each list versus the content and threshold settings. Again, three distinct regions appear in the graph. The lower region, with threshold being less than 0.2, is where virtually all defects are discarded. The upper region, above a threshold of 0.6, is where all defects are kept. The middle region provides a steady increase in the number of defects retained, with the rate of increase rising as the value of the content factor rises. Figure 7.11 shows the average percentage of duplicates for the data from Experiment 2. Once again, the graph has a similar form, with the

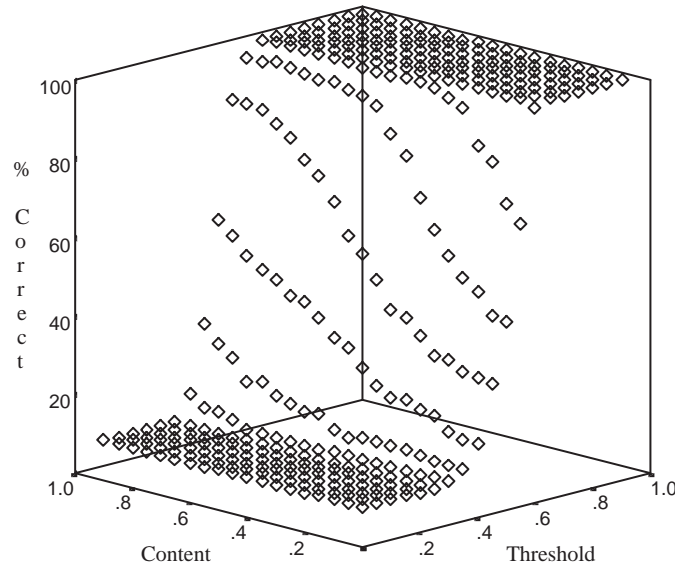


Figure 7.10: Average percentage of correct defects in collated lists for each value of content and threshold (Experiment 2 data).

boundaries occurring at threshold values of 0.35 and 0.6.

Figures 7.12, 7.13 and 7.14 show the average percentage of defects remaining and the average percentage of duplicates for contents factors of 0.05, 0.5 and 0.95 respectively. These graphs follow their counterparts for the Experiment 1 data very closely. Considering Figure 7.12, if the criteria applied is that 100% of correct defects must occur then these settings are obviously not useful, since 100% of duplicates occur for any threshold setting where 100% of defects occur. If, however, the condition is relaxed slightly, with around 99% correct being acceptable, then the threshold value of 0.65 allows rejection of almost 60% of duplicates. Since the 99% value is an average, in most cases 100% of all defects are being found with just a single instance of perhaps one defect being lost. In comparison with the same values for the Experiment 1 data, slightly fewer defects are lost, although many more duplicates remain. In Figure 7.13, when the 100% criterion for correct defects is fulfilled, less than 5% of duplicates have been removed. Relaxing the criterion slightly gives just over 20% of duplicates removed at a threshold of 0.6. This is a slightly worse result than for the Experiment 1 data, both for correct defects and duplicates remaining. Figure 7.14 follows the pattern observed so

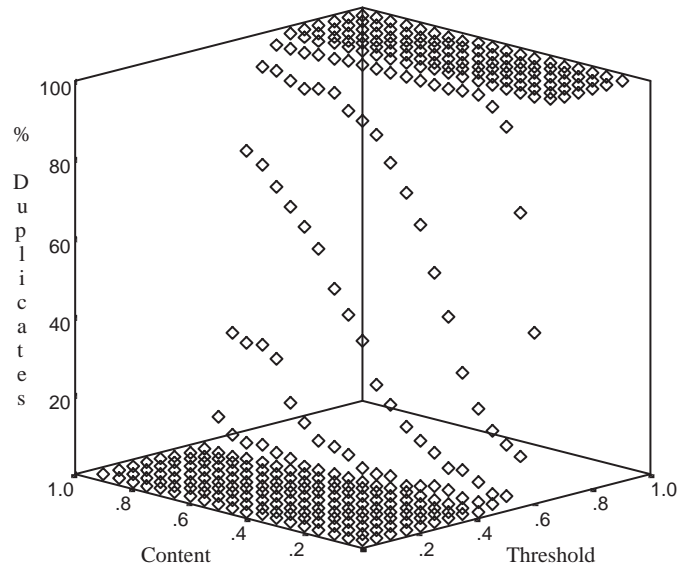


Figure 7.11: Average percentage of duplicates in collated lists for each value of content and threshold (Experiment 2 data).

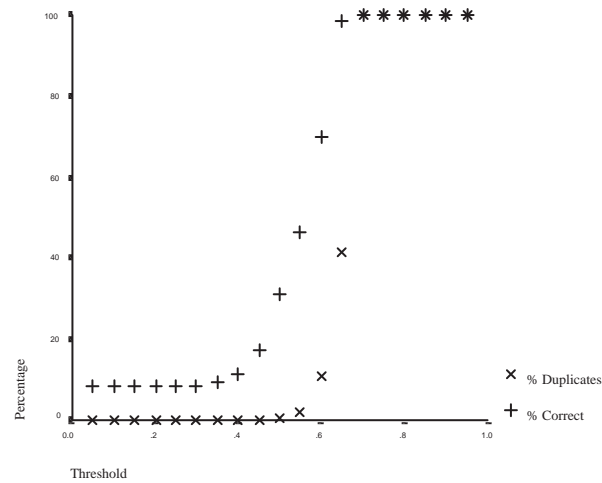


Figure 7.12: Average percentage of defects remaining and average percentage of duplicates in collated defect lists for a contents factor of 0.05 (Experiment 2 data).

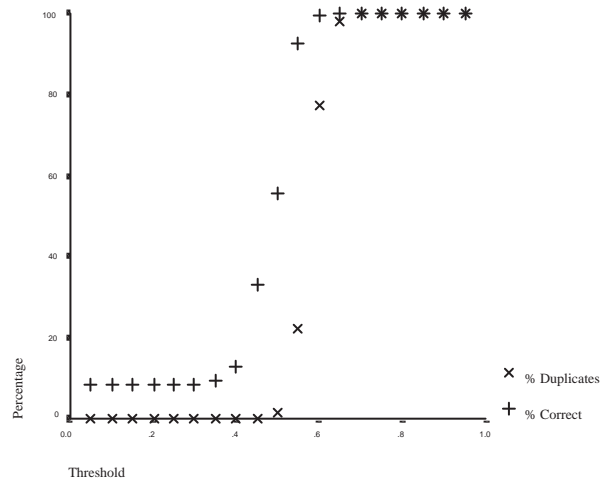


Figure 7.13: Average percentage of defects remaining and average percentage of duplicates in collated defect lists for a contents factor of 0.5 (Experiment 2 data).

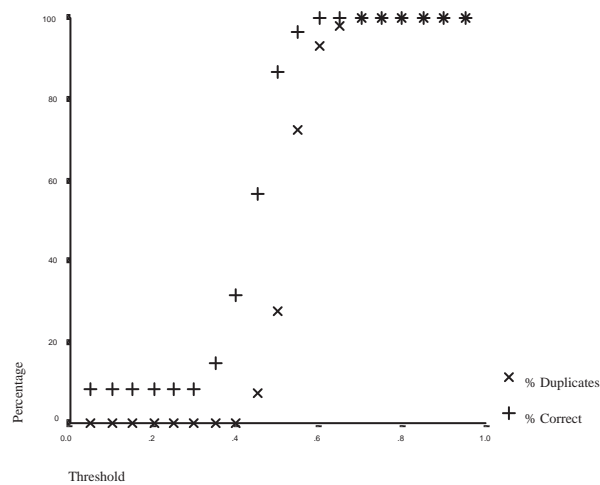


Figure 7.14: Average percentage of defects remaining and average percentage of duplicates in collated defect lists for a contents factor of 0.95 (Experiment 2 data).

far. At a threshold value of 0.55, it provides slightly worse results than the Experiment 1 data in terms of correct defects and duplicates remaining. Again, for acceptable duplicate removal performance, a small loss of real defects has to occur.

One reason why the data for Experiment 2 provides slightly worse results than that from Experiment 1 is the difference in the way in which defects positions are represented. In Experiment 1 only line numbers are used, while both line number and character positions are used in Experiment 2. This more accurate positioning system exaggerates small differences in defect positions. For example, when using line numbers only, two defects on the same line have identical positions. When using both line number and character position, two defects on the same line may have positions such as 32.0 and 32.4. Some subjects may consider the defect to occur at the start of the line (even if there is blank space at the start of the line), while others may mark its position exactly. Instructing subjects on a uniform method of deciding defect positions would help reduce this variability. The effect of the positioning strategy should be reduced as the contents factor is increased, since the value of the position factors decreases at the same time.

One factor which may generally have had an adverse effect on the results of the auto-collation experiments was the variability in spelling among subjects. Misspelling of long words such as 'initialisation' was commonplace. Such misspellings reduce the effectiveness of word matching between items, stop list matching and stemming. The result is a reduction in the probability of two items being declared duplicates, thereby reducing the effectiveness of auto-collation. Although misspellings could have been fixed before the experiments were carried out, this would not have reflected real usage. The obvious solution to this problem is provision of a spelling checker within the tool. Subjects also tended to use varying terminology. In an industrial setting terminology would probably be more consistent, which would also improve performance.

Another factor which may have an effect is the type of defect. Some defects can be reported more accurately than others, and are less likely to vary between subjects. For example, the use of the wrong operator can be easily described in terms of position and contents. Missing functionality, on the other hand, is subject to more variability in description and perceived position.

7.2.4 Conclusions

With the strict criterion of absolutely no loss of defects, auto-collation can reliably remove 10%-35% of duplicates. If the criterion is relaxed slightly, with one or two losses being acceptable, the rate of removal can be as high as 60-90%. The question of whether the defect

loss is acceptable to reach such removal rates is a difficult one to answer, and depends on the context in which the system is being used and the relative costs and benefits. The performance of the auto-collation system would almost certainly be improved by correcting misspellings in defects and more consistent use of terminology.

Chapter 8

Conclusions

8.1 Summary

Software inspection is widely regarded as an effective defect detection technique. Computer support for software inspection has been suggested as a means of further increasing its efficiency and effectiveness. This has resulted in the development of a number of prototype tools. Unfortunately, these systems suffer from a number of shortcomings.

This thesis has investigated the requirements of software inspection support systems. Existing support tools were critically reviewed and the major weaknesses were identified. Two such weaknesses were found. The first concerns process and document independent support. Virtually all existing tools support only a single inspection process, while there are a number of variations which may require support. Existing tools are also limited to inspection of a single document type. Support for multiple documents types is essential if computer supported inspection is to be integrated with existing development environments. The second concerns enhancing performance during inspection. Existing tools use a simple static representation of the document which the inspector may annotate. In fact, there is scope for providing facilities to enhance the inspection process.

To tackle the first concern, a language to allow modelling of inspection processes was defined. The language, IPDL, was derived by studying the eight processes reviewed in Chapter 2. IPDL was implemented in a prototype inspection support tool known as ASSIST. The tool can execute any process written in IPDL, ensuring that the process is followed precisely and that the correct documents are available at each phase. The first version of the tool also provided the all basic facilities required to support inspection, including a number of browsers, an annotation mechanism, and a voting system to help resolve issues at the inspection meeting.

The tool was also designed to allow new browsers to be added as required, allowing support for multiple document types.

Having implemented a tool providing the basic set of facilities required to support an inspection, an evaluation was performed to compare the effectiveness of paper-based inspection with tool-based. Questionnaires from the experiment provided useful feedback on the usability of ASSIST. The experiment also suggested features which ASSIST could provide to ease the task of the inspector. A number of features were then chosen and implemented. These included automatic cross-referencing and active checklists, which were used to create a C++ inspection environment. A means of automatically collating multiple defect lists into a single master list, with non-identical duplicates being removed, was also designed.

The new version of ASSIST was then used in a second evaluation, comparing the enhanced version of ASSIST with paper-based inspection. This was identical in execution to the first comparison experiment. An experiment was also performed to test the effectiveness of the auto-collation mechanism.

8.2 Contributions and Results

Several major contributions to computer supported software inspection have been made. The first concerns IPDL, a language which can be used to describe many inspection processes. Process descriptions can be used to unambiguously communicate that process, or as input to an inspection support tool to provide support for that process. The language is of low complexity, enabling processes to be quickly and easily written. Process descriptions are short, and the English-like syntax of IPDL provides very readable processes.

This thesis has also introduced ASSIST, a prototype inspection support tool used as a vehicle to implement this research. Table 8.1 compares ASSIST with the other on-line inspection tools described in Chapter 3. As can be seen, the only feature not implemented by ASSIST concerns analysis of documents to automatically find defects. In other regards, ASSIST provides the most comprehensive set of features of any inspection support system. In addition, ASSIST implements IPDL, allowing support for multiple inspection processes. The architecture of ASSIST also allows the support of many document types.

The first reported comparisons of paper-based and tool-based inspection are presented in this thesis. The first compared a basic version of ASSIST with paper-based inspection, and could detect no significant difference in the performance of subjects using each method. Hence, it was concluded that the concept of tool-based inspection was not fundamentally

	ASSIST	ICICLE	CSI	InspeQ	Scrutiny	TAMMi	DCI	CSRS	CAIS	AISA	NI	InspectA	hyperCode	WiP
Linked Annotations	•	•	•		•	•		•	•	•	•		•	•
Defect Classification	•	•	•		•	•		•	•	•		•		•
Cross-referencing	•	•												
Automated Analysis		•												
Checklists	•		•	•		•			•			•		•
Supporting Material	•	•		•			•					•		•
Distributed Meetings	•		•		•		•							
Decision Support	•				•			•	•	•	•			
Data Collection	•	•	•		•			•	•					•

Table 8.1: Comparison of ASSIST with other on-line inspection tools.

flawed, and could be explored further. Feedback from the experiment also provided information on ways in which the tool could be improved, and tasks which subjects found difficult. The second experiment compared an enhanced version of ASSIST with paper-based inspection. Again, no significant difference was found. Although this result was disappointing, feedback from subjects indicated that the usability of ASSIST had been increased.

8.3 Further Work

The research presented in this thesis can be extended in several ways. To begin with, IPDL is currently implemented as a text-only language. Although ASSIST implements a helper which will produce a skeleton process definition, there is scope for providing an easier means of producing definitions. For example, IPDL could be partially or fully implemented as a graphical language. This is most beneficial when considering the order and type of process phases. With an appropriate editor, symbols representing each phase type could quickly and easily be assembled in the correct order. Documents and participants could also be handled graphically as named objects. The checklist definition language would also benefit from graphical editing, easing the creation and modification of checklists.

Evaluation of the research undertaken has been a major theme in this thesis. In this respect, there is still more research which should be performed. The experiments comparing paper-based and tool-based inspection should be repeated to gain confidence in their results. To this end, a package containing all materials required to replicate these experiments has been created and is freely available to interested researchers. This could also be used to perform similar experiments. Similarly, ASSIST itself is also freely available. Other researchers could make use of ASSIST as a platform to investigate computer supported software inspection, extending it as required.

Although the experiments could be replicated without change, there are a number of respects in which they could be improved. To begin with, the use of student subjects limits the extent to which the results may be generalised. The use of industrial participants would greatly enhance the external validity. Using industrial code samples would also increase external validity. Some procedural aspects of the experiments could also be improved. For example, the enhanced version of ASSIST may not have provided an increase in inspection efficiency due to the abbreviated training given to subjects. Performing the experiments with experienced users of the tool would increase validity. Features of ASSIST could be evaluated individually, since it may be that a gain from one feature is being offset by a loss from another. For example, active checklists are designed to increase checklist usage. An experiment comparing

checklist usage may provide more insight into their value.

IPDL is based on existing paper-based processes. As previously stated, it may be the case that the introduction of tool support alters the manner in which these processes are performed. This effect may vary depending on the particular facilities offered by an individual tool, along with the familiarity of users with that tool. A simple tool will encourage less change in the method of inspection than a feature-rich tool. An expert user will have developed an individual manner of using the tool, and will be capable of using more features. Hence, these type of effects should be investigated. There is also an opportunity to investigate processes which are only feasible with the aid of tool support. Asynchronous inspection is such a process which has already been investigated. Such investigation may lead to refinement and extension of IPDL to model such tool-dependent processes. IPDL processes are also static, and ASSIST does not allow their modification once underway. Another avenue of research could consider the area of dynamic process change, allowing the inspection process to be modified part-way through execution.

The auto-collation experiments could also be repeated in several ways. To begin with, the use of other defect lists would help provide confidence in its performance. The same defect lists could also be used, this time with spelling errors corrected, to test the hypothesis that such spelling errors had an adverse effect on the auto-collation mechanism. Finally, the defects in the lists used were not classified. Lists of classified defects could be used to investigate the effect of classification.

Appendix A describes two related areas of research considered for this thesis but not pursued. One concerns methods of enhancing the defect detection capability of participants, with specific reference to object-oriented systems. This research is necessary since features of the object-oriented paradigm can hinder the inspection task. The second concerns the collection and analysis of inspection data. Several uses of this data are explored, including general inspection process improvement, checklist formation and improvement and estimating defects remaining after inspection.

8.4 Concluding Remarks

From the research performed, it is clear that computer supported software inspection is a valuable line of research. When tool-based inspection was compared with paper-based, no difference in inspector performance could be detected. This provides a baseline for exploring more advanced tool support, with enhanced facilities to help inspectors find defects. When coupled with the less quantifiable advantages of tool support – the ability to easily collect data,

process rigour, the use of electronic versions of documents, support for distributed inspection, etc. – the result indicates that a move to computer supported inspection could, in general, be beneficial.

The research methodology employed proved to be ideal. To begin with, existing tools were investigated and their weaknesses identified. These weaknesses were used to help create the specification for ASSIST. Existing inspection processes were used to help design IPDL. The first version of ASSIST was then implemented and evaluated. The results of this evaluation were used to design improvements for the second version. A second evaluation was then performed. This cycle of design/evaluation could have been repeated as required. There is one caveat: feedback from subjects during the second experiment was less useful than that from the first. The limited experience of subjects may have been a factor here. Hence, the design/evaluation cycle would be more effective if continued with software professionals.

There is much scope for further research in computer supported software inspection. The full potential of inspection may not yet have been reached, and innovative research in this area may be the means of achieving even greater results than the many positive experience reports already found in the literature. As one of the most successful defect finding techniques in use, it deserves much more research to explore its full potential.

Bibliography

- [1] T. K. Abdel-Hamid and S. E. Madnick. Lessons learned from modelling the dynamics of software development. *Communications of the ACM*, 32(12):1426–1438, December 1989.
- [2] J. E. Arnold and S. S. Popovich. Integrating, customising and extending environments with a message-based architecture. Technical Report CUCS-008-95, Department of Computer Science, Columbia University, New York, 1995.
- [3] J. T. Baldwin. An abbreviated C++ code inspection checklist. Available on the WWW, URL: <http://www.ics.hawaii.edu/johnson/FTR/Bib/Baldwin92.html>, 1992.
- [4] S. Bandinelli, A. Fuggetta, and C. Ghezzi. Software process model evolution in SPADE. *IEEE Transactions on Software Engineering*, 19(12):1128–1144, December 1993.
- [5] S. Bandinelli, A. Fuggetta, C. Ghezzi, and L. Lavazza. SPADE: An environment for software process analysis, design and enactment. In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors, *Software Process Modelling and Technology*, chapter 9, pages 223–247. Research Studies Press, Ltd., 1994.
- [6] H. J. Barnard and R. B. Collicott. COMPAS: A development process support system. *AT&T Technical Journal*, 62(2):52–64, March/April 1990.
- [7] J. Barnard and A. Price. Managing code inspection information. *IEEE Software*, 11(2):56–69, March 1994.
- [8] V. R. Basili and R. W. Selby. Comparing the effectiveness of software testing strategies. *IEEE Transactions on Software Engineering*, 13(12):1278–1296, December 1987.
- [9] Bell Communications Research. *ICICLE User's Guide*, January 1993.

- [10] C. A. Boneau. The effects of violations of assumptions underlying the t test. *Psychological Bulletin*, 57(1):49–64, 1960.
- [11] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin Cummings, second edition, 1994.
- [12] L. R. Brothers, V. Sembugamoorthy, and A. E. Irgon. Knowledge-based code inspection with ICICLE. In *Innovative Applications of Artificial Intelligence 4: Proceedings of IAAI-92*, 1992.
- [13] L. R. Brothers, V. Sembugamoorthy, and M. Muller. ICICLE: Groupware for code inspections. In *Proceedings of the 1990 ACM Conference on Computer Supported Cooperative Work*, pages 169–181, October 1990.
- [14] N. Brown. Industrial-strength management strategies. *IEEE Software*, 13(4):94–103, July 1996.
- [15] Bull HN Information Systems, Inc., U.S. Applied Research Laboratory. *Scrutiny User's Guide*, May 1994.
- [16] Bull, S.A. *Inspection Process Assistant: User Guide V3.0*, September 1997.
- [17] A. Burr and M. Owen. *Statistical Methods for Software Quality*. International Thomson Computer Press, 1996.
- [18] T. Cai, P. A. Gloor, and S. Nog. DartFlow: A workflow management system on the Web using transportable agents. Technical Report PCS-TR96-283, Dartmouth College, 1996.
- [19] J. K. Chaar, M. J. Halliday, I. S. Bhandari, and R. Chillarege. In-process evaluation for software inspection and test. *IEEE Transactions on Software Engineering*, 19(11):1055–1070, November 1993.
- [20] Y. Chernak. A statistical approach to the inspection checklist formal synthesis and improvement. *IEEE Transactions on Software Engineering*, 22(12):866–874, December 1996.
- [21] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M-Y. Wong. Orthogonal defect classification: A concept for in-process measurements. *IEEE Transactions on Software Engineering*, 18(11):943–956, November 1992.

- [22] A. M. Christie. A graphical process definition language and its application to a maintenance project. *Information and Software Technology*, 35(6/7):364–374, June/July 1993.
- [23] R. Conradi, M. Hagaseth, J-O. Larsen, M. N. Nguyễn, B. P. Munch, P. H. Westby, W. Zhu, M. L. Jaccheri, and C. Liu. EPOS: Object-oriented cooperative process modelling. In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors, *Software Process Modelling and Technology*, chapter 3, pages 33–70. Research Studies Press, Ltd., 1994.
- [24] R. T. Crocker and A. von Mayrhauser. Maintenance support needs for object-oriented software. In *Proceedings of COMPSAC '93*, pages 63–69, 1993.
- [25] C. B. Darling. Embrace change with workflow tools. *Datamation*, 42(16):102–111, October 1996.
- [26] A. Davis. cppp - a C++ parser. Available from the Brown Computer Science Software Catalog, URL: <http://www.cs.brown.edu/software/catalog.html>.
- [27] D. B. Davis. Software that makes your work flow. *Datamation*, 38:75–78, 15th April 1991.
- [28] H.M. Deitel and P.J. Deitel. *C: How to Program*. Prentice-Hall International, second edition, 1994.
- [29] A. Dillon. Reading from paper versus screens: a critical review of the empirical literature. *Ergonomics*, 35(10):1297–1326, October 1992.
- [30] B. S. Doherty and S. Sahibuddin. Software quality through distributed code inspection. In C. Tasso, R. A. Adeyi, and M. Pighin, editors, *Software Quality Engineering*, pages 159–168. Computational Mechanics Publications, 1997.
- [31] E. P. Doolan. Experience with Fagan's inspection method. *Software-Practice and Experience*, 22(2):173–182, February 1992.
- [32] R. G. Ebenau. Predictive quality control with software inspections. *CrossTalk*, 7(6):9–16, June 1994.
- [33] R. G. Ebenau and S. H. Strauss. *Software Inspection Process*. McGraw-Hill, 1994.
- [34] A. L. Edwards. *Statistical Methods*. Holt, Rinehart and Winston, Inc, second edition, 1967.

- [35] S. G. Eick, C. R. Loader, M. D. Long, L. G. Votta, and S. A. Vander Weil. Estimating software fault content before coding. In *Proceedings of the Fourteenth International Conference on Software Engineering*, pages 59–65, May 1992.
- [36] S. G. Eick, J. L. Steffen, and E. E. Sumner Jr. Seesoft - a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, November 1992.
- [37] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.
- [38] M. E. Fagan. Advances in software inspection. *IEEE Transactions on Software Engineering*, 12(7):744–751, July 1986.
- [39] International Institute for Applied Systems Analysis. Library search stop list. URL: <http://www.iiasa.ac.at/docs/R.Library/libsrchs.html>.
- [40] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: from process modelling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, April 1995.
- [41] T. Gilb and D. Graham. *Software Inspection*. Addison-Wesley, 1993.
- [42] J. W. Gintell. A brief history of Scrutiny, February 1997. Personal communication.
- [43] J. W. Gintell, J. Arnold, M. Houde, J. Kruszelnicki, R. McKenney, and G. Memmi. Scrutiny: A collaborative inspection and review system. In *Proceedings of the Fourth European Software Engineering Conference*, September 1993.
- [44] J. W. Gintell, M. B. Houde, and R. F. McKenney. Lessons learned by building and using Scrutiny, a collaborative software inspection system. In *Proceedings of the Seventh International Workshop on Computer Aided Software Engineering*, July 1995.
- [45] R. Gribben. Software snags hold back air-traffic control launch. *Daily Telegraph*, 9th April 1996.
- [46] L. Harjumaa and I. Tervonen. A WWW-based tool for software inspection. In *Proceedings of HICSS-98*, volume III, pages 379–388, 1998.
- [47] D. Harman. Automatic indexing. In R. Fidel, T. B. Hahn, E. M. Rasmussen, and P. J. Smith, editors, *Challenges in Indexing Electronic Text and Images*, chapter 13, pages 247–264. Learned Information, Inc., 1994.

- [48] D. Heimbigner. The ProcessWall: a process state server approach to process programming. *ACM SIGSOFT Software Engineering Notes*, 17(5):159–168, December 1992.
- [49] K. E. Huff and V. R. Lesser. A plan-based intelligent assistant that supports the software development process. *ACM SIGSOFT Software Engineering Notes*, 13(5):97–106, November 1988.
- [50] W. S. Humphrey. *Managing the Software Process*, chapter 10, pages 171–190. Addison-Wesley, 1989.
- [51] W. S. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley, 1995.
- [52] J. B. Iniesta. A tool and a set of metrics to support technical reviews. In *2nd International Conference on Software Quality Management*, volume 1, pages 579–594, 1994.
- [53] ISO. *Information Processing Systems - Open Systems Interconnection - LOTOS, A Formal Description Technique based on the Temporal Ordering of Observational Behaviour*. International Organisation for Standardisation, ISO 8807, August 1988.
- [54] M. L. Jaccheri and R. Conradi. Techniques for process model evolution in EPOS. *IEEE Transactions on Software Engineering*, 19(12):1145–1156, December 1993.
- [55] P. M. Johnson. An instrumented approach to improving software quality through formal technical review. In *Proceedings of the 16th International Conference on Software Engineering*, May 1994.
- [56] P. M. Johnson. Supporting technology transfer of formal technical review through a computer supported collaborative review system. In *Proceedings of the 4th International Conference on Software Quality*, October 1994.
- [57] P. M. Johnson and D. Tjahjono. CSRS users guide. Technical Report ICS-TR-93-16, Collaborative Software Development Laboratory, Department of Information and Computer Sciences, University of Hawaii, 1993.
- [58] C. Jones. Gaps in the object-oriented paradigm. *IEEE Computer*, 27(6):90–91, June 1994.
- [59] C. L. Jones. A process-integrated approach to defect prevention. *IBM Systems Journal*, 24(2):150–167, 1985.

- [60] G. E. Kaiser, N. S. Barghouti, and M. H. Sokolsky. Preliminary experience with process modelling in the Marvel software development environment kernel. In *Proceedings of the 23rd Hawaii Conference on System Sciences*, pages 131–140, January 1990.
- [61] G. E. Kaiser, P. H. Feller, and S. S. Popovich. Intelligent assistance for software development and maintenance. *IEEE Software*, 5(3):40–49, May 1988.
- [62] E. Kamsties and C. M. Lott. An empirical evaluation of three defect-detection techniques. Technical Report ISERN-95-02, International Software Engineering Research Network, May 1995.
- [63] S. Kaplan, W. J. Tolone, D. P. Bogia, and C. Bignoli. Flexible, active support for collaborative work with ConversationBuilder. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, 1992.
- [64] G. Kappel, B. Pröll, S. Rausch-Scott, and W. Retschitzegger. TriGS_{flow} active object-oriented workflow management. In *Proceedings of the 28th Hawaii International Conference on System Sciences*, pages 772–736, January 1995.
- [65] J. C. Knight and E. A. Meyers. Phased inspections and their implementation. *Software Engineering Notes*, 16(3):29–35, July 1991.
- [66] J. C. Knight and E. A. Meyers. An improved inspection technique. *Communications of the ACM*, 36(11):51–61, November 1993.
- [67] M. M. Lehman. Software engineering, the software process and their support. *Software Engineering Journal*, 6(5):243–258, September 1991.
- [68] M. Lutz. *Programming Python*. O'Reilly & Associates, first edition, 1996.
- [69] F. Macdonald. ASSIST V1.1 User Manual. Technical Report EFoCS-22-96, Department of Computer Science, University of Strathclyde, February 1997.
- [70] F. Macdonald. ASSIST V2.1 User Manual. Technical Report EFoCS-28-98, Department of Computer Science, University of Strathclyde, March 1998.
- [71] F. Macdonald and J. Miller. Modelling software inspection methods for the application of tool support. Technical Report EFoCS-16-95 [RR/95/196], Department of Computer Science, University of Strathclyde, December 1995.

- [72] F. Macdonald and J. Miller. ASSISTing with software inspection. In *Proceedings of the 1st International Software Quality Week/Europe*. Software Research Institute, November 1997.
- [73] F. Macdonald and J. Miller. Automated generic support for software inspection. In *Proceedings of the 10th International Software Quality Week*. Software Research Institute, May 1997.
- [74] F. Macdonald and J. Miller. A software inspection process definition language and prototype support tool. *Software Testing, Verification and Reliability*, 7(2):99–128, June 1997.
- [75] F. Macdonald and J. Miller. ASSIST - a tool to support software inspection. Submitted to the *Journal of Information and Software Technology*, 1998.
- [76] F. Macdonald and J. Miller. A comparison of computer support systems for software inspection. Submitted to *Automated Software Engineering: An International Journal*, 1998.
- [77] F. Macdonald and J. Miller. A comparison of tool-based and paper-based software inspection. *Empirical Software Engineering: An International Journal*, 3(3), Autumn 1998.
- [78] F. Macdonald, J. Miller, A. Brooks, M. Roper, and M. Wood. A review of tool support for software inspection. In *Proceedings of the Seventh International Workshop on Computer Aided Software Engineering*, pages 340–349, July 1995.
- [79] F. Macdonald, J. Miller, A. Brooks, M. Roper, and M. Wood. Applying inspection to object-oriented code. *Software Testing, Verification and Reliability*, 6(2):61–82, June 1996.
- [80] F. Macdonald, J. Miller, A. Brooks, M. Roper, and M. Wood. Automating the software inspection process. *Automated Software Engineering: An International Journal*, 3(3/4):193–218, August 1996.
- [81] B. Marick. A question catalog for code inspections. Available via anonymous FTP from cs.uiuc.edu as /pub/testing/inspect.ps, 1992.
- [82] J. Martin and W.-T. Tsai. N-Fold inspection: A requirements analysis technique. *Communications of the ACM*, 33(2):225–232, February 1990.

- [83] V. Mashayekhi. *Distribution and Asynchrony in Software Engineering*. PhD thesis, University of Minnesota, March 1995.
- [84] V. Mashayekhi, J. M. Drake, W.-T. Tsai, and J. Reidl. Distributed, collaborative software inspection. *IEEE Software*, 10(5):66–75, September 1993.
- [85] V. Mashayekhi, C. Feulner, and J. Reidl. CAIS: Collaborative Asynchronous Inspection of Software. In *Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, December 1994.
- [86] I. R. McChesney. Towards a classification scheme for software process modelling approaches. *Information and Software Technology*, 37(7):363–374, July 1995.
- [87] E. A. Meyers and J. C. Knight. An improved software inspection technique and an empirical evaluation of its effectiveness. Technical Report TR-92-15, Department of Computer Science, University of Virginia, May 1992.
- [88] J. Miller and F. Macdonald. ASSISTing management decisions in software inspection processes. In *Proceedings the 13th IEEE Conference on Automated Software Engineering*, October 1998.
- [89] J. Miller and F. Macdonald. An incremental approach to tool development and evaluation. Submitted to the *Journal of Systems and Software*, 1998.
- [90] J. Miller, M. Roper, and M. Wood. Further experiences with scenarios and checklists. *Journal of Empirical Software Engineering*, 3(1):37–64, 1998.
- [91] J. A. Miller, D. Palaniswami, A. P. Sheth, K. J. Kochut, and H. Singh. WebWork: METEOR₂'s Web-based workflow management system. *Journal of Intelligent Information Systems*, 10(2):1–30, 1998.
- [92] J. A. Miller, A. P. Sheth, K. J. Kochut, and D. Palaniswami. The future of Web-based workflows. In *Proceedings of the International Workshop on Reserahc Directions in Process Technology*, July 1997.
- [93] P. Murphy and J. Miller. A process for asynchronous software inspection. In *Proceedings of The 8th International Workshop on Software Technology and Engineering Practice*, pages 96–104, July 1997.
- [94] D. L. Parnas, J. Madey, and M. Iglewski. Precise documentation of well-structured programs. *IEEE Transactions on Software Engineering*, 20(12):948–976, December 1994.

- [95] D. L. Parnas and D. M. Weiss. Active design reviews: Principles and practices. In *Proceedings of the Eighth International Conference on Software Engineering*, pages 132–136, August 1985.
- [96] W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the behaviour of object-oriented systems. In *Proceedings of the 8th International Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 326–337, 1993.
- [97] J. M. Perpich, D. E. Perry, A. A. Porter, L. G. Votta, and M. W. Wade. Anywhere, anytime code inspections: Using the web to remove bottlenecks in large-scale software development. In *Proceedings of the 19th International Conference on Software Engineering*, pages 14–21, 1997.
- [98] C. Ponder and W. Bush. Polymorphism considered harmful. *ACM SIGSOFT Software Engineering Notes*, 19(2):35–37, April 1994.
- [99] A. A. Porter, L. G. Votta, and V. R. Basili. Comparing detection methods for software requirements inspections: A replicated experiment. *IEEE Transactions on Software Engineering*, 21(6):563–575, June 1995.
- [100] M. F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.
- [101] M. Putaala and I. Tervonen. Inspecting Postscript documents in an object-oriented environment. In *5th European Conference on Software Quality*, 1997.
- [102] J. Reidl, V. Mashayekhi, J. Schnepf, M. Claypool, and D. Frankowski. Suitesound - a system for distributed collaborative multimedia. *IEEE Transactions on Knowledge and Data Engineering*, 5(4):600–610, 1993.
- [103] G. W. Russell. Experience with inspection in ultralarge-scale developments. *IEEE Software*, 8(1):25–31, January 1991.
- [104] V. Sembugamoorthy and L. R. Brothers. ICICLE: Intelligent Code Inspection in a C Language Environment. In *Proceedings of the 14th Annual Computer Software and Applications Conference*, pages 146–154, October 1990.
- [105] M. E. Shaw. *Group Dynamics: The Psychology of Small Group Behaviour*. McGraw-Hill, 1971.
- [106] E. Soloway, J. Pinto, S. Letovsky, D. Littman, and R. Lampert. Designing documentation to compensate for delocalized plans. *Communications of the ACM*, 31(11):1259–1267, November 1988.

- [107] P. Sparaco. Board faults Ariane-5 software. *Aviation Week and Space Technology*, 145(5):33–34, 1996.
- [108] M. Stein, J. Riedl, S. J. Harner, and V. Mashayekhi. A case study of distributed, asynchronous software inspection. In *Proceedings of the 19th International Conference on Software Engineering*, pages 107–117, 1997.
- [109] S. M. Sutton, Jr., D. Heimbigner, and L. J. Osterweil. APPL/A: A language for software process programming. *ACM Transactions on Software Engineering and Methodology*, 4(3):221–286, July 1995.
- [110] M. Suzuki and T. Katayama. Metaoperations in the process model HFSP for the dynamics and flexibility of software processes. In *Proceedings of the First International Conference on the Software Process*, pages 202–217, 1991.
- [111] K. D. Swenson. A visual language to describe collaborative work. In *Proceedings of the 1993 IEEE Symposium on Visual Languages*, August 1993.
- [112] K. D. Swenson, R. J. Maxwell, T. Matsumoto, B. Saghari, and K. Irwin. A business process environment supporting collaborative planning. *Journal of Collaborative Computing*, 1(1):119–153, Spring 1994.
- [113] I. Tervonen. Consistent support for software designers and inspectors. *Software Quality Journal*, 5:221–229, 1996.
- [114] I. Tervonen. Support for quality-based design and inspection. *IEEE Software*, 13(1):44–54, January 1996.
- [115] C. Thompson and J. Riedl. Collaborative asynchronous inspection of software using Lotus Notes. Technical Report TR 95-047, Computer Science Department, University of Minnesota, 1995.
- [116] D. Tjahjono. Comparing the cost effectiveness of group synchronous review method and individual asynchronous review method using CSRS: Results of pilot study. Technical Report ICS-TR-95-07, University of Hawaii, January 1995.
- [117] D. Tjahjono. *Exploring the Effectiveness of Formal Technical Review Factors with CSRS, a Collaborative Software Review System*. PhD thesis, Department of Information and Computer Sciences, University of Hawaii, June 1996.

- [118] L. G. Votta. Does every inspection need a meeting? In *Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 107–114, December 1993.
- [119] E. F. Weller. Lessons from three years of inspection data. *IEEE Software*, 10(5):38–45, September 1993.
- [120] S. A. Vander Wiel and L. G. Votta. Assessing software designs using capture-recapture methods. *IEEE Transactions on Software Engineering*, 19(11):1045–1054, November 1993.
- [121] N. Wilde, P. Matthews, and R. Huitt. Maintaining object-oriented software. *IEEE Software*, 10(1):75–80, January 1993.
- [122] M. Wood, M. Roper, A. Brooks, and J. Miller. Comparing and combining software defect detection techniques: a replicated empirical study. In M. Jazayeri and H. Schauer, editors, *Proceedings of The Sixth European Software Engineering Conference / Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 262–277, September 1997.
- [123] K. Yasumoto, T. Higashino, and K. Taniguchi. Software process description using LOTOS and its enactment. In *Proceedings of the 16th International Conference on Software Engineering*, pages 169–178, May 1994.
- [124] E. Yourdon. *Structured Walkthroughs*. Yourdon Press, fourth edition, 1989.

Appendix A

Future Directions in Computer Supported Software Inspection

Providing computer support for software inspection is an open-ended task. This appendix details some avenues of research considered for this thesis, but not pursued. There are two main themes of research which require much consideration. The first concerns facilities to enhance the defect detection process, allowing inspectors to find more defects with less effort. One of the most important areas here is considered: the support of object-oriented software. The second theme concerns the collection of data from the inspection to predict and control the process. Several areas within this theme are described.

A.1 Applying Inspection to Object-Oriented Code

The last decade or so has seen an explosion in the use of object-oriented techniques, with languages such as C++ and now Java becoming incredibly popular. The object-oriented paradigm is claimed to provide a number of benefits [11]. These include improved modularity, data hiding and encapsulation via the class mechanism, which results in low coupling and high cohesion. Reuse via inheritance and generic classes is another quoted benefit, and reduced maintenance costs are also claimed. Given that inspection is supposedly the most cost-effective means of finding defects, and the popularity of object-oriented programming languages, it is surprising that there is no published experience of inspecting object-oriented code, as indicated by Jones [58] and supported by a search of the literature.

In fact the very features of object-oriented languages which are believed to deliver benefits also make the code more difficult to inspect. A similar effect has been found in when testing object-oriented software. The problems are compounded by the static nature of inspection: the effect of many object-oriented constructs can only be easily seen when the program is executed. Similar problems have been reported concerning maintenance of object-oriented code [121]. Like inspection, maintenance requires understanding of the code. Unlike inspection, maintenance is by definition performed on a complete (and hence theoretically executable) system.

A typical object-oriented system can consist of many classes each of which may contain many small methods. Each of these methods provides only a little functionality (for examples of this, see Wilde *et al.* [121]). Therefore to understand more than just trivial parts of the system, large numbers of these methods must be cognitively grouped together and chains of method invocations followed. This is similar to the “delocalised plans” proposed by Soloway *et al.* [106], which occur when conceptually related code is spread over spatially distributed parts of the program.

Inheritance also causes difficulty in understanding code due to the distribution of behaviour over several classes. With single inheritance, when an inherited method is called the inspector must traverse the inheritance hierarchy to find its definition. In deep hierarchies with many inherited classes the definition may take some time to locate, with the inspector moving further and further from the original code. Multiple inheritance causes similar problems, but these are exacerbated by having a number of paths to follow when searching for a method or feature definition.

Polymorphism is the ability to take more than one form. In object-oriented programming, it generally denotes the ability of a declaration to refer to more than one class of object. Polymorphism goes hand in hand with dynamic binding, which allows the function associated with such a reference to be inferred at run time. This contrasts with static binding, where the exact function call is known when the executable is produced. The concept of polymorphism is very powerful, but this power comes with a price. Ponder and Bush [98] have written about the problems that polymorphism causes for program understanding due to dependence on the dynamic data state of the program. The problem is especially acute when combined with inheritance. Essentially, the specific methods called when the program is executed depend on the state of data within the program and cannot be easily inferred from a static code listing.

Genericity is used to define a related family of classes. The class is defined with one or more type parameters which can then be used as normal types within the class definition. When the class is instantiated with the appropriate type, all instances of the argument are

replaced by the new type to produce a new class. Generic classes are difficult to inspect because the behaviour of the class depends on its instantiation.

So far it has been assumed that the entire system is capable of being inspected at once. In reality, most systems will be far too complex to be inspected in a single step, and will be partitioned into smaller sections according to the appropriate inspection rate. For a simple object-based system, the problem is no worse than for modular procedural code. For a system with a large inheritance hierarchy, the problem is much more difficult. There are many dependencies which must be resolved, and if the system is arbitrarily split then inspectors may be left with references to code which they have no access to. When inheritance is involved there is a problem similar to that found in testing, where although it is tempting to inspect a class in isolation, it must actually be inspected in the context of its parent classes because of the possibility of hidden interactions. On the other hand single methods may be too small a unit to inspect, with very little semantic information to allow an accurate characterisation of the behaviour of the system.

Two techniques which can make code more amenable to inspection are better programming styles, such as limiting the use of inheritance, and better documentation, such as that proposed by Soloway *et al.* [106] or Parnas *et al.* [94]. These are not considered further here. Instead, the opportunities for tool support for existing code are investigated.

Existing inspection tools treat code as a static document. From the above discussion it is clear that this is not sufficient, since the dynamic behaviour of the system is far more important with object-oriented code. The static nature of inspection should not change when it is applied to object-oriented code, however. Exploration of the dynamic properties of an object-oriented system should not require that the code be executable, otherwise the nature of the inspection itself has been altered and moved towards testing. The solution may come from intelligent browsers which allow the inspector to explore possible execution paths without requiring the entire system to be finished and executable. For example, given a polymorphic function call, the browser could list all possible functions which may be called in the subset of the system which is being inspected. The inspector is presented with a more active document which allows the dynamic properties of the program to be explored.

Tools designed to support maintenance have some value for inspection. One example is Valhalla, a prototype object-oriented development environment described by Wilde *et al.* [121]. This system provides object animation capabilities, allowing the viewing of messages passed between objects. This can aid understanding of the dynamic properties of the system. Instead of analysing static pages of text, the inspection would then consist of analysing these animations. Among the tools described by Crocker and Mayrhauser [24] are several which

would be useful for inspection. The *inheritance hierarchy generator* generates a graph of the inheritance relationship in the system, which can then be studied to enhance understanding of the system. A *code browser* can be used to display the program and, when used in conjunction with the *code slicer*, allows the view of the program to be limited by certain criteria, such as occurrences of a certain variable or method call.

Given that a major problem in inspecting object-oriented code is tracing method calls and references over several classes, it may be useful to have some form of reduced representation, providing an overall view of the code being inspected. Such a representation would be similar to that used by Seesoft, as described by Eick *et al.* [36]. Seesoft is a tool designed for visualising line-oriented software statistics. The main window consists of a number of columns, each of which represents a source code file. Within these columns, a horizontal line is used to represent a line of code within the file. These lines are coloured according to the value of some attribute, e.g. age. A separate scale is used to display the entire value range for this attribute. The user can click on values in this scale, or the columns and lines themselves to toggle each value on and off. This allows the display of code with just a certain value or a range of values, and allows the user to find useful patterns in the code. This type of tool could be extended to assist inspection of object-oriented code as follows. While inspecting code in a reading window the reduced representation would highlight the current line of code. If this line was a method invocation, the definition of that method would also be highlighted. The inspector could then immediately move to that definition, and so on. The history of such a progression may be stored and when the inspector comes to a suitable understanding of some method, it would be possible to quickly backtrack to the previous method, where this understanding could be applied. This process would continue until the original starting point was reached. By speeding the traversal between methods, it is easier for an inspector to gain an understanding of the code, forming a mental picture of the system with the reduced representation. This system could also be to help decide which code should be included in an inspection. By tracing method invocations, the classes required to perform the inspection could quickly be found.

Other visualisation systems designed for object-oriented code may provide help with inspection. For example, De Pauw *et al.* [96] describe a language independent visualisation system which uses a preprocessor to instrument programs with code which generates events. These events can be received by a visualisation application which can use the data to update one or more views of the program, such as an inter-class call matrix, showing patterns of communication between each class. Although intended for use in debugging and code tuning and relying on having the entire system available and running, the principles could be applied to

smaller sections of code. Visualisation could allow the inspection team to provide summary information on which methods and classes are used by each class. They can then use this information to partition code for inspection.

There is much research ongoing in program understanding and visualisation tools for object-oriented software. Software inspection is an ideal platform for evaluating this research, since a fundamental task during inspection is understanding code. Defect detection efficiency of the inspection can indicate the usefulness of a certain method or tool. Note that a fuller discussion of all the above issues can be found in [79].

A.2 Data Collection and Analysis

Gathering data concerning the inspection process is essential to improve and fine-tune the process [41]. While collecting such data manually is time-consuming and error-prone, automatic collection can occur transparently, with far greater accuracy. All effort can therefore be expended on the main task of inspection: finding defects. Implementing such data collection facilities is also vital when empirically investigating the inspection process. Data collected can provide insights into which aspects of the tool and process are working well, and it may be possible to measure improvements. IPDL already provides a basic facility to express data collection, although user-controllable data collection is not available in its implementation in ASSIST.

The data collected must be used in some meaningful way to justify its collection. Tool support provides an opportunity to automatically analyse such data and provide instant feedback on the inspection. It also allows non-traditional measures to be collected and to be used in ways specific to tool-based inspection. This section describes scope for data collection and the uses of such data.

A.2.1 Process Measurement

Gilb and Graham [41] define over fifty measures which should be collected. Measures defined by others, such as Ebenau and Strauss [33], are simply subsets of those provided by Gilb and Graham, perhaps using differing terminology. They can be divided into three different categories: size measures, duration measures and rate measures. These are described separately, along with representative examples and how they may be collected. Note that the names of these measures are those used by Gilb and Graham and do not necessarily reflect terminology used in other literature, although they are still valid measures in all inspection processes.

Size Measures

Size measures are the simplest to collect, involving simple direct measurement of such variables as document lengths and defect counts.

number of checkers - the number of inspectors actually searching for defects during the inspection. While this may appear to be a simple measure to collect, it is complicated by the fact that there may be one or more participants in the inspection who does not perform defect detection, e.g. the author.

items-noted - a generic count of all issues and defects raised by an individual inspector. This is simply the count of items in an inspector's defect list. If defect classification is used, the count can be subdivided according to type, class and severity.

pages-studied - the number of pages which have been inspected during individual preparation (where a page is a well-defined entity, e.g. a specific word count or line count). In its simplest form it is the length of the document under inspection, which is easy to automatically collect. A tool-supported system could also collect the exact amount of the document viewed. For example, the text browser in ASSIST has the idea of a "current focus", i.e. an area of the document currently under scrutiny. The sum of all parts of the document which have been the focus can be considered to be the amount inspected.

Duration Measures

Duration measures concern the amount of time required to perform certain activities.

checking-time - total time spent by all inspectors in individual preparation. This is another measure which is easy to define but more difficult to collect. With paper-based inspection, each inspector can make an estimate of their time, although this estimate is rather error-prone (either intentionally or unintentionally). With a tool-based inspection, it is theoretically easy to collect: the system simply measures how long the inspector uses the tool. In practice, however, an inspector may start up the tool then become sidetracked on another task, either on the computer or away from the computer. The result is an artificially inflated time. This can be partially overcome by monitoring the position of the mouse pointer and only counting time when the mouse pointer is within windows belonging to the inspection tool.

logging-meeting-duration - the time spent in the group meeting. This can be split into two subcomponents: the **logging duration** (time spent reporting issues) and the **discussion-duration** (time spent discussing issues). The purpose of keeping these subcomponents separate is to provide a more exact estimate of the time spent in inspection activities. A tool-based inspection could easily support the collection of these components by providing the moderator

with a control to indicate when the meeting goes from issue reporting to issue discussion, and vice versa.

Rate Measures

Rates are not measured directly. Instead they are calculated from various size and duration measures.

logging-rate - the number of items logged per minute during the group meeting. This is simply the total number of items logged divided by the logging-meeting-duration, and can be easily calculated by a support tool.

defect-density - the number of defects per page in the product. This is calculated by dividing the total number of defects found in the product by the size of the product. Again, this is easily calculated by the tool

efficiency - defects found per hour of time spent detecting and correcting defects. This is the total number of defects found divided by the total time spent detecting defects and removing them. This also easily calculated by the tool provided all time invested is logged.

A.2.2 General Process Feedback

In its most basic form, the data gathered can provide simple process feedback. A database of information from previous inspections would be stored by the tool, which could be queried on parameters such as the inspection process used, number of inspectors, product type inspected, and so on. One use of this data is to determine the parameters for the most cost-effective inspection of each product type, in terms of the process used, number of inspectors, etc.

Historical data for the appropriate inspection type can be compared with data from the current inspection. If the data from the current inspection appears to fall outwith the bounds considered “normal” for that inspection type, further action could be taken. For example, the number of defects found can be compared against the historical average, allowing the moderator to estimate whether the inspection has been sufficiently successful or if a re-inspection should be held. The defect profile (i.e. the relative mix of defect types found) could be studied to check for abnormalities [19]. Other measures, such as defect detection rate and time spent in inspection, can be considered in a similar manner.

A more formal approach to the above can be found in Statistical Process Control [17]. SPC is based around a control chart which allows process variations to be monitored. The chart plots the value of the attribute under consideration over time, the mean value of the attribute and an upper and lower control limit. The format of a control chart is shown in Figure A.1.

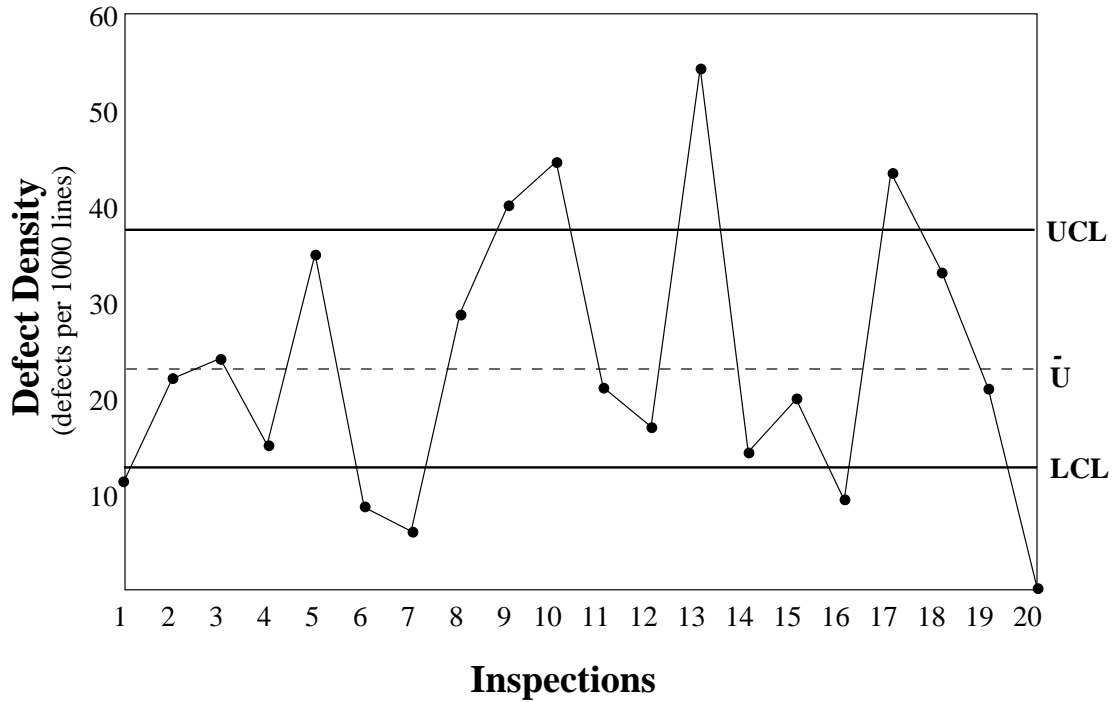


Figure A.1: The format of a control chart. The data presented is hypothetical.

The mean is defined as:

$$\bar{\mu} = \frac{\sum_{t=1}^n defects_t}{\sum_{t=1}^n part\ size_t}$$

Since the part size (i.e. document size) varies, Ebenau [32] suggests simplified upper and lower control limits, which are held constant over a set of analysed inspections. The upper control limit is calculated as:

$$UCL = \bar{\mu} + 3\sqrt{\frac{\bar{\mu}}{average\ work\ product\ size}}$$

The lower control limit is:

$$LCL = \bar{\mu} - 3\sqrt{\frac{\bar{\mu}}{maximum\ work\ product\ size}}$$

or 0, whichever is greater. Rather than use the average work product size, Ebenau recommends that the maximum size is used to “increase the sensitivity of the control chart to poorly inspected products”. If the value of the measure falls outwith these control limits, it may indicate the need for further investigation. This is performed using charts of dispersion, where other measures are investigated to determine whether they fall outside process norms. Ebenau [32] recommends that document size and inspection rates should be investigated if the defect density lies outside control limits. This can be used to determine if the inspection has been

performed correctly. An inspection support tool can be used to maintain and display these charts, and to even automatically flag inspections where measures fall outside their control limits.

A.2.3 Checklist Formation and Improvement

Checklists are an important aspect of the inspection process. A good checklist consists of hints that help inspectors quickly and easily find defects. Checklists must be dynamic. As new items are discovered (e.g. from previous inspection experience), they must be added to the checklist. As old items become obsolete they must be removed. The length of the checklist frequently becomes an issue, however. On one hand, the checklist should be as comprehensive as possible, to help find the maximum number of defects. On the other hand, the size of the checklist may hinder the inspector, with far too many items to check, or too many irrelevant items. Gilb and Graham [41], therefore, recommend that a checklist should not exceed one page, and should reflect the latest experience in the document type under review.

One approach to tackling this problem has already been described in this thesis. The C++ inspection environment described in Section 6.3 provides links between features in the code under inspection and the checklist in use. The checklist can therefore be longer and more comprehensive, with the system presenting only links to those items which are relevant. With this method, the inspection is driven by the product rather than the checklist.

Returning to a checklist-driven inspection, Chernak [20] describes a method for the statistical checklist formation and improvement, based on the analysis of defect data. Defects are classified using Orthogonal Defect Classification [21]. According to this system, a defect has two attributes. Chernak uses the *defect type* as an indication of where to look for that defect, while the *defect trigger* gives an indication of how to detect the defect. Multiple triggers can be associated with each type, corresponding to different defect types occurring in a single document feature. When existing defect data is classified according to this scheme, the defect types and triggers which cover the majority of defects can be extracted and used to prepare a checklist. In a tool-supported inspection environment, this could be accomplished automatically by allowing users to classify defects as they are found, as is possible in ASSIST. One drawback is that new checklist items could not automatically be generated and would have to be added manually.

An alternative approach in a checklist-driven inspection is to monitor checklist usage. The active checklists described in Section 6.2 allow inspectors to provide an answer to each checklist item. Monitoring these answers would allow the frequency of use of each checklist item to be found. Items which are used infrequently can either be automatically removed or

flagged for attention. The item can then be studied to determine why it is infrequently used. For example, it may be genuinely irrelevant or it may simply be badly written. Alternatively, the checklist can be prioritised, with less frequently used items assigned lower priorities. The inspector can then start with high priority items and move to lower priority items as time permits. As with the defect monitoring system described above, this system can only remove infrequently used items. New items still have to be generated and added manually.

In summary, while checklists are a vital part of the inspection process, a checklist which is too long is unmanageable and will be ignored. Hence, the checklist must either be shortened or relevant items brought to the inspectors attention. Tool support can help with both of these solutions.

A.2.4 Estimating Defects Remaining

Despite its effectiveness, an inspection is unlikely to find all defects in a product. Hence, it would be useful to estimate how many defects remain in the document. Ecological capture-recapture methods, used for estimating animal populations, have been suggested as a means to determine this, e.g. [35, 120]. When applied to wildlife, a number of animals belonging to the population under consideration are trapped and marked, then released back into the wild. After some time (to allow marked and unmarked animals to mix) a second trapping occurs. The ratio of marked to unmarked animals in this trapping can then be used to estimate the total population.

In the context of software inspection, each inspector locates a number of defects in the document, corresponding to a single trapping. The overlap between inspectors can provide an estimate of the total number of defects in the document. A number of different models can be used, allowing for differences in the difficulty of finding each defect and in inspector ability.

Computer support can apply capture-recapture methods to inspectors' defect lists to estimate the number of defects remaining, requiring only a slight change to the inspection process. This would consist of an extra phase after the inspection meeting, where inspectors tag each defect in their list to show which defect in the master list it corresponds to. The system can then use this information to produce the estimate. A number of models may be used, with the moderator being presented with a summary of estimates, on which the decision whether or not to re-inspect can be made. The tagging process can also be used by the system to gather data on the false positive/real defect ratio and meeting gains.

A.2.5 Inspector Experience and Behaviour

The final use of data gathered during inspections concerns individual inspectors. The use of performance data as a means of evaluating individuals is usually forbidden during inspection, however, other data can be collected and used for more constructive purposes.

Data on inspector experience can be gathered by the tool. Such experience includes the number of inspections performed, the document types inspected, roles played, and so on. This experience data can then be used by moderators to decide on appropriate participants for each inspection. For example, several experienced inspectors may be teamed with an inexperienced inspector for educational purposes. On the other hand, a critical document may require the participation of the most experienced inspectors. A tool supported environment could make suggestions about appropriate inspectors based on the type and criticality of the inspection about to be undertaken.

An inspector's behaviour during the inspection may also be recorded. This is important for three reasons. Firstly, data such as time spent in inspection must be collected to measure the inspection process, as discussed above. Secondly, it is useful in terms of process enforcement. Monitoring inspector behaviour can help decide whether the inspection task is being tackled in a uniform and correct way. For example, checklist usage may be monitored to ensure they are being uniformly applied. If participants have different responsibilities it is helpful to know that they are following these responsibilities and their efforts are not overlapping. Thirdly, data on how inspectors perform inspection may be useful as a means of understanding what makes a good inspection. Some strategies employed by inspectors will be more effective than others. If these strategies can be captured and shared with others, the overall inspection effectiveness can be increased. For example, a particular reading strategy for some documents may enhance defect detection. Tool supported inspection provides an opportunity for collecting fine-grain data on inspector behaviour which simply cannot be collected with paper-based inspection.

A.3 Conclusions

Two main areas of research in tool support for software inspection have been identified. One concerns methods for enhancing the defect detection capability of inspectors. Support of the object-oriented paradigm is one important area here, with much scope for the application of visualisation, maintenance and program understanding tools. This type of feature must be properly evaluated in an appropriate setting to gauge its usefulness.

The second major area concerns the collection and analysis of inspection data. Tool support allows many traditional inspection measures to be automatically collected and analysed. In addition, non-traditional measures, e.g. checklist usage, can also be captured and used to monitor and improve the process, providing even greater feedback. As with features intended to enhance defect detection, data collection facilities require testing with large amounts of real data to have confidence in their usefulness.

Appendix B

ASSIST V2.1 User Manual

B.1 Introduction

Asynchronous/Synchronous Software Inspection Support Tool (ASSIST) is a generic tool designed to allow the enforcement and support of any inspection process. This is achieved with a custom-designed process modelling language (Inspection Process Definition Language, or IPDL), and a flexible document type system. ASSIST is based on a client/server architecture, where the server is used as a central repository of documents and other data. ASSIST supports both individual and group-based phases of inspection. Group-based phases can be performed synchronously or asynchronously, with the choice of same-place or different-place synchronous meetings. This section provides an introduction to installing, starting and using ASSIST.

B.1.1 Requirements

This software requires Python 1.5, available from <http://www.python.org> and Tcl/Tk 8.0. Python must be compiled with support for Tk and dbm. ASSIST has been tested on Solaris 2.5.1.

B.1.2 Installation

Start by using `uncompress` and `tar` to uncompress and extract the file `assist.tar.Z`. A number of configuration options will have to be set for your system. These are detailed on a per-file basis.

```
assist/assist_server
```

Edit the following environment variables:

- PYTHON - your Python interpreter (full path).
- ASSIST_HOME - the location of the ASSIST directory (ending in `assist`, as this is the directory created when you untar the file)
- PYTHON_LIBS - the location of the standard Python libraries.

This file should be placed in an appropriate `bin` directory.

```
assist/assist
```

Edit the following environment variables:

- PYTHON - your Python interpreter (full path).
- ASSIST_HOME - the location of the ASSIST directory (ending in `assist`, as this is the directory created when you untar the file)
- PYTHON_LIBS - the location of the standard Python libraries.
- ASSIST_RX_HOST - the name of the machine on which the server is running.

This file should be placed in an appropriate `bin` directory.

```
assist/lib/assist_defs.py
```

ASSIST can make use of Netscape and Ghostview to view HTML and PostScript documents.

This file defines where these executables can be found with the following lines:

```
NETSCAPE = '/usr/X/local/netscape'  
GHOSTVIEW = '/usr/local/gnu/bin/ghostview'
```

The paths should be changed to those appropriate for your system, or to any other HTML and PostScript viewers you may have.

The tools to support distributed inspection require several definitions. These are the TTL value, the multicast address to use and the multicast port to use. They are defined with the following lines:

```
MULTICAST_TTL = 2
MULTICAST_ADDRESS = '224.0.1.0'
MULTICAST_PORT = 5000
```

The address and port can usually be left as is, however they may require changing if permanent conflicts arise. The TTL value must be changed if you require to hold distributed inspections between non-local machines.

Finally, the executables to be used must be defined. The following lines perform this function:

```
WB_PATH = '/usr/local/mbone/bin/wb'
NV_PATH = '/usr/local/mbone/bin/nv'
VAT_PATH = '/usr/local/mbone/bin/vat'
```

These paths should be changed as appropriate for your system.

B.1.3 Starting the Server

The server is started by typing

```
assist_server
```

at the prompt. After a few seconds, two windows will appear, as shown in Figure B.1. The main window has two panels. The left hand side is an information window in which messages pertaining to the state of the server appear. These mainly indicate client connects/disconnects. The right hand side contains a list of clients currently connected. There are only two controls on the server. The **Remove User** button disconnects a client. To use this, simply select the name of the required client then click the button. This should only be used if the machine on which that client is running has crashed, leaving the server in an inconsistent state. The **Close** button closes down the server, but only if all clients have been disconnected. The **Remove User** button may be used to clear any clients remaining, if required. The second window display status messages from the Discourse server.

B.1.4 Starting the Client

Provided you have been entered into the personnel database, ASSIST may be started by typing

```
assist
```

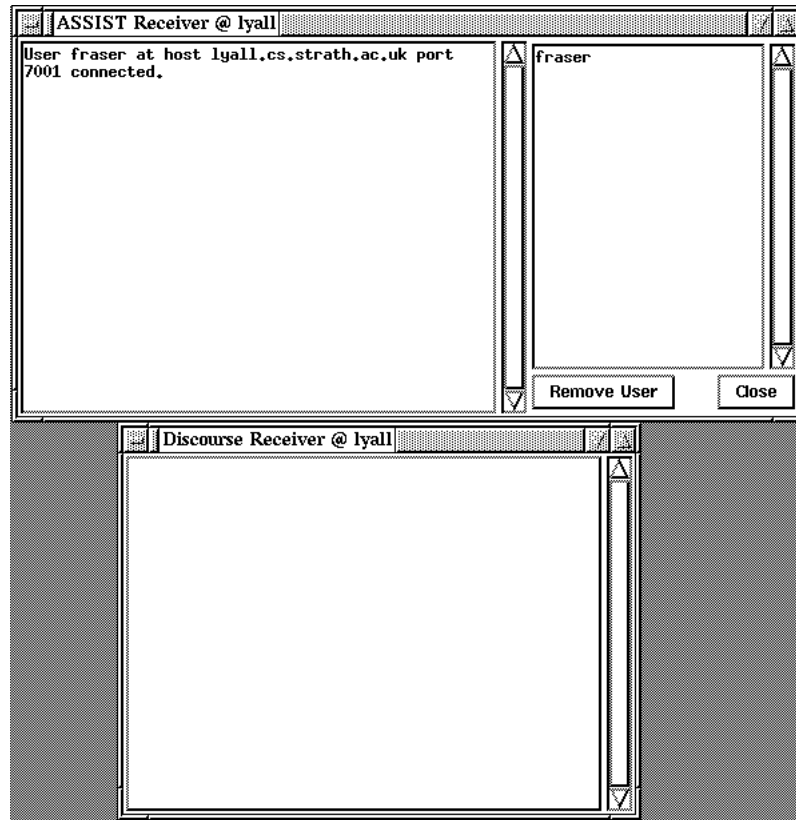


Figure B.1: The ASSIST server.

at the prompt. After a few seconds the main ASSIST window will appear, as shown in Figure B.2. It consists of a list of pending inspections and a number of menus. The functionality available via these menus will depend on the role that has been defined for you within ASSIST. For example, an inspector simply has the facility to continue an existing inspection, while a moderator will have the facility to start a new inspection. The following menus are available:

- **File**
 - **About** Provides copyright and author information on ASSIST.
 - **Quit** Quits ASSIST.
- **Inspection**
 - **New** - Starts a new inspection, allowing you to associate personnel and documents with the inspection. See Section B.2.5.
 - **Continue** - Continues an inspection from where you last left off. You must select an inspection from the list before you click the button. Alternatively, you can

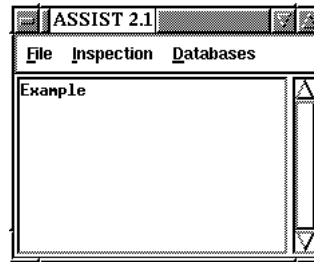


Figure B.2: The main ASSIST window.

double-click directly on the name of the inspection you wish to continue. See Section B.3.

- **Databases**

- **Document** - Allows you to browse and edit the database containing the documents available to ASSIST. See Section B.2.2.
- **Personnel** - Allows you to browse and edit the personnel database, where details of personnel available to perform inspections are stored. See Section B.2.3.
- **Process** - Allows you to enter a new inspection process, or to edit an existing one. See Section B.2.4.

B.2 Preparing for an Inspection

B.2.1 Introduction

Before you can use ASSIST to carry out inspections, there are several tasks which must first be undertaken. First of all, documents which are going to be inspected, along with documents used to support the inspection must be registered with ASSIST, which will then store copies of these in its own database. Secondly, personnel capable of carrying out the inspection task must be registered with the system. Finally, ASSIST must have your required inspection process entered and compiled. Although ASSIST provides some processes to get you started, it is inevitable that these will not completely match your needs. They can be copied and edited as required.

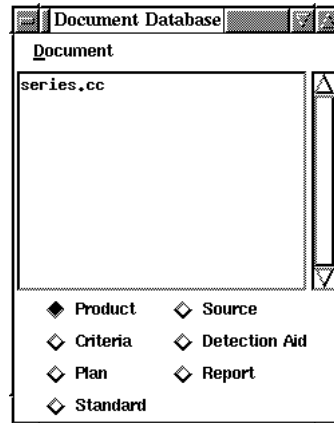


Figure B.3: The main document window.

B.2.2 The Document Database

A fundamental action is the registering of documents with the ASSIST system. This includes all products to be inspected, standards that may be used, detection aids which may be available, and so on. ASSIST will then make copies of these documents and allow them to be accessed from any client. Selecting the **Document** option from the **Databases** menu reveals the document control window (Figure B.3).

The window consists of a number of buttons for selecting the required document type. For each document type, the available documents are listed. Documents may be of one of the following fundamental types:

- **Product** - A document undergoing inspection.
- **Source** - A document used to produce the document undergoing inspection, for example, the design document for a section of code.
- **Criteria** - This document type is a list of criteria which must be satisfied. All criteria documents must be in a specific format which ASSIST can interpret. The format is described in Section B.6.
- **Standard** - The product will usually have to conform to a set of standards for that document type. These standards are used for compliance checking during the inspection. Other standards include the procedures to be used at each stage of the inspection, lists of terminology and so on.
- **Report** - A report simply details the outcome of a phase, or of an entire inspection. It is usually completed by the moderator. Like criteria, reports must be in a specific format

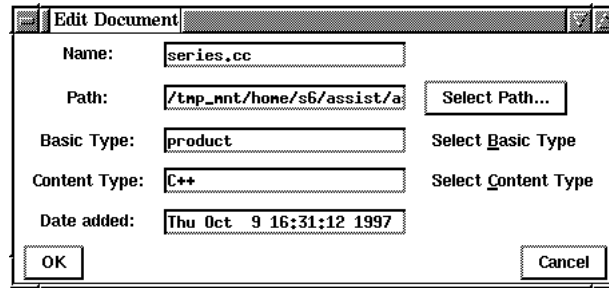


Figure B.4: The document edit window.

which ASSIST can interpret, described in Section B.6.

- **Plan** - The definitive description of the inspection process and the people who will be involved in it. Plans must also be in a specific format for ASSIST. See Section B.6 for more details.
- **Detection Aid** - A document which assists the inspector with finding errors, such as checklists.

The following items are available in the **Document** menu:

- **New** - Allows you to enter the details of a new document. The operation of this window is described below.
- **Edit** - Edits the current document, indicated by the current selection in the document list. A shortcut for this action is double-clicking on the appropriate document in the document list.
- **Delete** - Deletes the current document, indicated by the current selection in the document list. **Warning:** there is no 'undo' option to restore document details removed by accidental deletion.
- **Save** - Opens up a window allowing a copy of the document to be saved on your own file system.
- **Close** Closes the document database window.

Both the **New** and **Edit** commands bring up the same window (Figure B.4). The only difference is that the **Edit** option preloads the data entry fields with the values for the current document, which can then be edited, while the **New** option ensures that all fields are blank. The fields and associated controls are as follows:

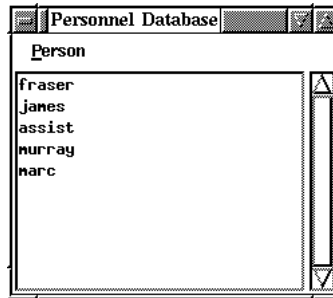


Figure B.5: The main personnel window.

- **Name** -The name by which the document is known within ASSIST. This should be fairly descriptive (e.g. 'ASSIST Server Design')
- **Path** - Clicking on the **Select Path** button brings up a dialogue box allowing the selection of the physical document to be associated with this entry. When the document details are completed, this document is copied across to the server.
- **Type** - Indicates the fundamental type of the document. Use the **Select Type** menu to select the appropriate type.
- **Content Type** - Indicates the type of the contents of the document. The default is ASCII, and more can be added by the user (see Section B.8). Use the **Select Content Type** menu to select the appropriate content type.
- **Date added** - The date indicates when this document was added to the database. This field is not editable; it is automatically filled when a new document is added, or an existing document updated.
- **OK** - Submits the new or updated details to the server, and closes the window.
- **Cancel** - Closes the window, ignoring any changes or additions made.

B.2.3 The Personnel Database

Before an inspector can use ASSIST, that person must be registered with the system. This is achieved with the personnel database function (only available to administrators). Selecting the **Personnel** option in the **Databases** menu reveals the window shown in Figure B.5.

The following actions are available under the **Person** menu:

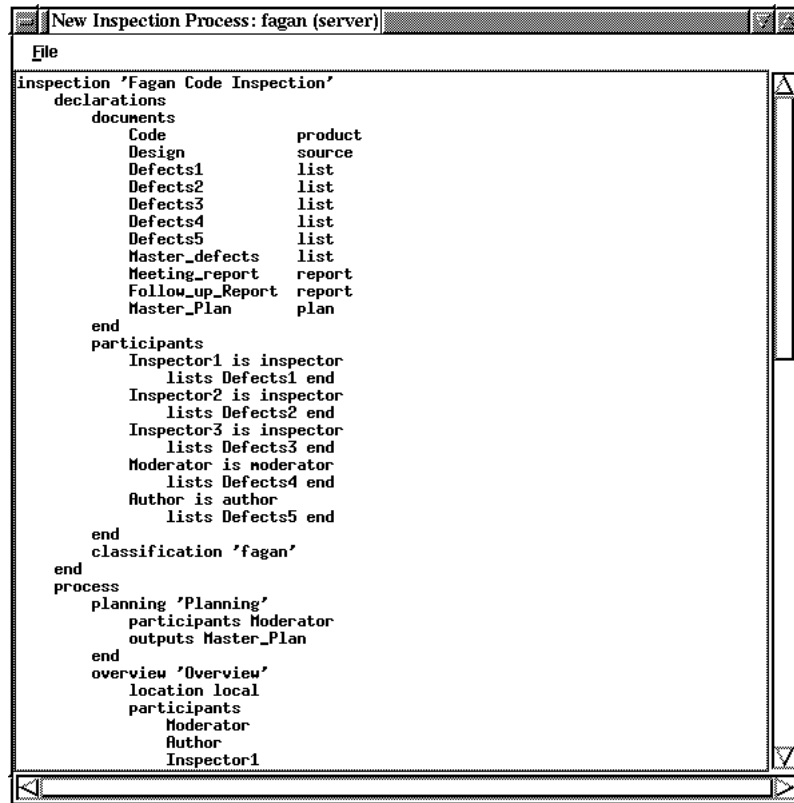
- **New** - Allows you to enter the details of a new person. The operation of the window for this facility is described below.

Figure B.6: The personnel edit window.

- **Edit** - Edits the details of the current person, indicated by the current selection in the personnel list. A shortcut for this action is double-clicking on the appropriate person in the personnel list.
- **Delete** - Deletes the current person, indicated by the current selection in the personnel list. **Warning:** there is no `undo` option to restore personnel details removed by accidental deletion.
- **Close** - Closes the personnel database window.

Both the **New** and **Edit** commands bring up the same window (Figure B.6). The only difference is that the **Edit** option preloads the data entry fields with the values for the current person, which can then be edited, while the **New** option ensures that all fields are blank. The fields are as follows:

- **User name** - The person's UNIX login name. If users from more than one UNIX system are expected to use ASSIST (e.g. over a WAN), it is the administrators responsibility to ensure that login names do not clash.
- **Name** - The person's real name.
- **Email** - The person's e-mail address (used for sending notifications from ASSIST).
- **Moderator** - This checkbutton indicates the person's ability to moderate an inspection.
- **Administrator** - This checkbutton indicates the person's ability to perform administrative tasks, such as adding new personnel to the system.



```

File
inspection 'Fagan Code Inspection'
  declarations
    documents
      Code          product
      Design        source
      Defects1      list
      Defects2      list
      Defects3      list
      Defects4      list
      Defects5      list
      Master_defects list
      Meeting_report report
      Follow_up_Report report
      Master_Plan   plan
    end
  participants
    Inspector1 is inspector
      lists Defects1 end
    Inspector2 is inspector
      lists Defects2 end
    Inspector3 is inspector
      lists Defects3 end
    Moderator is moderator
      lists Defects4 end
    Author is author
      lists Defects5 end
  end
  classification 'fagan'
end
process
  planning 'Planning'
    participants Moderator
    outputs Master_Plan
  end
  overview 'Overview'
    location local
    participants
      Moderator
      Author
      Inspector1

```

Figure B.7: The new process window, with a process loaded.

B.2.4 The Process Database

Another requirement before starting an inspection is having an inspection process for ASSIST to enforce. Although ASSIST comes with several well-known inspection variants, it is highly likely that some variations of these processes, and even completely new processes, will be required. Selecting the **Process** item from the **Databases** menu gives access to the facilities for creating and compiling a new inspection process. This section presents ASSIST's facilities for entering and compiling processes. For more details on writing new IPDL processes, see Section B.4 and Section B.5.

The New Inspection Process Window

This window (Figure B.7) allows you to enter a new inspection process, or to edit an existing process, which can then be compiled ready for use. The main feature of this window is a simple text editor for entering processes. The remaining features are available from the file menu:

- **New** - Clears the current process, allowing a new process to be entered from scratch. This is the default state when the window is first opened.
- **Load** - Loads an existing process, either from the server, or from a local file. The process can then be edited or refined as required. The title of the window changes to reflect the process name and where it was loaded from.
- **Save** - Saves the current process under the current name. This option is only available if the process has previously been saved using **Save As**, or if the process has been loaded from the server or from a file.
- **Save As** - Allows the current process to be saved under a new name. This is the only save option available for a newly entered process. You can choose between saving the process on the server, or to a local file. After this operation, the title of the window changes to reflect the process name and where it was saved to.
- **Compile** - Only if a process is successfully compiled will it become available for use, and an edited process must be compiled for the changes to become available. The compile option is only available when the process has been loaded from or saved to the server. Processes loaded from local files cannot be compiled until they are saved on the server.
- **Delete** - Brings up a requester allowing a process on the server to be deleted. Processes can be deleted by double-clicking on the name, or by single-clicking on the name then selecting the **OK** button.
- **IPDL Helper** - Starts a helper application to provide a skeleton process description. See Section B.2.4 for more details.
- **Close** - Close the new process window, without saving the current process.

The IPDL Helper

The IPDL Helper is designed to ease the task of creating a new inspection process by creating a skeleton process which can be fleshed out. By entering the documents for this inspection, the participants who will take part, and the number of phases required for the inspection, the helper will generate a partial process, leaving blanks where specific details should be filled in. Figure B.8 shows the IPDL helper window. The window is divided into three main sections. From top to bottom these are **Documents**, **Participants** and **Process**.

The screenshot shows the IPDL Helper window with the following sections:

- Documents:** A grid of eight empty boxes labeled Products, Reports, Sources, Detection Aids, Standards, Lists, Plans, and Criteria. Below this grid is a "Document Name" input field, "Add" and "Remove" buttons, and a "Document type" dropdown menu with options: Product, Report, Source, Detection Aid, Standard, List, Plan, Criteria.
- Participants:** A "Name (role)" list box on the left. To its right is a "Participant Name" input field, "Add" and "Remove" buttons, and a "Role" dropdown menu with options: Coordinator, Moderator, Inspector, Author.
- Process:** "Number of folds:" input field with value 1, "Number of meetings:" input field with value 0, "Add" and "Remove" buttons, and a row of checkboxes for Entry, Planning, Overview, Rework, Follow up, Exit, and Metrics.
- Meeting counts:** An empty list box on the left.
- Buttons:** "Generate" button at the bottom left and "Close" button at the bottom right.

Figure B.8: The IPDL helper window.

The **Documents** section allows you to enter the names of all documents which will be used and created during the inspection. To add a document, enter the name in the **Document Name** box and select the document type from the row of types, then click on **Add**. To remove an unwanted document, select the document from the appropriate list (with a single mouse click), then click on **Remove**.

The **Participants** section allows you to enter the details of the participants taking part in this inspection. For each participant required, enter the name in the **Participant Name** box, select the appropriate role for that person from the four available, then click on **Add**. The participant name and role will then appear in the list at the left hand side of the window. To remove a participant, select the name in the list then click on **Remove**.

The **Process** section allows you to enter some details of the process you require. A fundamental decision to be made is the number of independent folds for this inspection. A straightforward inspection will consist of only one, while a more rigorous inspection will involve more. Enter the number of folds in the box labelled **Number of folds**. For each fold, you

should then enter the number of meetings. Type each number into the box labelled **Number of meetings**, then click on **Add** to add it to the list of meeting counts at the left hand side. Meeting counts can be removed from the list by selecting the appropriate number and clicking on **Remove**. Finally, the optional phases of entry, planning, overview, rework, follow-up and exit can be added by selecting the appropriate checkbox.

When each item has been completed to your satisfaction, clicking on **Generate** creates a skeleton process which is loaded into the new process window. Clicking on **Close** closes the IPDL Helper.

B.2.5 Starting a New Inspection

Assuming you've created and compiled your required inspection process, you can now actually instantiate and run that process:

- Select the **New** option from the **Inspection** menu in the main ASSIST window (Figure B.2).
- Select the required process from the list presented.
- Enter a name for this inspection. Since this is the name by which the inspection will be known to its participants it should therefore be fairly precise (and almost certainly unique).

After a few moments, a screen will open allowing you to select the participants and documents for this inspection (Figure B.9). The **Inspection** menu in the top left hand corner has two items: **Start** takes the current details and starts the new inspection, while **Abort** closes the window without starting the inspection.

The window contains two scrolling lists of items which have to be completed to start the inspection: Documents and Participants. For each item in the documents section, clicking on **Select** brings up a list of available documents which are of the same type as the named document (as declared in the process definition for this inspection). One of these may then be selected and will be appear in the box below the document name. For reports, plans, criteria and detection aids only one document may be chosen. For the remaining document types, multiple documents can be chosen. Names can be removed by clicking on **Remove**, when a list of the documents selected will appear, allowing one to be chosen and removed. For each name in the participants section, a person registered with ASSIST may be selected and added in the same way as for a document. In this case, only one person can be selected for each part in the inspection. The list of people available for each part will depend on the qualifications

Figure B.9: The new inspection window, where documents and participants for the inspection are chosen.

required for each part, in that people who have not qualified as moderators will not be available when the part requires implementing the role of moderator or coordinator. The **Remove** button simply erases the person associated with the part.

When the required details have been entered and are correct, use the **Start** item in the **Inspection** menu to start the inspection. ASSIST will check that no duplicate participants have been entered, although duplicate documents are permissible.

B.3 Executing An Inspection

When an new inspection has been started, its name will appear in the pending inspection list of every inspector involved. To join that inspection double-click on its title. The main ASSIST window is then replaced by the *Execute* window shown in Figure B.10. If your participation is not required for this phase a message will appear informing you of this and you will not be able to join the inspection.

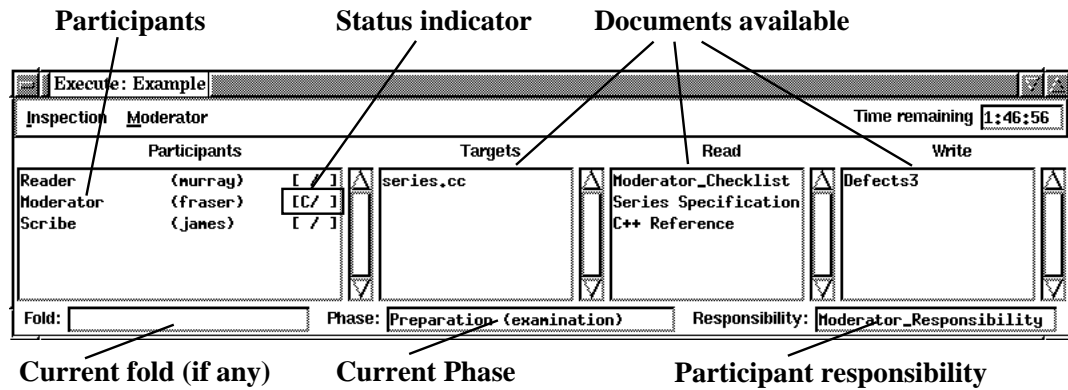


Figure B.10: The main execute window showing the people taking part in this phase and the documents available for use.

B.3.1 The Execute Window

The execute window contains two menus. The **Inspection** menu contains two entries. The **Finished** item is used by each participant to indicate that they have completed this phase. The **Close** button closes the execute window, saving the participant's state in the inspection. The **Moderator** menu appears only for the moderator of this inspection. It contains five entries. **Previous Phase** returns the inspection to the previous phase. **Next Phase** moves the inspection on to the next phase in succession. **Skip Phase** skips the next phase in succession, moving to the phase after the next. This item is only available during a consolidation type phase (see Section B.3.3). **Restart Inspection** returns the inspection to the first phase, while **Abort Inspection** is used to completely abort the current inspection, deleting all data. Each of these controls brings up a confirmation dialogue box before the action is carried out. Also at the top of this window is an indicator showing the time remaining for this inspection session, should such a limit be defined for the process being used.

The execute window also contains four lists. The **Participants** list contains the names of all participants involved in the current phase of the inspection. The name of the part they play in this inspection is followed by their user name in brackets. This is followed by a status indicator. A “C” indicates that person is currently connected to ASSIST and taking part in this inspection. If an “F” is showing, it indicates that the person has finished their work for this stage and used the **Finished** item in the **Inspection** menu. This information should be used by the moderator to decide when to move on to the next phase.

The remaining three lists contain the names of all documents available during this phase. By double-clicking on the document name, the appropriate browser will be opened with the document. **Targets** indicates documents which are the target of the phase. **Read** indicates

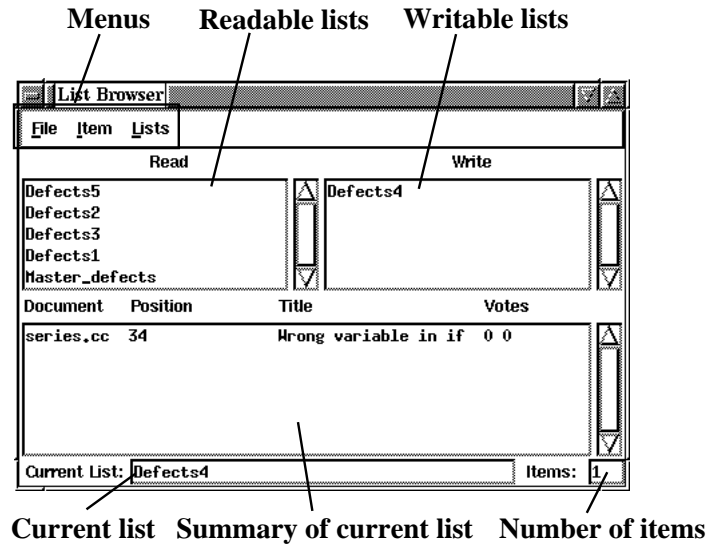


Figure B.11: The main list browser window.

documents which the participant may read but not alter. **Write** indicates documents which this participant may alter, such as lists and plans.

Finally, there are three further items along the bottom of the execute window. **Fold** indicates the current fold, and is only used during an N-Fold inspection. **Phase** indicates the current inspection phase. This contains both the user defined name of the phase and its type. **Responsibility** contains the name of any responsibility assigned to the participant.

The above description applies to a traditional single-fold inspection. The execute window which appears during an N-Fold inspection has further controls for the coordinator of that inspection. These are described in Section B.3.3.

B.3.2 Document Browsers

The List Browser

The List Browser is the ASSIST tool for handling list-type documents. Each inspector may have access to multiple lists, and the list browser may be used to add items to each list, remove them, move items between lists, and so on. The facilities of the browser may also be accessed from any active browser supplied with ASSIST (such as the text browser described in Section B.3.2). Lists are either readable or writable. Only writable lists may be edited. Figure B.11 shows the list browser for an inspector with six lists: five readable and one writable.

There are five main items in this window. The **Read** lists summary indicates all readable lists available to the inspector. The **Write** lists summary shows those lists which the inspector

can browse and edit. Double clicking on a name in either of these lists opens that list into the list summary area on the right. Each item within the lists is summarised into one line consisting of the file where the item occurs, the position of the item within that file, the title of the item, and the number of votes for and against the item. If an item has been voted on, a “V” will appear at the end of the item. The list currently open is indicated in the **List** item in the top right hand corner of the window, with the number of items in the list indicated by the **Items** entry.

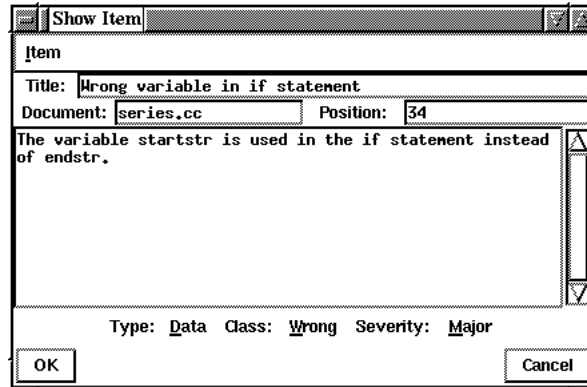


Figure B.12: The show item window.

Three menus appear in the top left-hand corner of the window. The **File** menu has two items: **Close**, which closes the browser, and **Print** which prints the currently selected list. The print facility uses the standard `lpr` command and prints to the line printer by default. Section B.8.1 describes how this may be altered.

The **Item** menu has entries relating to item manipulation. Almost all of these commands only work when an item has been selected (with single mouse click) in the summary window. Commands such as **Edit** and **Cut** will only work on writable lists. Two of these commands open a window like that shown in Figure B.12. The various fields of this window are as follows:

- **Title** - The title of the item.
- **Document** - The document in which this item occurs.
- **Position** - The position within the file where the item occurs.
- A free-form text description of the item.
- Classification buttons, which are used to set up to three classification terms for this defect. The classification scheme used depends on the process being executed. See

Section B.8.4 for details on providing your own classification scheme.

The **Item** menu in this window has four entries:

- **New** - Creates a new item as an annotation on the current item.
- **Propose/Vote** - During a synchronous collection meeting, the **Propose** option is available to allow participants to discuss items. During asynchronous collection meetings, the **Vote** option is available to allow participants to vote on items.
- **Show Source** - Sends a message to the appropriate browser to move the focus position to that of the current item.
- **Next** - If the current item has any annotations associated with it, their titles appear in this submenu. Selecting a title from this submenu opens a window containing that item.

Considering the main List Browser window, the commands available from the **Item** menu are:

- **Show** - Opens a window like that in Figure B.12 allowing the details of the item to be examined and edited. This can also be achieved by double-clicking on the relevant item in the summary window. If the item is from a writable list, it will be editable. Click on **OK** to save any changes and close the window, or **Cancel** to ignore the changes and close the window. Note that the document name can never be edited.
- **New** - Opens a window like that in Figure B.12 allowing details of a new item to be added. Click on **OK** to add the item, or **Cancel** to forget about the new item. If a new item is requested from a browser, the document name and position fields will already have been filled and cannot be edited.
- **Cut** - Removes the current item from the list and stores it internally for later retrieval in a **Paste** operation.
- **Copy** - Copies the current item from the list and stores it internally for later retrieval in a **Paste** operation.
- **Paste** - Provided an item has been cut or copied, this allows it to be pasted into the current list.
- **Show Source** - Sends a message to the appropriate browser to move the focus to the position of the current item, allowing the position of items within the appropriate file to be easily found.

- **Propose/Vote** - During a synchronous collection meeting, the **Propose** option is available to allow participants to discuss items. During asynchronous collection meetings, the **Vote** option is available to allow participants to vote on items.

ASSIST implements a voting mechanism to allow inspectors to decide on the validity of a item during a collection meeting. Voting can take place either synchronously or asynchronously.

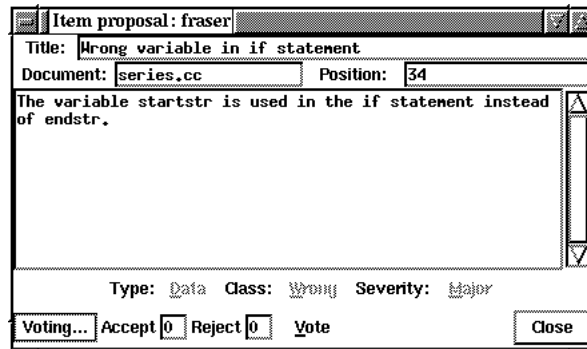


Figure B.13: The item proposal window.

Synchronous voting is achieved with the **Propose** item in the **Item** menu. When an item is proposed, a proposal window like that in Figure B.13 appears on all inspectors screens. This window is similar to the edit window, but has additional controls for voting, and counts of the number of votes cast. There are two buttons which may be used to cast votes on the item:

- **Accept** - indicates agreement with the item.
- **Reject** - indicates disagreement with the item.

When a vote has been cast, the vote counts are updated and the voting buttons become inactive for that inspector. If all votes have been cast (i.e. all currently connected participants have voted), the result is displayed in the bottom left hand corner. The proposal window may then be closed. Where the item is accepted, the scribe will have to choose a destination list. This list is usually some master list output from the meeting. After the list is chosen and the item added, the meeting can move on to the next issue. The scribe also has the option of editing the item, and can make use of the **Update** button to ensure that all participants see any proposed changed to the item before they vote on it. When a vote is underway, the scribe is usually not able to close the propose window until the vote is complete. In exceptional circumstances, the scribe may hold down <shift> and click on **Close** to force the window to close. Note that if a vote is tied, ASSIST currently forces the item to be accepted.

Asynchronous voting is achieved with the **Vote** item in the **Item** menu. This brings up a voting window similar to that in Figure B.13, and voting is carried out as above. Although votes are propagated as above, there is no requirement for every participant to vote simultaneously, and the item does not require a scribe to choose a destination item list. The scribe must manually copy items when consensus has been reached.

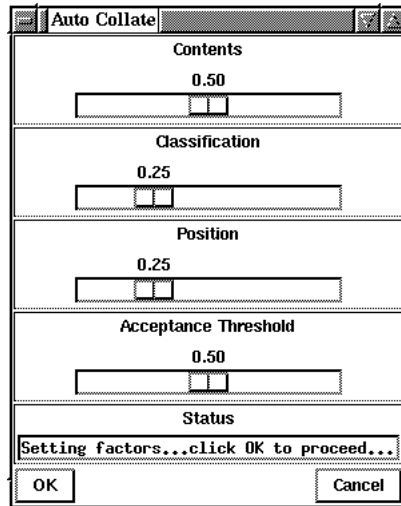


Figure B.14: The Auto Collate control window.

The **Lists** menu contains one item: **Auto Collate**. This function allows several lists to be combined into one with ASSIST removing duplicate entries. These duplicates need not be exact; ASSIST scores list items on position, content and classification. If two items match with a score above a user-defined threshold, one of the items will be discarded. Auto-collation starts by asking you to select a number of source item lists. These are the lists which you wish to merge together. The requester displays all lists available. Click on the name of each list you wish to include, holding down <shift> to select multiple lists, then click on **OK**. You can select a single destination list by clicking on its name then on **OK**. The control window shown in Figure B.14 then appears, allowing you to set various factors affecting the auto-collation process. The **Contents**, **Classification** and **Position** values indicate the relative importance of the appropriate part of the item when calculating the similarity between two items. The total of all these factors must sum to 1, hence increasing (decreasing) one factor decreases (increases) the others. **Acceptance Threshold** is the value of similarity that two items must have to be declared duplicates. The higher the threshold value, the more similar two items must be to be declared duplicates. However, too high a threshold will result in no matches being made. Clicking on **OK** starts the auto-collation process. Status messages appear in the control window to indicate progress. When auto-collation is complete, the control window

disappears.

The Text Browser

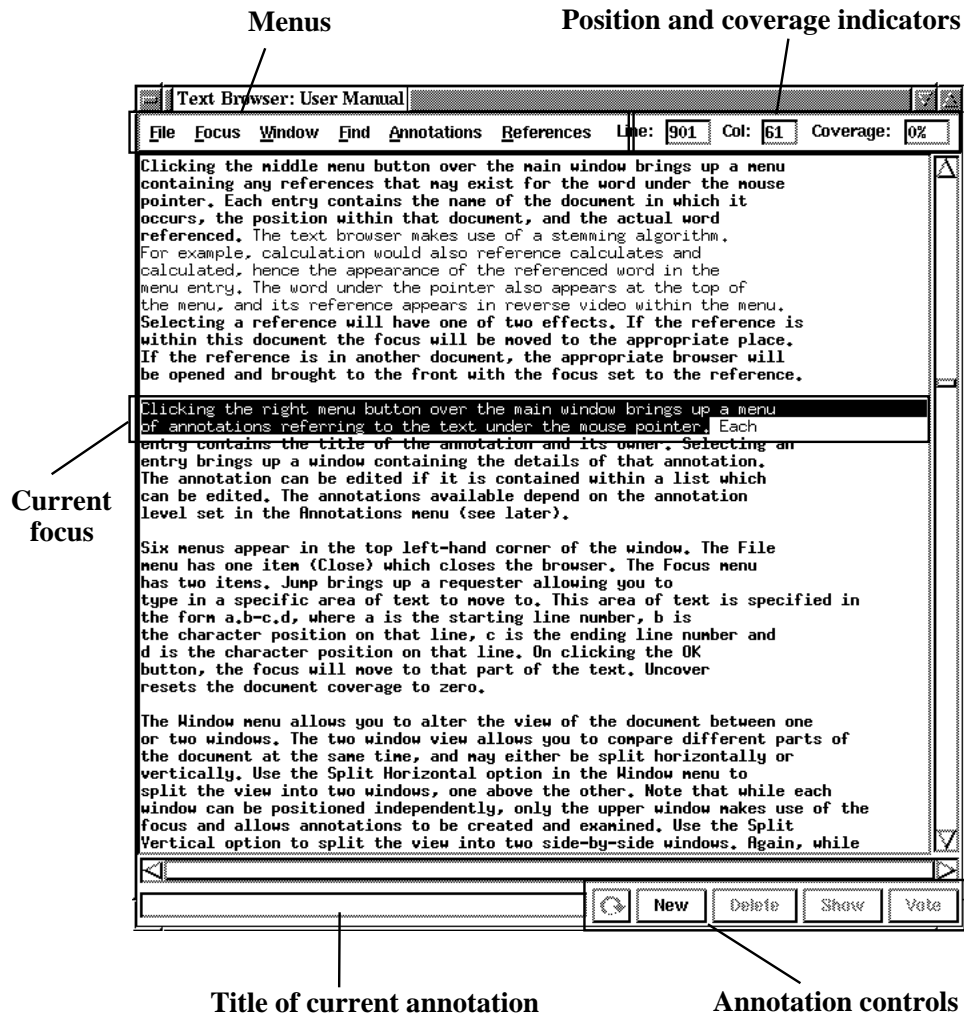


Figure B.15: The text browser.

The text browser (Figure B.15) is a text-only document viewer with fine-grain annotation facilities. The browser is based on the idea of a current focus, i.e. an area of text which is currently under scrutiny. The focus may consist of any contiguous area of text. The current focus can be annotated, or existing annotations can be read or edited. When a synchronous group meeting is being held, the current focus is controlled by the reader, and as the reader moves the focus so too does the focus of all other participants. The current focus is indicated by reverse video. The focus is set by holding down the left (select) mouse button and dragging over the required area of text. The selected area of text is now the focus and appears in

reverse video. Annotations referring to any part of the selected area then become available for manipulation, or annotations referring to the whole area can be added.

Annotations are indicated on the text in one of two ways. If your display is monochrome, text which has been annotated appears underlined. If you have a colour display, annotated text appears as black characters on a red background. The intensity of the background indicates the number of annotations for the text: the brighter the red, the more annotations there are, up to a maximum of eight different shades.

The browser also introduces the concept of coverage, i.e. the amount of the document which has been inspected. An area of text which has been the focus is considered to have been inspected, and therefore counts towards the coverage total. This total is shown in the top left corner of the window, along with the current line number. Inspected text is shown in a reduced weight font.

Clicking the middle menu button over the main window brings up a menu containing any references that may exist for the word under the mouse pointer. Each entry contains the name of the document in which it occurs, the position within that document, and the actual word referenced. The text browser makes use of a stemming algorithm. For example, `calculation` would also reference `calculates` and `calculated`, hence the appearance of the referenced word in the menu entry. The word under the pointer also appears at the top of the menu, and its reference appears in reverse video within the menu. Selecting a reference will have one of two effects. If the reference is within this document the focus will be moved to the appropriate place. If the reference is in another document, the appropriate browser will be opened and brought to the front with the focus set to the reference.

Clicking the right menu button over the main window brings up a menu of annotations referring to the text under the mouse pointer. Each entry contains the title of the annotation and its owner. Selecting an entry brings up a window containing the details of that annotation. The annotation can be edited if it is contained within a list which can be edited. The annotations available depend on the annotation level set in the **Annotations** menu (see later).

Six menus appear in the top left-hand corner of the window. The **File** menu has one item (**Close**) which closes the browser. The **Focus** menu has two items. **Jump** brings up a requester allowing you to type in a specific area of text to move to. This area of text is specified in the form `a . b - c . d`, where `a` is the starting line number, `b` is the character position on that line, `c` is the ending line number and `d` is the character position on that line. On clicking the **OK** button, the focus will move to that part of the text. **Uncover** resets the document coverage to zero.

The **Window** menu allows you to alter the view of the document between one or two

windows. The two window view allows you to compare different parts of the document at the same time, and may either be split horizontally or vertically. Use the **Split Horizontal** option in the **Window** menu to split the view into two windows, one above the other. Note that while each window can be positioned independently, only the upper window makes use of the focus and allows annotations to be created and examined. Use the **Split Vertical** option to split the view into two side-by-side windows. Again, while each view may be positioned independently, only the left-hand window makes use of the focus and allows annotation. Use the **Join** option to return to a single window view. During each of these operations, the windows are positioned to show the current focus. **Create Separate** creates a completely separate window containing the text.

The **Find** menu provides access to a simple search mechanism.

- **Find Forward** - brings up a dialogue box allowing you to enter an expression to be searched for. The search proceeds forwards from the current position of the cursor.
- **Find Backward** - brings up a dialogue box allowing you to enter an expression to be searched for. The search proceeds backwards from the current position of the cursor.
- **Find Again** - repeats the last find operation in the same direction as before, starting from the position at which the the last find stopped.

The **Annotations** menu has an entry for each annotation which refers to the current position of the cursor. Choosing an annotation from this menus brings up a window containing the details of the annotation and allowing it to be edited (if it belongs to an editable list). The menu also has another option allowing the annotation level to be set. This option has three choices. **All** allows the browser to display all annotations referring to this document. **Own** restricts annotations to those belonging to the user. **None** prevents any annotations from being displayed.

The **References** menu has an entry for each reference for the current position of the cursor. This menu operates in the same way as that described above for the middle mouse button.

The strip of controls along the bottom of the window provide quick access to the annotation functions available from the List Browser. See Section B.3.2 for a detailed guide to using the list browser.

- **Cycle** - is used to select between annotations if there is more than one for the current focus. The **Cycle** button is labelled with a circular arrow.
- **New** - presents a window allowing you to enter a new annotation for the current focus.

- **Delete** - removes the currently selected annotation.
- **Show** - displays the details of the current annotation, if one exists, also allowing the annotation to be edited.
- **Propose/Vote** - During a synchronous collection meeting, this button appears as **Propose**, allowing you to propose the current annotation to the whole meeting. During an asynchronous collection, the button appears as **Vote**, allowing you to cast an asynchronous vote on the item.

The Code Browser

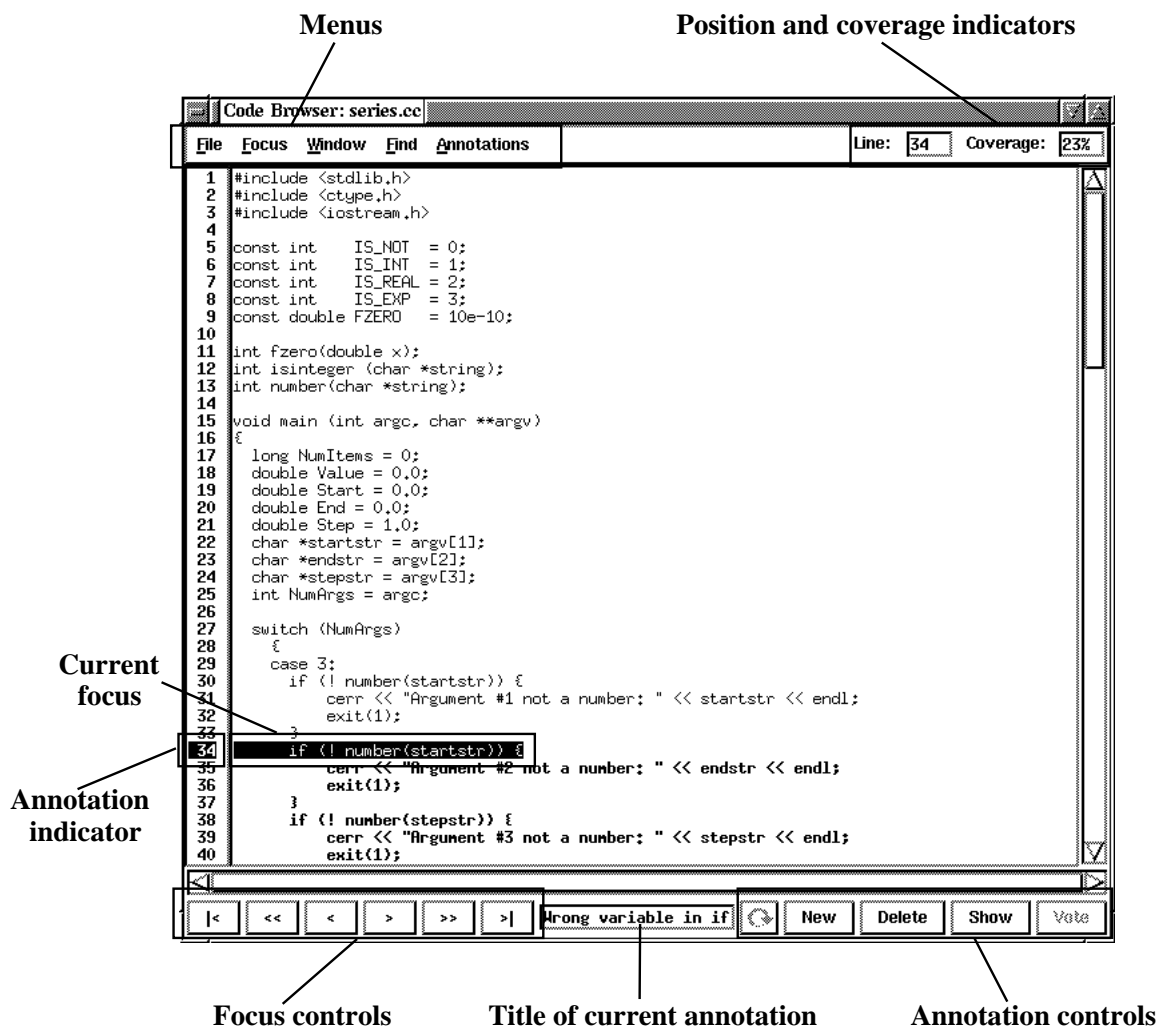


Figure B.16: The code browser.

The code browser (Figure B.16) is a text-only source code viewer with annotation facilities. Like the text browser, this browser is also based on the idea of a current focus. This time the focus consists of a single line of code. The current focus can be annotated, or existing annotations can be read or edited. When a synchronous group meeting is being held, the current focus is controlled by the reader, and as the reader moves the focus so too does the focus of all other participants. The current focus is indicated by reverse video. The focus can be set in a number of ways. The up/down cursor keys move the focus up or down one line. Holding down the <shift> key at the same time moves the focus ten lines in either direction. The focus can also be directly set by clicking on a line. Other controls are available for moving the focus backwards and forwards (see later).

This browser also has the concept of coverage. A line which has been the focus is considered to have been inspected, and therefore counts towards the coverage total. Inspected lines are shown in italics. The browser also performs line numbering, which appears to the left hand side of the main text window. Line numbers appear in reverse video when that line has one or more annotations associated with it. Unlike the text browser, there is no indication of annotations on the text itself.

Six menus appear in the top left-hand corner of the window. The **File** menu has one item (**Close**) which closes the browser.

The **Focus** menu has controls to allowing the current focus to be moved.

- **Start** - moves the focus to the first line of the document.
- **FastRewind** - moves the focus ten lines backwards.
- **Rewind** - moves the focus to the previous line of the document.
- **Forward** - moves the focus to the next line of the document.
- **FastForward** - moves the focus ten lines forward.
- **End** - moves the focus to the last line of the document.
- **Jump** - brings up a requester allowing you to type in a line number. On clicking the **OK** button, the focus will move to that line.
- **Uncover** - resets the document coverage to zero.

The **Annotation** menu has controls for manipulating annotations. Most of these commands interact directly with the list browser. See Section B.3.2 for a detailed guide to using the list browser.

- **View** - displays the details of the current annotation, if one exists. This can also be achieved by double-clicking on the appropriate line.
- **New** - presents a window allowing you to enter a new annotation for the current line.
- **Delete** - removes the current annotation.
- **Edit** - allows you to edit the details of the current annotation.
- **Propose/Vote** - During a synchronous collection meeting, this option appears as **Propose**, allowing you to propose the current annotation to the whole meeting. During an asynchronous collection, the option appears as **Vote**, allowing you to cast an asynchronous vote on the item.
- **Cycle** - is used to select between annotations if there is more than one for this line.

The **Window**, **Find** and **Annotations** menus work in an almost identical manner to those in the text browser. The only difference worth noting is that annotations can only refer to single lines. Annotation at a lower level is not available.

In the top right hand corner you can find an indication of the current line number (**Line**) and the amount of the document which has been covered, in terms of text examined as a proportion of the total amount of text in the document (**Coverage**).

The strip of controls along the bottom of the window provide quick access to the most used menu functions. The focus controls duplicate the first six entries in the **Focus** menu (from left to right: **Start**, **FastRewind**, **Rewind**, **Forward**, **FastForward** and **End**).

The annotation controls duplicate the **Cycle**, **View**, **New**, **Delete**, **Edit** and **Propose/Vote** items of the **Annotation** menu (note that the **Cycle** button is labelled with a circular arrow). The central text gadget displays the title of the current annotation, if any.

The C++ Browser

The C++ browser is identical to the text browser in all but one respect. Instead of simple keyword cross-referencing, it provides C++ specific cross-referencing, such as links between function declaration and usage. To gain the maximum benefit from C++ cross-referencing, the C++ checklist (document type: `detection aid`, content type: `checklist`) and C++ Reference (document type: `standard`, content type: `Help`) supplied with ASSIST should be made available during the inspection. The C++ browser will then make use of these to provide context specific checklist items and help.

The Simple Browser

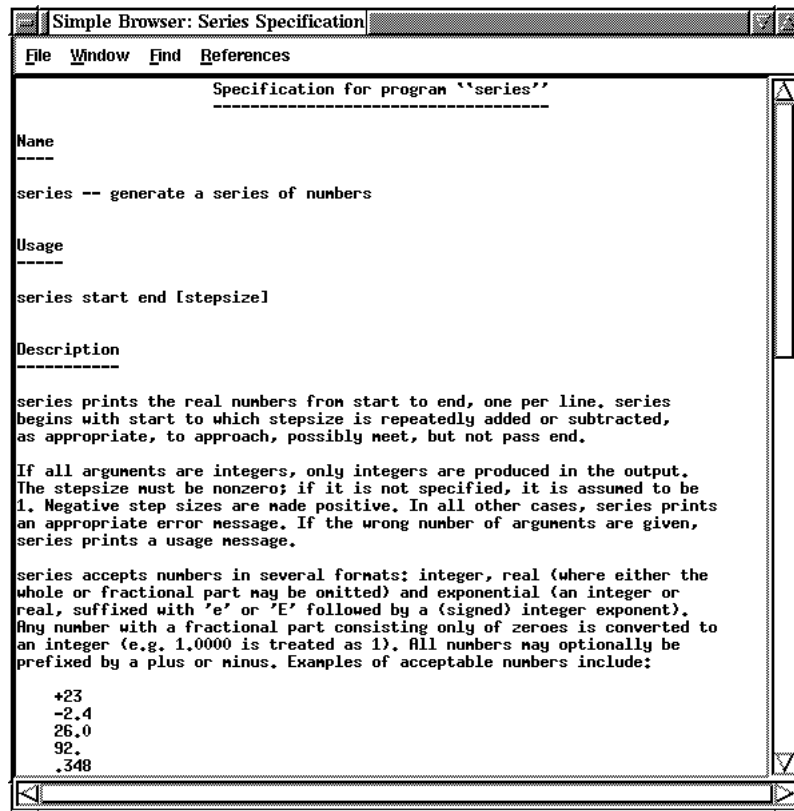


Figure B.17: The simple browser.

The simple browser, shown in Figure B.17 allows ASCII documents to be viewed. There are no annotation or focus controls, although find, window-splitting and cross-referencing facilities identical to those in the text browser are available via the menus. The middle mouse button can also be used to follow cross-references. Use **Close** in the **File** menu to close the browser.

The C/C++ Library Function Browser

The library function browser is available for `standard`-type documents, where the content type is `CLibraryFunctions`. It assumes the document consists of a number of descriptions of functions, each separated by a blank line. Each description should consist of a single line denoting the function header, followed by a blank line and ending with a paragraph explaining the function.

The top part of the window contains four menus, plus an indication of the current function

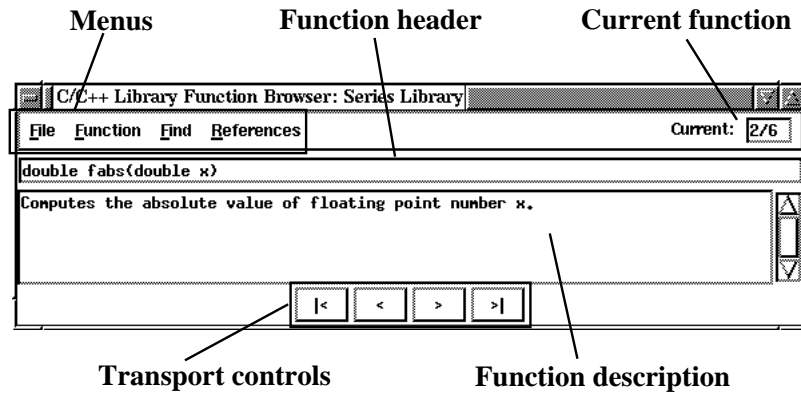


Figure B.18: The C/C++ library function browser.

being viewed, along with the total number of functions available. A single line of text is used to display the header of the current function, while a multi-line text box is used to display the description of the function.

The bottom of the window contains transport controls allowing you to scroll through all the functions available. From left to right these are, start, previous, next and end. These controls are duplicated in the **Function** menu.

The **Find** menu allows you to search for words within a function entry. Its operation is identical to that in the text browser (Section B.3.2), but a successful find jumps to the appropriate function. Both the header and the description are used in the find.

The **References** menu works in a manner identical to that of the text browser (Section B.3.2). References generated for library functions consist only of the function name. The **Reference** menu shows references for the name of the current function. References are also available via the middle mouse button. Finally, the **Close** option in the **File** menu is used to close the browser.

The Help Browser

The help browser (Figure B.19) implements a subset of HTML tags, allowing structured documents to be browsed. The document is organised into major sections, each of which may have a number of subsections, which may in turn have a number of subsections, and so on up to a maximum of six levels. A subsection consists of a title, the contents and a list of related subsections (under **See Also**). Each (sub)section is displayed on its own, with links to lower subsections appearing at the bottom of the page. Links under **See Also** and further subsections appear underlined, and the links can be followed by clicking on them. Other navigation facilities are available, and are described later. When the browser first appears, a contents page is

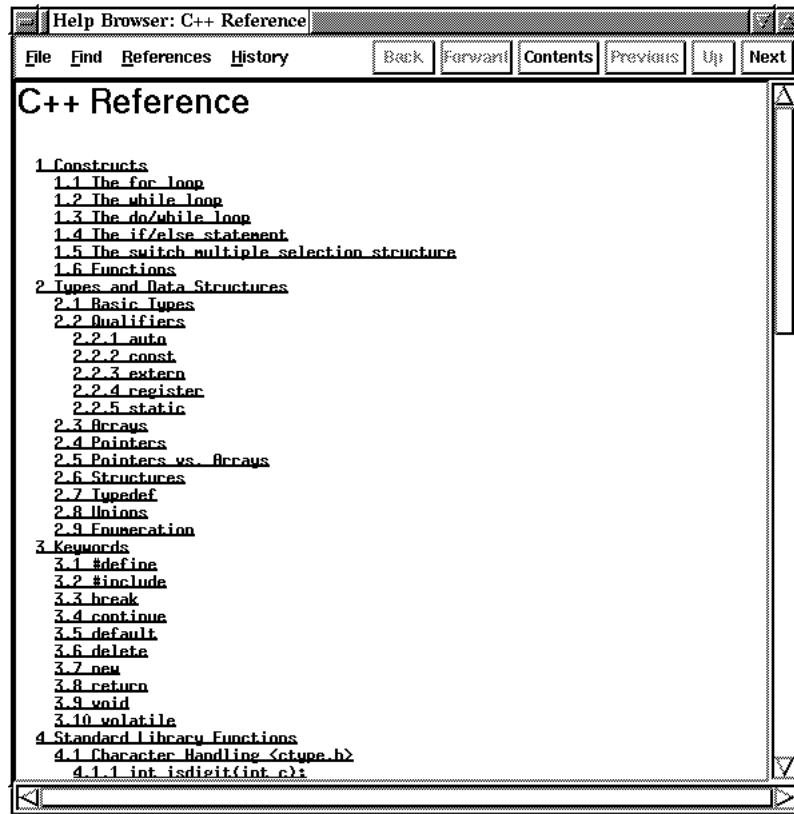


Figure B.19: The help browser.

shown, containing links to every (sub)section within the document.

Four menus can be found in the top left corner of the window. The **File** contains one item, **Close**, which closes the browser. The **Find** menu allows you to search for words in the document. Its operation is similar to that in the text browser (Section B.3.2), except that a successful find takes you to the page in the document where the word occurs. The **References** menu contains a list of references for the current page. Its use is identical to that of the text browser. The middle mouse button can also be used to access references for the current page.

The **History** menu contains a trace of up to 25 pages of the document last visited. The most recently visited page appears as the top entry, the least recent page appears as the final entry. Selecting one of these entries takes you directly to that page. An asterisk indicates the current page. Whenever a new page is visited, it is added after the current page in the history list. If the current page is somewhere in the middle of the history list, the most recent pages up to the current page are deleted and the new page added.

A number of shortcut buttons can be found in the top right of the window. **Back** takes you to the last page visited (as shown in the history list). **Forward** takes you to the next page in

the history list. **Contents** takes you to the contents page for this document. **Previous** takes you to the previous logical page in the document. **Up** takes you one level up in the document (e.g. from section 1.4 to 1). **Next** takes you to the next logical page in the document. The file format of help documents is described in Section B.6.2.

The Checklist Browser

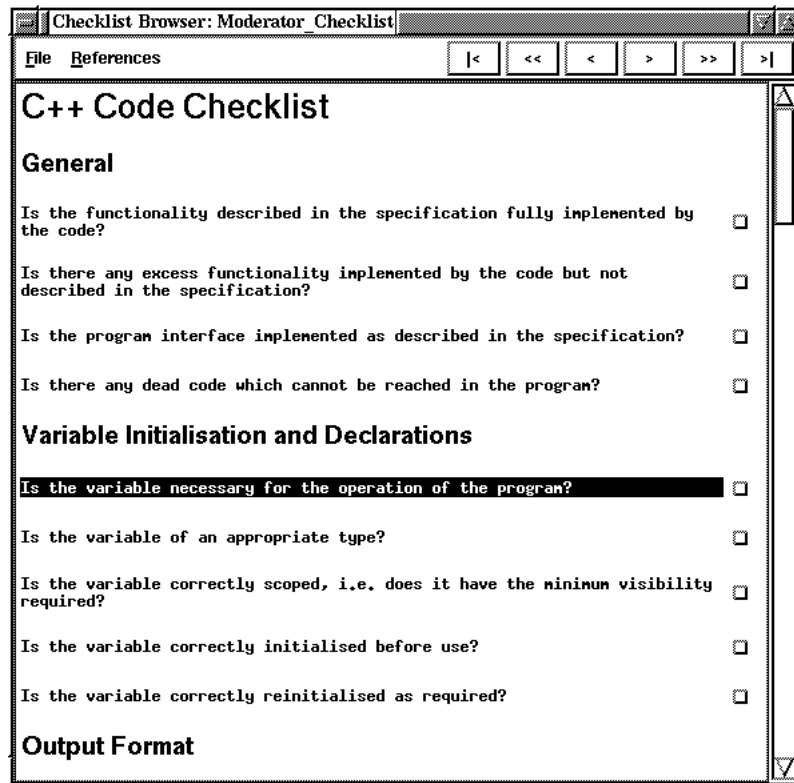


Figure B.20: The checklist browser.

If a document is of type `detection_aid` and is given the content type `checklist`, the Checklist Browser will be used to view it. A typical example is shown in Figure B.20. The window consists of a scrollable list of checklist items which may be answered. Each item may be one of five types, each of which is described in Section B.6.1. Answering a checklist item may involve supplying the correct answer for the given item, or it may simply require an answer to be given. See Section B.6.1 for details on the format of checklist documents.

One item at a time may be the focus of the browser. This item is displayed in reverse video. Clicking on an item forces it to become the current focus. Alternatively, the transport controls in the top left of the window can be used to move the focus from one item to the next.

From left to right these are: move to first item, move back ten items, move back one item, move forward one item, move forward ten items, and move to last item.

The **File** menu contains two entries. The **Check** entry forces the answer to each item to be checked against the required answer, if any. A message will appear informing you whether all items have been completed or not. The **Print** option prints out the checklist. The **Close** option in the **File** menu may be used to close the browser.

The **Reference** menu contains any references for the item which is the current focus. This menu works in the same manner as that in the Text Browser (Section B.3.2). References for each item can also be found by clicking the middle mouse button over that item.

The Criteria Browser

The criteria browser works in an identical fashion to the checklist browser described in Section B.3.2. See Section B.6.1 for details on the format of criteria lists.

The Plan Browser

The plan browser works in an identical fashion to the checklist browser described in Section B.3.2. See Section B.6.1 for details on the format of plans.

The Report Browser

The report browser works in an identical fashion to the checklist browser described in Section B.3.2. See Section B.6.1 for details on the format of reports.

Other Browsers

As distributed, ASSIST can also make use of Netscape and Ghostview to allow HTML and PostScript documents to be viewed. ASSIST also allows new browsers to be easily added. See Section B.8 for more information.

B.3.3 Process Phases

Entry and Exit

The purpose of the entry and exit phases is to ensure that some criteria are met before the start of the inspection and before the end of the inspection. Only output documents are defined for these phases, typically being a criteria list. Double-clicking on each output document opens the appropriate browser to allow the document to be completed. **Next Phase** in the **Moderator**

menu can then be used to move to the next phase. In the case of an exit phase, the inspection will then be complete.

Planning

Planning allows the moderator to decide on the final details of an inspection, such as the time for meetings to occur. This plan is then available to all participants throughout the entire inspection. The plan browser (Section B.3.2) allows the moderator to enter the final details. When the plan is complete, **Next Phase** in the **Moderator** menu can then be used to move to the next phase.

Overview

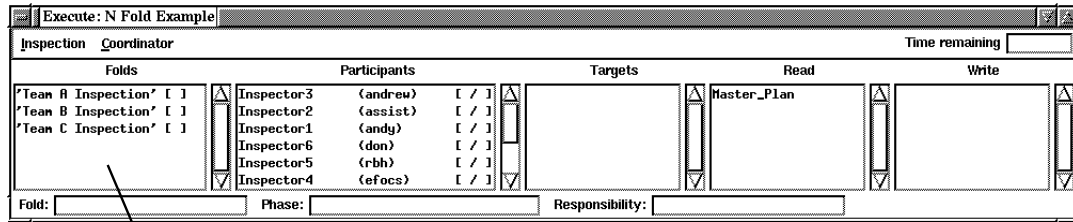
During the overview phase, a participant (usually the author of the material under inspection) guides the inspectors through the material they will be inspecting, giving background information and general guidance. With ASSIST, this stage may be held in a local environment, or may be distributed (indicated by the `location` statement in the process definition). A distributed overview will be supported by the standard ASSIST tools (see Section B.3.3).

Detection

Detection is the term used for the period of inspection where defects are found and catalogued. The detection stage in ASSIST can be arbitrarily complex, consisting of a number of separate detections (called folds), each of which can contain a number of sequential phases. Each of these phases may in turn be split into a number of parallel phases involving a subset of all inspection participants.

Single Meetings A single meeting may have one of three purposes: examination, detection or collection. Examination is concerned with becoming familiar with the target documents. Detection phases are designed for active defect hunting. Collection meetings are designed to merge the efforts of individual inspectors together. Meetings may be either synchronous or asynchronous. During a synchronous meeting, all participants are required to attend at the same time. During an asynchronous meeting, participants may “attend” (i.e., perform the required activity) at their own discretion.

Synchronous meetings may be held locally (i.e., all participants in the one location), or participants may be geographically distributed. ASSIST provides several communications facilities if a synchronous meeting is being held in a distributed fashion. There are four possible



Folds

Figure B.21: The coordinator's view of an N-Fold inspection.

communications channels allowing participants to share text, audio, video and diagrams. **Dis-course** provides a textual discussion medium. The window is divided into two parts. The upper part shows text broadcast by meeting participants, prepended by the name of the participant who broadcast the text. The lower part allows you to broadcast your text by simply typing it and pressing <return>. The tool **vat** is used to provide audio conferencing, while **nv** provides video and **wb** provides a shared whiteboard. See Section B.8 for details of how their use with ASSIST can be controlled. Full details on the use of these tools can be found in their respective manual pages.

During any meeting, the execute window provides each participant with documents which that participant can access. Double-clicking on the document name opens the appropriate browser. See Section B.3.2 for details of the browsers available.

Multiple Meetings A multiple meeting is simply a phase with multiple single meetings which take place in parallel. Each of these phases take place in the way described above, with the same facilities available.

Consolidation The consolidation phase allows the person in charge of the inspection to check the progress of the inspection and decide whether a further meeting may occur. During this phase, the **Skip Phase** item in the **Moderator** menu becomes active. If the moderator decides that the extra phase is not required, **Skip Phase** may be used to skip the next phase and move directly to the following phase. If the next phase is required, the **Next Phase** item can be used as normal. The other facilities available during consolidation are identical to the basic meeting facilities, allowing the defined documents to be viewed, reports completed, and so on.

N-Fold Inspections During an N-Fold inspection there may be two or more independent detection stages. Each stage is controlled by a moderator, while a coordinator is in overall

charge of the inspection. In an N-Fold inspection, the execute window for the coordinator looks like that in Figure B.21. An extra panel appears on the left hand side showing the status of each fold of the inspection. The name of each fold is followed by a status indicator which shows an “F” when the fold is complete.

During the organisation and completion phases of an inspection, the coordinator has the phase controls normally associated with the moderator (**Previous Phase** and **Next Phase**), along with the **Restart Inspection** and **Abort Inspection** controls, all in the **Coordinator** menu. When the inspection moves into the detection stage, the phase controls are disabled for the coordinator, and the **End N-Fold** control (in the **Coordinator** menu) becomes active. This control is used when each fold is indicated as being complete (“F” appears beside that phase), allowing the inspection to move to collation. At this point, the phase controls are re-enabled, and the **End N-Fold** control disabled.

Moderators still have the **Previous Phase**, **Next Phase** and **Skip Phase** controls, as in a single fold inspection, but these are only active during the N-Fold part of the inspection, and only apply to that fold.

Finally, the **Fold** item in the execute window becomes active during the detection stage for all participants except the coordinator. This shows the name of the fold in which you are involved.

An N-Fold inspection will finish with one or more collation meetings, allowing the results of individual folds to be collated. The collation phase provides similar controls to that of detection meetings. A scribe must be defined in the process description: that person will be able to edit output documents, such as collated defect lists and reports. The coordinator is also usually invited to these meetings, and may be given the role of scribe.

Rework

This phase allows a participant to modify the documents under inspection to account for the findings of the inspection. The phase definition allows a single participant to be defined, usually the author. During this phase, target documents become editable. All other facilities work as normal. An additional facility is provided by the list browser. When an item is opened, a button allows the author to mark the item as implemented. Use **Next Phase** in the **Moderator** menu to move on to the next phase.

Follow-up

During follow-up, a participant, usually the person in charge of the inspection, must examine the edited target documents to ensure they have been properly amended. The phase definition allows target and input documents to be specified; these are usually the edited products and appropriate lists of changes to be made. The facilities offered during this phase allow the participant to examine these documents. There may also be a number of reports to be completed. The report browser is available for that purpose.

Inspection Finish

When the final phase of the inspection has been completed, selecting **Next Phase** from the **Moderator** menu takes the inspection into the finished state, where the inspection is removed from the pending lists of all inspectors. If a target document has been edited during rework, use the document database to retrieve the edited version.

B.4 Writing IPDL Processes

The full definition of IPDL can be found in Section B.5. This section provides a tutorial introduction to defining a new inspection process.

B.4.1 Process Outline

An IPDL definition of an inspection is similar to a program written in a procedural programming language. Each inspection description consists of a set of declarations (for participants and documents), and a description of the process to be followed. The declarations, process and the entire description are delimited by keywords, and the whole process may be named. The outline is then:

```
inspection 'Example Inspection'  
  declarations  
  
  <...declarations go here...>  
  
end  
  
process
```

```
<...process description goes here...>  
  
end  
end
```

Note that IPDL makes no use of punctuation characters such as “;” and whitespace is not significant. However, it is advisable to use whitespace to promote the readability of your process descriptions. Note also that the name is an arbitrary string surrounded by single quotes. This applies to any other place in IPDL where a string is required.

B.4.2 Declarations

Declarations are used to introduce the documents and participants associated with the process. Responsibilities, consisting of sets of documents, may also be associated with each participant.

Document declarations consist of a list of document names and their types, surrounded by the keywords `documents` and `end`. Documents can be one of nine types:

- `product` - A document undergoing inspection.
- `source` - A document used to produce the document undergoing inspection, for example, the design document for a section of code.
- `criteria` - This document type is a list of criteria which must be satisfied. These documents must be in a specific format which ASSIST can interpret. The format is described in Section B.6.
- `standard` - The product will usually have to conform to a set of standards for that document type. These standards are used for compliance checking during the inspection.
- `report` - A report simply details the outcome of a phase, or of an entire inspection. It is usually completed by the moderator.
- `plan` - The definitive description of the inspection process and the people who will be involved in it.
- `detection_aid` - A document which assists the inspector with finding errors, such as checklists.
- `list` - A list of comments or annotations, including defect lists.

A typical document declaration list would be:

```
documents
    Code                product
    Design              source
    Individual Defects  list
    Moderator_Defects  list
    Maintenance_Defects list
    Standards_Defects  list
    Maintenance_Checklist detection_aid
    Maintenance_Requirements source
    Coding_Standard    source
end
```

Note that names are can be of arbitrary length and can consist of any alphanumeric characters, along with the underscore, and must not contain whitespace. When the inspection is instantiated and run, each of these document declarations will have a real document associated with it.

Responsibilities define a specific area of concern for an inspector. Each responsibility consists of a list of documents associated with that responsibility. The entire responsibility section is surrounded by the expected keywords:

```
responsibilities
    Maintenance requires
        Maintenance_Requirements
        Maintenance_Checklist
    end
    Standards requires
        Coding_Standard
    end
end
```

Each document within the responsibilities must have been previously declared.

The participants description is simply a list of people involved and the names of their roles. Note that the participant list is *not* a list of the real people involved; it simply lists the names of the 'characters' in the inspection and the roles they will play. Each definition consists of the participant name, the keyword *is* and the participants role. IPDL defines four possible roles:

- `coordinator` - person in overall charge of an N-Fold inspection.
- `moderator` - person in overall charge of a single-fold inspection.
- `author` - the author of the document under inspection.
- `inspector` - has the responsibility for finding defects.

The role may be followed by list and responsibility declarations, indicating the list documents belonging to that person and the responsibilities of that person, respectively. These must be previously defined. For example:

```
participants
  Moderator is moderator
    lists Moderator_defects
  end
  Inspector_Maintenance is inspector
    lists Maintenance_defects
    responsibility Maintenance
  end
  Inspector_Standards is inspector
    lists Standards_defects
    responsibility Standards
  end
end
```

Putting the above examples together gives a complete declarations section for an IPDL definition:

```
inspection 'Example Inspection'
  declarations
    documents
      Code                product
      Design              source
      Individual Defects  list
      Moderator_Defects  list
      Maintenance_Defects list
      Standards_Defects  list
      Maintenance_Checklist detection_aid
```

```

        Maintenance_Requirements source
        Coding_Standard          source
    end # Documents
    responsibilities
        Maintenance requires
            Maintenance_Requirements
            Maintenance_Checklist
        end
        Standards requires
            Coding_Standard
        end
    end # Responsibilities
    participants
        Moderator is moderator
            lists Moderator_defects
        end
        Inspector_Maintenance is inspector
            lists Maintenance_defects
            responsibility Maintenance
        end
        Inspector_Standards is inspector
            lists Standards_defects
            responsibility Standards
        end
    end # Participants
end # Declarations
process

....

end
end # Example Inspection

```

This example further demonstrates how the declarations section appears within the entire definition. It also introduces the comment character “#”. Anything appearing between this character and the next newline is entirely ignored.

B.4.3 Process Definition

The heart of IPDL lies in defining the inspection process itself. The process consists of three main stages: organisation, detection and completion. Each of these stages is tackled in turn in the following sections.

Organisation

The organisation stage may have up to three phases: entry, planning and overview. The phases must be defined in that order. A typical definition of entry is:

```
entry 'Entry'
    participant Moderator
    outputs Entry_criteria
end
```

This names the phase “Entry” and states that the single participant of `Moderator` should be present. The phase also produces a single document called `Entry_criteria`. Both the participant and document must be defined in the declarations section. Only a single participant may be specified, using the keyword `participant`, but multiple output documents may be specified (keyword `outputs`). Output documents must be either plans, reports, criteria lists, detection aids or lists, and this applies in any phase where output documents are required. The definition is delimited by the keywords `entry` and `end`.

Planning has a similar format, but both multiple participants and documents may be specified:

```
planning 'Planning'
    participants Moderator
    outputs Master_Plan
end
```

Overview is slightly more complex. Firstly, the location of the phase must be stated, either `local` or `distributed`. Along with the participants list (keyword `participants` - note the plural), a *presenter* must also be specified. This is the person who presents the overview material. Finally, three types of documents may be specified: `targets` (the document(s) under inspection), `inputs` (documents which are present at the phase but not edited) and `outputs` (documents produced or edited during the phase). Target and input documents may be of any type, while output documents have been previously defined. A typical example of an overview phase is:

```

overview 'Overview'
  location local
  participants
    Moderator
    Author
    Inspector1
    Inspector2
    Inspector3
  presenter Author
  targets Code
end

```

Detection

Detection is the most complex part of the process. It can consist of a single sequential stage, or a number of parallel “folds”. The simplest case of a single sequential detection stage is tackled first.

A single detection stage consists of one or more meeting phases, each of which may be followed by a consolidation step. A consolidation step consists of a consolidation meeting and an optional meeting; the consolidation meeting is used to decide if the extra meeting is required. An example of this is:

```

meeting 'Collection'
  objective collection
  timing synchronous
  location local
  visibility public
  participants
    Moderator
    Producer
    Reviewer1
    Reviewer2
  roles
    Reviewer1 is reader
    Reviewer2 is scribe
  targets Code

```

```
        inputs Design
        outputs Master_Defects
    end
    consolidation 'Consolidation'
        participant
            Moderator
        targets Code
        inputs
            Master_Defects
            Design
        outputs
            Unresolved_Defects
            Consolidation_report
    end
    meeting 'Group Review Meeting'
        objective collection
        timing synchronous
        location local
        visibility public
        participants
            Moderator
            Producer
            Reviewer1
            Reviewer2
        roles
            Reviewer1 is reader
            Reviewer2 is scribe
        targets Code
        inputs
            Unresolved_Defects
            Design
    end
```

The meeting definition introduces several new items. The keyword `objective` determines the objective of the meeting: either examination, detection or collection. A meeting

may be either `synchronous` or `asynchronous`, indicated by the `timing` clause. During a synchronous meeting, all participants must be present at one time, while an asynchronous meeting allows participants to attend at their own discretion. The `location` may be `local` or `distributed`, as for an overview phase. The `visibility` keyword indicates the visibility of individual participants' data, i.e. their documents defined under the `lists` subclause in the participant definition section. If this is `public`, all of these documents are available to all participants. If the keyword `private` is used, all such documents remain private to their owners. Participant definitions follow those already seen, with the further option of defining two extra roles: `reader` and `scribe`. The reader has overall control of the focus of an asynchronous meeting, while the scribe is tasked with completing documents. Finally, document definitions follow the same pattern as previously shown. Consolidation meetings have a similar structure to simple meetings, with the participants and documents present being defined.

As an alternative to a single meeting, two or more meetings to be held in parallel may be defined:

```
parallel 'Parallel'

    <...meeting definition 1...>

    <...meeting definition 2...>

    ...

    <...meeting definition n...>

end
```

Each meeting definition is simply a single meeting as previously shown. A given participant should not appear in more than one meeting definition of a parallel phase. This is not enforced by ASSIST, but is an obvious constraint that should be applied.

N Fold Detection Instead of a simple sequential detection stage, there may be a number of complete stages held concurrently, known as an N-Fold inspection. Each fold is a single detection phase surrounded by the keywords `fold` and `end`. The concurrent stages must be followed by at least one collation phase, where the results of the folds are brought together. The entire N-Fold stage is surrounded by the keywords `n_fold` and `end`:

```

n_fold
  fold 'Fold 1'

  <...detection definition...>

end
fold 'Fold 2'

<...detection definition...>

end

....

collation 'Collation'
  timing synchronous
  location local
  participants
    Coordinator
    Moderator1
    Moderator2
    Author
  roles
    Moderator1 is reader
    Moderator2 is scribe
  targets Code
  inputs
    Team_A_Defects
    Team_B_Defects
  outputs Master_Defect_List
end
end

```

The example shows a typical collation phase, with timing, location, participants, roles and documents clauses like those of simple meetings. An N-Fold inspection must have a single

participant defined to be a `coordinator`, who is the person in overall charge of the inspection. Each fold must have a single moderator.

Completion

Completion consists of three possible phases: rework, follow-up and exit. The phases must be defined in that order. Rework must have a single participant defined, along with targets, inputs and possible outputs. For example:

```
rework 'Rework'
  participant Author
  targets Code
  inputs
    Defects
    Design
end
```

Follow-up has much the same format, however targets, inputs and outputs must all be specified, along with a single participant.

```
follow_up 'Follow-up'
  participant Moderator
  targets Code
  inputs
    Defects
    Design
  outputs Follow_up_Report
end
```

Exit takes a form similar to that of entry. For example:

```
exit 'Exit'
  participant Moderator
  outputs Exit_criteria
end
```

B.4.4 Putting It All Together

The above demonstrate individual aspects of IPDL. To see how the declarations and process parts fit together, please refer to the supplied process definitions. Further help on writing

```

software_inspection ::= inspection inspection_name
                        declarations
                        process
                        end

inspection_name ::= string
declarations ::= declarations
                 document_declarations
                 responsibility_declarationsopt
                 participant_declarations
                 classification_declarationopt
                 end

process ::= process
            organisation_process
            detection_process
            completion_process
            end

organisation_process ::= entryopt planningopt overviewopt
detection_process ::= [detection|n_fold]
completion_process ::= reworkopt follow_upopt exitopt metricsopt
string ::= “’ character+ “’”
character ::= Any printable character or white space.

```

Figure B.22: Initial process definitions.

new processes can be found by reading the language reference (Section B.5) and the supplied process definitions.

B.5 IPDL Reference

The following sections describe the grammar of the inspection description language. It is described using a Backus-Naur style of notation. In this notation, a phrase in italics is non-terminal, while words in typewriter style indicate language keywords. The ‘`::=`’ operator is used to show expansion of non-terminal clauses. A plus indicates one or more instances of a given clause, while *opt* indicates that the clause is optional. Finally, square brackets indicate alternatives, with the alternatives separated by vertical bars.

B.5.1 Structure of Process Description

The description of a software inspection consists of two parts. The first part contains declarations listing the participants and their roles, along with the documents which will be used and created during the process. The second part describes the process itself, split into three stages. There should also be a facility for naming the inspection. The initial definition of the inspection is that given in Figure B.22. The keywords *inspection* and *end* are used to

```

document_declarations ::= documents document_definition+ end
document_definition ::= document_name document_type
document_name       ::= identifier
identifier         ::= non_whitespace_character+
non_whitespace_character ::= Any printable character which is not white space.
document_type      ::= [product|report|source|standard|
                        list|criteria|plan|detection.aid]

```

Figure B.23: Document definitions.

```

targets ::= targets document_name+
inputs  ::= inputs document_name+
outputs ::= outputs document_name+

```

Figure B.24: Document inclusion clauses.

delimit the description. *inspection name* is simply an arbitrary string surrounded by quotes. The declaration section consists of inspection documents, responsibilities, participants and classification scheme. Each of these is described in the following sections. The inspection process itself mirrors the process described earlier, consisting of the three major phases of organisation, detection and completion. The initial definitions of these are also presented and each will be described in more detail later.

B.5.2 Inspection Document, Participant and Responsibility Declarations

The first section within the declaration part of the description describes all documents which are available and created during the entire inspection, and is defined in Figure B.23. This section simply defines names for each of the documents to be used within the inspection. When the inspection is instantiated and run, part of the planning task is to associate the real inspection documents with each document defined. The format of such documents as criteria lists and reports is therefore left to the implementation.

The definition of each phase of the inspection will require the documents present and created during that phase to be defined. This will be achieved by the use of several clauses, defined in Figure B.24. Any document name appearing in these clauses must be declared within the document declaration section. The first of these introduces *target* documents, which may be of any type described above and are the actual documents being inspected. There is no constraint on the type since it is not unreasonable to allow standards, checklists and other supporting material to be inspected at the same time as the product. The *inputs* keyword indicates documents which are made available to this phase, and may also include documents of any type. The *outputs* keyword indicates the documents created or edited during the

```

responsibility_declarations ::= responsibilities responsibility_definition+ end
responsibility_definition ::= responsibility_name requires {document_name+}opt end
responsibility_name ::= identifier

```

Figure B.25: Responsibility definitions.

```

participant_declarations ::= participants participant_definition+ end
participant_definition ::= participant_name is
                             role
                             participant_listsopt
                             responsibility_assignmentopt
                             end
participant_name ::= identifier
participant_lists ::= lists document_name+
responsibility_assignment ::= responsibility responsibility_name+
role ::= [coordinator|moderator|author|inspector]

```

Figure B.26: Participant definitions.

phase, which may either be reports, plans, criteria lists, detection aids or lists. During each phase, participants will only have access to documents defined for that phase using these clauses.

The next part of the description is concerned with describing the participants involved with the process and the responsibilities which they may be assigned. A common inspection practice is to assign reviewers responsibility for certain defect types, thus hopefully improving the coverage and effectiveness of the inspection. This responsibility usually comes in the form of a checklist or other defect finding aid. Figure B.25 shows how a responsibility may be in terms of documents. Each responsibility has a name and a list of documents associated with it. These will usually be checklists or other detection aids, but may also include standards or any other document type. These documents should be made available to the appropriate participant by the support tool.

The definitions for inspection participants are shown in Figure B.26. There will usually be several constraints on the selection of participants. Typically, there must either be one moderator, *or* one coordinator and several moderators, depending on the type of inspection. This constraint is not part of the language because it may unnecessarily limit its flexibility. Instead, it is left to the implementation to enforce such restrictions as required. Zero or more authors may be declared, to allow maximum flexibility. Any number of inspectors may also be declared. The `lists` subclause indicates the document which this participant will use to record defects, change requests or other such items previously discussed when considering the list document type. Finally, the responsibility names used must be previously declared in the

```

classification_declaration ::= classification classification_name
classification_name       ::= string

```

Figure B.27: Item classification clauses.

responsibility declaration section above. Note that any single person may have more than one role or responsibility. During each phase, any participant with no defined responsibility will only be given access to documents defined in that phase, i.e. those generally available. One such possible document is a general checklist used by everyone.

The participants description is simply a list of people involved and the names of their roles. Note that the participant list is *not* a list of the real people involved; it simply lists the names of the `characters' in the inspection and the roles they will play. For example, a Fagan inspection may have:

```
Moderator is moderator
```

indicating that the person called Moderator is executing the moderator's duties. Contrast this with a Gilb and Graham-type inspection, where the person carrying out the task of the moderator is known as the leader:

```
Leader is moderator
```

This convention allows the naming of roles in any way required, allowing us to use terms which coincide with any inspection practice. This also allows many people to take the same role, for example to have multiple inspectors, each with a unique responsibility:

```

Inspector_MF is inspector
  lists MF_defects
  responsibility Missing_Functionality
end
Inspector_AM is inspector
  lists AM_defects
  responsibility Ambiguity
end

```

Figure B.27 shows the item classification clauses. These are used to optionally specify the classification scheme to be used for list items. The name of the classification scheme used must be known to ASSIST (e.g. `Fagan'). Classification names are not part of the language definition. See Section B.8.4 for details on adding new classification schemes to ASSIST.

```

participant ::= participant participant_name
participants ::= participants participant_name+

```

Figure B.28: Participant inclusion clauses.

```

entry ::= entry phase_name
          participant
          targetsopt
          inputsopt
          outputs
          end
phase_name ::= string

```

Figure B.29: Entry phase definition.

Finally, for each phase of the inspection the participants required to be present must be indicated. This is achieved with the two definitions shown in Figure B.28. The first definition indicates that only one participant should be present, while the second indicates the possibility of more than one person taking part. The use of these definitions will be shown along with each phase, but any participant name used within these clauses must have previously been declared in the participants declaration section.

B.5.3 The Organisation Process

The organisation stage may have three phases: entry, planning and overview. Figure B.22 shows the order in which these phases must occur, and indicates that the entry and overview phases are optional. Each of these phases is defined in turn, starting with the entry phase, shown in Figure B.29. This defines a name for the entry phase and indicates that only a single participant is required during this phase, usually either the moderator or the coordinator, depending on the type of inspection. At least one output document must be defined, usually a criteria list. A report detailing the outcome of the phase is also usually defined. Other documents may also be present using the targets and inputs keywords

The next phase defined is planning, shown in Figure B.30. Again, this phase may be

```

planning ::= planning phase_name
              participants
              targetsopt
              inputsopt
              outputs
              end

```

Figure B.30: Planning phase definition.

```

overview ::= overview phase_name
           location [local|distributed]
           participants
           presenter participant_name
           targets
           inputsopt
           outputsopt
           end

```

Figure B.31: Overview phase definition.

```

detection ::= [meeting_phase consolidation_stepopt]+
meeting_phase ::= [multi_meeting|single_meeting]
multi_meeting ::= parallel phase_name
                  single_meeting
                  single_meeting+
                  end
consolidation_step ::= consolidation meeting_phase

```

Figure B.32: Detection stage definition

named according to the method being described. Although planning will generally involve a single moderator, multiple participants must be allowed for, especially in the case of an N-Fold inspection, where the cooperation of several moderators and a coordinator may be required to form the inspection plan. In this case, the coordinator should have overall control over the planning stage, while the other participants can provide input. With multiple participants there must be either a single moderator or a single coordinator. Again, this constraint is left to the implementation. At least one output must be defined (usually a plan), but others outputs, along with targets and inputs, may be defined.

The final organisation activity is overview, shown in Figure B.31. This phase requires the definition of the participants involved, the format of the meeting, either local (same place) or distributed (different place). The presenter is the person who carries out the briefing; this is usually the author. The overview phase is optional.

B.5.4 The Detection Process

Detection activities consist of either a single detection activity or an N-Fold activity. This is shown in Figure B.22. A single detection activity was defined to consist of at least one meeting phase, possibly interspersed with consolidation steps. At this point the possibility of having several parallel meetings is also introduced to provide extra flexibility. This allows subsets of the team to meet separately. Consolidation steps consist of a consolidation meeting, where it is decided if a further meeting is required, and the meeting itself. The definition of *detection*

```

single_meeting ::= meeting phase_name
                   objective [examination|detection|collection]
                   timing [synchronous|asynchronous]
                   location [local|distributed]
                   visibility [public|private]
                   durationopt
                   participants
                   rolesopt
                   targets
                   inputsopt
                   outputsopt
                   end
duration       ::= duration integer

```

Figure B.33: Meeting phase definition.

```

roles           ::= roles role_assignment+
role_assignment ::= participant_name is meeting_role
meeting_role   ::= [reader|scribe]

```

Figure B.34: Role definition.

is shown in Figure B.32.

A meeting is defined to be a phase with one or more participants who may meet synchronously or asynchronously, and whose discussion may be private or public. The meeting may have one of three objectives: examination, defect detection, or defect collection. The assignment of roles during the meeting must also be allowed. Finally, the documents produced and used in the meeting must be defined. The definition of a meeting is shown in Figure B.33.

The definition starts with the keyword `meeting`, followed by the meeting name. The objective, timing, location and visibility are then set, along with the maximum duration of the meeting in minutes. The implementation should use the duration to help guide the moderator during the meeting. This is followed by a list of all meeting participants, as defined earlier. The roles of reader and scribe may be assigned. If no reader is specified, then it is assumed that any participant can guide the meeting (such as in a Humphrey-type inspection where the document is not paraphrased). If the scribe is not specified then the moderator should be given that role by default. The roles are followed by target documents, inputs from previous phases (such as lists) and outputs generated during this meeting (such as reports). All documents are optional except for target documents. The role assignment section is defined in Figure B.34. Only the roles of `Reader` and `Scribe` are defined.

The consolidation phase may follow any meeting, and is used to decide on the need for a further meeting to resolve any remaining issues. The definition is shown in Figure B.35.


```

consolidation ::= consolidation phase_name
                  participant
                  targets
                  inputs
                  outputs
                  end

```

Figure B.35: Consolidation phase definition.

```

n_fold ::= n_fold phase_name
            fold
            fold+
            collation+
            end
fold ::= fold phase_name
          detection
          end

```

Figure B.36: N-Fold stage definition.

Again, the phase may be named, and this is followed by the single participant who will perform the consolidation (usually the moderator). The target documents and input documents to this phase are then specified, which generally consist of the product and one or more lists, respectively. Finally, at least one output must be defined: this is usually a report.

The alternative to a single detection activity is to have multiple, parallel detection activities with a collation stage, i.e. N-Fold inspection. To increase flexibility, there is the possibility of holding more than one collation meeting. The definition is given in Figure B.36. As usual, the phase may be named. The definition will then consist of two or more detection activity definitions, as described above, surrounded by the keywords `fold` and `end`, along with one or more collation meeting definitions.

The collation meeting definition is shown in Figure B.37. For each collation, a number of participants can be listed, usually several moderators along with the coordinator, one of whom must be nominated scribe with a role definition, another of whom may be nominated

```

collation ::= collation phase_name
               timing [synchronous|asynchronous]
               location [local|distributed]
               participants
               roles
               targets
               inputs
               outputs
               end

```

Figure B.37: Collation meeting definition.

```

rework ::= rework phase_name
           participant
           targets
           inputs
           outputsopt
           end

```

Figure B.38: Rework phase definition.

```

follow_up ::= follow_up phase_name
               participant
               targets
               inputs
               outputs
               end

```

Figure B.39: Follow-up phase definition.

reader. Inputs will generally consist of a collected list of defects from each inspection. The output will usually be a single master list of defects for the entire inspection, but reports may also form outputs from this phase. Several collation meetings may take place, to allow for the possibility of the coordinator meeting with a subgroup of moderators. In this case, an input to subsequent meetings should be the collated lists of defects from previous meetings.

B.5.5 The Completion Process

The completion process consists of four activities, as shown in Figure B.22: rework, follow-up, exit and metrics, all of which are optional.

The rework phase is defined in Figure B.38. Although rework is generally carried out by the author, the possibility of another participant performing rework is catered for. This may occur if the author is not part of the inspection team, or is otherwise unavailable. Various documents may be made available during this phase. Target documents are always required, with the implementation having to provide some means of editing these documents. Input documents will typically consist of one or more lists. The output of the phase may consist of one or more reports, or other documents as required.

The next phase is follow-up, involving checking the work performed in rework, and is defined in Figure B.39. Only one person should perform follow-up: this is usually the moderator (or coordinator), but there is the possibility of another participant performing this task. A target document is always required, and other input documents (usually a list of defects) must also be present. Finally, the defined output is one or more reports.

Next is the optional exit phase, defined in Figure B.40. This is similar to the entry phase

```

exit ::= exit phase_name
      participant
      targets_opt
      inputs_opt
      outputs
      end

```

Figure B.40: Exit phase definition.

```

metrics ::= metrics phase_name
         participant
         data
         targets_opt
         inputs_opt
         outputs
         end
data     ::= data measure+
measure  ::= identifier participant_name_opt phase_name_opt

```

Figure B.41: Metrics collection phase definition.

in that it defines one or more output documents, usually lists of criteria which must be met. A report detailing the outcome of the phase may also be appropriate. Input and target documents may also be defined. One single participant is involved in this phase: this is either the moderator or the coordinator, depending on the inspection type.

Finally, the metrics collection and analysis phase is shown in Figure B.41. This follows the format of other phases. The main difference is the `data` subclause. This is used to indicate the measures which must be supplied by the tool for this phase. Each measure consists of its name, an optional participant name for whom this measure applies, and an optional phase name which states which phase that particular measure is to be taken from. For example, to collect the number of list items produced by the participant `Moderator` during the phase `'Preparation'`, the following might be used:

```

data
  list_items Moderator 'Preparation'

```

Other metrics, such as the length of the product, are not specific to a single phase or a single participant and do not require these to be specified. No measures are defined in IPDL. Details of measures available in ASSIST can be found in Section B.7.

ASSIST is supplied with eight standard process definitions. When defining a new process, it is usually easier to edit an existing process which is similar to that required. The example processes provide demonstrations of the main IPDL constructs.

```

criteria_list ::= criteria_section+
criteria_section ::= heading criterion+
criterion ::= [subheading|multi|open|numeric|date|check]

```

Figure B.42: The format of a criteria list.

```

report ::= report_section+
report_section ::= heading report_item+
report_item ::= [subheading|multi|open|numeric|date|check]

```

Figure B.43: The format of a report.

B.6 Formats

B.6.1 Checklists, Criteria, Reports and Plans

The criteria, report, plan and checklist browsers supplied with ASSIST require documents to be in a specific format. This format allows different types of items within the document to be easily specified. Figure B.42 shows the format of a criteria document. Each document consists of a number of sections, each of which consists of a heading plus one or more individual criteria items. In the same manner, reports consist of a number of sections each of which contains a heading plus one or more items (Figure B.43). The different item type are identical to those of criteria documents. As can be seen from Figure B.44, plans have a similar format. The format for checklist is also similar, as shown in Figure B.45. The different item type are identical for criteria documents, reports, plans and checklists. They are shown in Figure B.46, along with the definition of a heading.

A heading is simply the word `heading` followed by a string. The four item types are `multi`, `open`, `numeric` and `date`. A `multi` is a multiple choice question, consisting of the question itself plus two or more responses. An `open` question allows a freeform textual answer to be given. This requires a definition of the question and the maximum length of the answer “box” required. A `numeric` question is used when the answer is expected to be

```

plan ::= plan_section+
plan_section ::= heading plan_item+
plan_item ::= [subheading|multi|open|numeric|date|check]

```

Figure B.44: The format of a plan.

```

checklist ::= checklist_section+
checklist_section ::= heading checklist_item+
checklist_item ::= [subheading|multi|open|numeric|date|check]

```

Figure B.45: The format of a checklist.

```

heading ::= heading string keywordopt
subheading ::= subheading string keywordopt
multi ::= multi question response response+ text_answeropt keywordopt
open ::= open question length text_answeropt keywordopt
numeric ::= numeric question length unitopt numeric_answeropt keywordopt
check ::= check question check_answeropt keywordopt
date ::= date question date_answeropt keywordopt
length ::= length integer
unit ::= unit string
text_answer ::= answer string
numeric_answer ::= answer integer
date_answer ::= answer DD"/"MM"/"YYYY
check_answer ::= answer [yes|no]
question ::= string
response ::= string
string ::= "' character+ "'
character ::= Any printable character or white space.
integer ::= Any standard integer.

```

Figure B.46: Items common to criteria lists, reports, plans and checklists.

Figure B.47: An example of a criteria list.

an integer. Again, the question must be specified along with the maximum answer length. Furthermore, the units following the answer may be specified. A date question requires a date in DD/MM/YYYY format as answer. This simply requires the question to be specified. Similarly, a check item, which requires a binary answer, only requires the question to be specified.

Each item may have a “correct” answer associated with it. This answer may be used for checking that the document has been completed correctly. The answer is indicated by the keyword `answer` followed by the answer itself, in the appropriate format. Finally, each item may have a keyword associated with it. This keyword is used by the automatic cross-referencer within ASSIST, and allows document features to be associated with specific items. For example, all `for` loops in a C++ file may be linked to a checklist item which deals specifically with `for` loops.

An example of a criteria list is:

```
heading 'Criteria Example'
check 'Code passed static analysis?' answer 'yes'
open 'Authors name' length 30
```

```

numeric 'Estimated defects remaining'
      length 3 unit 'defects/KLOC'
date 'Scheduled inspection completion date'
```

The document produced by the above file is shown in Figure B.47. Reports, plans and check-lists have similar layouts.

B.6.2 Help Documents

The help browser supplied with ASSIST requires documents to be in a specific format. This format uses a subset of HTML tags. Tags are used to divide the document into pages, add formatting, and allow internal and external cross-referencing. A document begins with its title followed by one or more pages. Each page consists of a heading (with the header tag indicating the section level of this page), the text of the page, a list of internal cross-references, and a list of keywords which the page may be cross-referenced on.

The following tags are used:

- `<title>...</title>` are used to surround the title of the document. This tag must be the first item in the document.
- `<hn>...</hn>` (where $n = 1..6$) denote sections and subsections. `h1` is a top-level section, while `h2` to `h6` are progressively lower subsections.
- `<pre>...</pre>` are used to surround text which is already formatted and should not be formatted by the browser.
- `<p>` takes a new paragraph in the document.
- `<see_also>...</see_also>` indicate internal cross-references. To add a reference to another page, place (one of) its keywords here.
- `<keywords>...</keywords>` gives a list of keywords for this document. Keywords can be referenced manually using the `<see_also>` tag. ASSIST will also automatically use these keywords when building cross-references to other documents.

To demonstrate the help format, here is a sample of the C++ reference supplied with ASSIST.

```

<title> C++ Reference </title>

<h1>Constructs</h1>
```

<h2>The for loop</h2>

The for loop has the general form

<p>

<pre>

```
    for (expression1; expression2; expression3)
        statement
```

</pre>

<p>

First, expression1 is evaluated, and typically contains an initialisation expression. Then expression2 is evaluated. If it is non-zero, statement is executed, expression3 is evaluated and control passes back to the beginning of the for loop, except that expression1 is not evaluated again. This cycle continues until expression2 is zero, when control passes to the following statement.

<see_also>cpp_keyword_break cpp_keyword_continue</see_also>

<keywords>cpp_keyword_for</keywords>

<h2>The while loop</h2>

The while statement has the form

<p>

<pre>

```
    while(expression)
        statement
```

</pre>

<p>

First, expression is evaluated. If it is non-zero, statement is executed and control returns to the start of the while loop. Therefore, the body of the while loop is executed until expression becomes zero. The body can be executed zero or

more times.

```
<see_also>cpp_keyword_break cpp_keyword_continue</see_also>  
<keywords>cpp_keyword_while</keywords>
```

The C++ reference is shown in Figure B.19.

B.7 Metrics Available in ASSIST

No metrics are currently available.

B.8 Customising and Extending ASSIST

B.8.1 Altering the Printer Setup

As standard, ASSIST uses the `lpr` command for printing. You can alter this in the setup file

```
$ASSIST_HOME/client/assist_defs.py
```

The line

```
PRINT = 'lpr'
```

indicates the print command to be used. Simply substitute your required print command here. For example, to print on a specific printer, use

```
PRINT = 'lpr -P<printer-name>'
```

B.8.2 The `.assistrc` file

The `.assistrc` file allows the user to customise certain aspects of ASSIST. This file is automatically created the first time you run ASSIST, and is consulted each time thereafter.

Distributed Support

The tools used to provide distributed support are by default always on. They can be turned off by altering the relevant lines in the `.assistrc` file. Replacing the '1' with a '0' in each of the following lines turns the relevant tool off:

```
whiteboard      1
video           1
audio           1
text            1
```

B.8.3 Adding New Browsers

ASSIST can be extended by adding new document browsers. This can either be written directly in Python, or browsers in other languages can be added by providing a Python interface. This section describes the browser interface specification and how to add a new browser to ASSIST. It assumes a detailed knowledge of the Python language.

Browser Interface Definition

The browser must be written in the following basic form:

```
from assist_defs import *
from Tkinter import *

class Browser(Toplevel):

    def __init__(self, DocumentName, Filename, ListItems,
                 Finished, NewAnnotation, ShowAnnotation,
                 DeleteAnnotation, ProposeAnnotation,
                 GetReferences, JumpToReference,
                 DocumentChecked, Coverage, SecondaryData,
                 Reader, BroadcastJump, Save, Objective,
                 Timing, WriteDocument):
```

The parameters passed to the `__init__` function are:

- `DocumentName` - the name of the document which the browser has to display (string).
- `Filename` - the filename under which the document can be found for loading (string).
- `ListItems` - contains details of the annotations for this document. It is a dictionary using the positions of the annotations as keys. Each entry consists of a list of one or more objects, each of which is an entire `ListItem`. These items have the following attributes:

- `Filename` - the name of the document which this item refers to (string)
 - `Position` - the position of the item within the document (document specific, but always a string)
 - `Title` - title of item (string)
 - `Text` - textual description (string)
 - `Type` - first classification (string)
 - `Class` - second classification (string)
 - `Severity` - third classification (string)
 - `ID` - ID of item, unique within the list to which the item belongs (integer)
 - `Time` - time of creation of item (string)
 - `Owner` - username of the person who created this item (string)
 - `Implemented` - has value 0 if item has been declared implemented, otherwise has the value 0
 - `Accept` - list of users who have voted to accept this item
 - `Reject` - list of users who have voted to reject this item
- `Finished` - the function to be called when the browser is closed, allowing the system to update its status. See the `Close` function, described later.
 - `NewAnnotation` - a function passed to the browser to allow annotation of documents via the list browser. The call has the following form:

```
NewAnnotation(Title, Position, File, Text, Type, Class,
              Severity)
```

`Title` is the title of the annotation. `Position` is the position within the document which this annotation refers to. `File` is the name of the document which the annotation refers to. `Text` is the actual text of the annotation. The remaining three parameters are three strings used as to classify the item. Any parameter not required must be replaced by an empty string.

- `ShowAnnotation` - a function passed to the browser to allow annotations to be viewed and updated via the list browser. The call has the following form:

```
ShowAnnotation(List, ListItemID)
```

List is the name of the list in which the annotation occurs. ListItemID is the ID of the annotation which is being shown. Both parameters must be present.

- DeleteAnnotation - a function passed to the browser to allow annotations to be deleted via the list browser. The call has the following form:

```
DeleteAnnotation(List, ListItemID)
```

List is the name of the list in which the annotation occurs. ListItemID is the ID of the annotation which is being updated. Both parameters must be present.

- ProposeAnnotation - a function passed to the browser to allow annotations to be proposed via the list browser. The call has the following form:

```
ProposeAnnotation(List, ListItemID)
```

List is the name of the list in which the annotation occurs. ListItemID is the ID of the annotation which is being updated. Both parameters must be present.

- GetReferences - allows the browser to make queries concerning cross-references. Its form is:

```
GetReferences(Keyword)
```

Keyword is the reference term to be looked up. The function returns a list of references. Each reference is itself a list, the first item of which is the name of the document in which the reference may be found, the second is the position of the reference within that document, and the third contains the word being referenced.

- JumpToReference - allows this browser to cross-reference with other browsers. Calling this function make ASSIST jump to the indicated reference. If the required browser is not open, ASSIST will open it.

```
JumpToReferences(Reference, Keyword)
```

Keyword is the index term, while Reference is a list containing the document name and the position which should be highlighted.

- DocumentChecked - used to inform the system that the document has been satisfactorily completed, if such completion is required.

`DocumentChecked(DocumentName)`

`DocumentName` is the name of the document which has been checked.

- `Coverage` - coverage of the document achieved so far. This may take any form required. For example, the text browser uses a list of line numbers to indicate which lines have been inspected.
- `SecondaryData` - document and browser specific data generated at the start of the inspection. See the `GenerateSecondaryData` function described later for more details.
- `Reader` - has the value 1 if this participant has the role of reader during this phase, otherwise has the value 0.
- `BroadcastJump` - a function which is passed to allow the propagation of current focus if the browser is being used by the reader during a synchronous phase.

`BroadcastJump(Position, Document)`

`Position` is the position which is to be moved to. `Document` is the name of the document.

- `Save` - a function used to save the document if it has been edited. The call has the following form:

`Save(DocumentName, Contents)`

`DocumentName` is the name of the document, as passed to the browser. `Contents` is the contents of the document.

- `Objective` - the objective of the meeting, one of three constants: `EXAMINATION`, `DETECTION` or `COLLECTION`.
- `Timing` - the timing of the meeting, either `SYNCHRONOUS` or `ASYNCHRONOUS`.
- `WriteDocument` - has the value 1 if this document can be edited, otherwise has the value 0.

The class must provide a number of functions. If the the function is not implemented, its header must still be present and the body should consist of a `pass` statement. The functions are:

- `Close(self)` - the function called when the browser is closed. This must make a call to the `Finished` function passed to the browser. This call is usually

```
self.Finished(self.DocumentName, self.Covered)
```

It passes the document name and the updated coverage (if any).

- `JumpTo(self, Position, Keyword = None)` - Used to move the focus of the browser to a certain position. The position is given by the `Position` parameter, while the `Keyword` parameter can be used to indicate a specific reference of interest.
- `RepositionBrowser(self, List, ListItem)` Repositions the browser at the item indicated. The contents of `ListItem` must be examined to find the appropriate position.
- `BrowserAddAnnotation(self, List, ListItem)` - informs the browser of any new items (annotations) added to this document. The parameters are the list to which the item belongs (a string), and the item itself.
- `BrowserDeleteAnnotation(self, List, ListItem)` - tells the browser when an item (annotation) is deleted. The list to which the item belongs (a string), and the item itself are passed as parameters.
- `ItemVotedOn(self, List, ListItem, User, Vote)` - tells the browser when an item has been voted on. `List` is the name of the list which the item belongs to. `ListItem` is the item which has been voted on. `User` is a string containing the name of the person casting the vote. `Vote` is the vote itself.
- `deiconify(self)` This function must be defined if the browser is not Tk based. This function should open the browser from its iconified state, if it has one. Alternatively, the body can simply consist of a `pass` statement. If the browser is Tk based, this function is automatically defined - do not override it.
- `tkraise(self)` This function must be defined if the browser is not Tk based. This function should move the browser above any other windows which may be on the screen. Alternatively, the body can simply consist of a `pass` statement. If the browser is Tk based, this function is automatically defined - do not override it.

In addition, two other functions must be defined at the module level. These are called when an inspection is started, for each document which makes use of the browser. The first

generates cross-references for the document, while the second is a non-specific function which can be used to generate other data about that document.

- `GenerateXRefs(Filename, DocumentName)` generates cross-references for the document called `DocumentName` which has been stored under `Filename`. This function may return two items, held in a list. The first item in the list is a set of references, stored as a Python dictionary. The keys of this dictionary are the reference terms. Each item in this dictionary must consist of a list of reference items. A reference item is a list containing the name of the document, the position of the reference and a title for the reference. If no references are generated, an empty dictionary should be returned as the first element of the list. The second item can be used to return any other document specific data, such as the output of a static analysis tool. This avoids running such tools multiple times for the same document during a single inspection. No format is defined for this data. If no data is to be generated, `None` should be returned as the second element of the list.

A template which you can use as a starting point for writing your own browser can be found in `$ASSIST_HOME/client/browser_template.py`. See also the supplied browsers.

Adding New Content Types

When a new browser is added to ASSIST, it must have an entry in one of the content-type files to allow it to be used. The directory `$ASSIST_HOME/server_data/content-types` contains a number of file specifying content types for a subset of the document types. Each file contains entries of the form

```
<content-type> <browser-name>
```

For example, the content-types file for products has the following by default:

```
ASCII tbrowser  
code codebrowser  
C++ cppbrowser
```

When a document is added to the document database (Section B.2.2), this list of content types is displayed under the **Select Content Type** menu. During an inspection, the content type of the document is used to determine the actual browser used to display that document. The browser name must match the name of the Python file containing the code for that browser (minus the `.py` extension).

B.8.4 Adding New Classification Schemes

New item classification schemes can be easily added to ASSIST. Each scheme may have up to three classification levels. These are referred to within ASSIST as type, class and severity, but may be given any name desired. `$ASSIST_HOME/server_data/classifications` contains all the data on classification schemes. Within this directory there is a directory for each level of classification: `types`, `classes` and `severities`. To add a new classification scheme you must create a file in one or more of these directories. This file should have the same name as that of your classification scheme. The first line of the file should contain the name of this classification level. This should be followed by the names of the different categories, one per line. For example, the Fagan-type classification scheme which comes with ASSIST has a file called `fagan` in each subdirectory `types`, `classes` and `severities`. The file in `classes` is

```
Class
Missing
Wrong
Extra
```

`Class` is the name of this classification level, and it has three categories. To use this classification scheme in ASSIST, the IPDL line

```
classification 'fagan'
```

is used. If less than three classification levels are used, ASSIST will automatically disable the others.

Appendix C

IPDL Processes

C.1 Fagan Inspection

```
inspection 'Fagan Code Inspection'  
  declarations  
    documents  
      Code          product  
      Design        source  
      Defects1      list  
      Defects2      list  
      Defects3      list  
      Defects4      list  
      Defects5      list  
      Master_defects list  
      Meeting_report report  
      Follow_up_Report report  
      Master_Plan   plan  
    end  
  participants  
    Inspector1 is inspector  
      lists Defects1 end  
    Inspector2 is inspector  
      lists Defects2 end  
    Inspector3 is inspector  
      lists Defects3 end  
    Moderator is moderator  
      lists Defects4 end  
    Author is author  
      lists Defects5 end  
  end  
  classification 'fagan'  
end  
process
```

```
planning 'Planning'
  participants Moderator
  outputs Master_Plan
end
overview 'Overview'
  location local
  participants
    Moderator
    Author
    Inspector1
    Inspector2
    Inspector3
  presenter Author
  targets Code
end
meeting 'Preparation'
  objective examination
  timing asynchronous
  location local
  visibility private
  participants
    Moderator
    Author
    Inspector1
    Inspector2
    Inspector3
  targets Code
  inputs Design
end
meeting 'Inspection'
  objective collection
  timing synchronous
  location local
  visibility public
  participants
    Moderator
    Author
    Inspector1
    Inspector2
    Inspector3
  roles
    Inspector1 is reader
    Inspector2 is scribe
  targets Code
  inputs Design
  outputs
    Master_defects
    Meeting_report
end
rework 'Rework'
```

```

        participant Author
        targets Code
        inputs
            Master_defects
            Design
    end
    follow_up 'Follow-up'
        participant Moderator
        targets Code
        inputs
            Master_defects
            Design
        outputs Follow_up_Report
    end
end
end
end

```

C.2 Structured Walkthrough

```

inspection 'Structured Walkthrough'
    declarations
        documents
            Code          source
            Design        source
            Error_List1   list
            Error_List2   list
            Error_List3   list
            Error_List4   list
            Error_List5   list
            Master_Error_List list
            Maintenance_Requirements source
            Coding_Standard standard
            User_Requirements source
            Summary       report
            Follow_up_Report report
            Master_Plan   plan
        end
        responsibilities
            Maintenance requires
                Maintenance_Requirements
            end
            Standards requires
                Coding_Standard
            end
            User requires
                User_Requirements
            end
        end
    end
participants

```

```

Coordinator is moderator
    lists Error_List1
end
Producer is author
    lists Error_List2
end
Maintenance_Oracle is inspector
    lists Error_List3
    responsibility Maintenance
end
Standards_Bearer is inspector
    lists Error_List4
    responsibility Standards
end
User_Representative is inspector
    lists Error_List5
    responsibility User
end
Reviewer is inspector
    lists Error_List5
end
end
end
process
    planning 'Planning'
        participants Coordinator
        outputs Master_Plan
    end
    meeting 'Preparation'
        objective examination
        timing asynchronous
        location local
        visibility private
        participants
            Coordinator
            Producer
            Maintenance_Oracle
            Standards_Bearer
            User_Representative
            Reviewer
        targets Code
        inputs Design
    end
    meeting 'Walkthrough'
        objective collection
        timing synchronous
        location local
        visibility public
        duration 60
        participants

```

```

        Coordinator
        Producer
        Maintenance_Oracle
        Standards_Bearer
        User_Representative
        Reviewer
    roles
        Producer is reader
        Reviewer is scribe
    targets Code
    inputs Design
    outputs
        Master_Error_List
        Summary
    end
    rework 'Rework'
        participant Producer
        targets Code
        inputs
            Master_Error_List
            Design
    end
    follow_up 'Follow-up'
        participant Coordinator
        targets Code
        inputs
            Master_Error_List
            Design
        outputs Follow_up_Report
    end
end
end
end

```

C.3 Humphrey Inspection Process

```

inspection 'Humphrey Inspection Process'
    declarations
        documents
            Code                product
            Design              source
            Error_Log1          list
            Error_Log2          list
            Error_Log3          list
            Error_Log4          list
            Error_Log5          list
            Consolidated_Error_Log list
            Master_Error_Log    list
            Meeting_report      report
            Follow_up_Report    report

```

```

        Master_Plan          plan
        Entry_criteria       criteria
        Exit_criteria        criteria
    end
    participants
        Reviewer1 is inspector
            lists Error_Log1 end
        Reviewer2 is inspector
            lists Error_Log2 end
        Reviewer3 is inspector
            lists Error_Log3 end
        Moderator is moderator
            lists Error_Log4 end
        Producer is author
            lists Error_Log5 end
    end
end
process
    planning 'Planning'
        participants Moderator
        outputs Master_Plan
    end
    overview 'Overview'
        location local
        participants
            Moderator
            Producer
            Reviewer1
            Reviewer2
            Reviewer3
        presenter Producer
        targets Code
    end
    meeting 'Preparation'
        objective detection
        timing asynchronous
        location local
        visibility private
        participants
            Moderator
            Producer
            Reviewer1
            Reviewer2
            Reviewer3
        targets Code
        inputs Design
    end
    meeting 'Analysis'
        objective collection
        timing asynchronous

```

```
        location local
        visibility private
        participants
            Producer
        targets Code
        outputs Consolidated_Error_Log
    end
    meeting 'Inspection'
        objective collection
        timing synchronous
        location local
        visibility public
        participants
            Moderator
            Producer
            Reviewer1
            Reviewer2
            Reviewer3
        roles
            Reviewer1 is reader
            Reviewer2 is scribe
        targets Code
        inputs
            Consolidated_Error_Log
            Design
        outputs
            Master_Error_Log
            Meeting_report
    end
    rework 'Rework'
        participant Producer
        targets Code
        inputs
            Master_Error_Log
            Design
    end
    follow_up 'Follow-up'
        participant Moderator
        targets Code
        inputs
            Master_Error_Log
            Design
        outputs Follow_up_Report
    end
end
end
```

C.4 Gilb and Graham

```

inspection 'Gilb and Graham Inspection'
  declarations
    documents
      Code                product
      Design              source
      Issue_List1         list
      Issue_List2         list
      Issue_List3         list
      Issue_List4         list
      Issue_List5         list
      Issue_Log           list
      Process_Improvement_Log list
      Meeting_report      report
      Follow_up_Report    report
      Master_Plan         plan
      Entry_criteria      criteria
      Exit_criteria       criteria
    end
  participants
    Checker1 is inspector
      lists Issue_List1 end
    Checker2 is inspector
      lists Issue_List2 end
    Checker3 is inspector
      lists Issue_List3 end
    Leader is moderator
      lists Issue_List4 end
    Author is author
      lists Issue_List5 end
  end
end
process
  entry 'Entry'
    participant Leader
    outputs Entry_criteria
  end
  planning 'Planning'
    participants Leader
    outputs Master_Plan
  end
  overview 'Kickoff'
    location local
    participants
      Leader
      Author
      Checker1
      Checker2
      Checker3

```



```
    presenter Author
    targets Code
end
meeting 'Checking'
  objective detection
  timing asynchronous
  location local
  visibility private
  participants
    Leader
    Author
    Checker1
    Checker2
    Checker3
  targets Code
  inputs Design
end
meeting 'Logging'
  objective collection
  timing synchronous
  location local
  visibility public
  participants
    Leader
    Author
    Checker1
    Checker2
    Checker3
  roles
    Checker1 is reader
    Checker2 is scribe
  targets Code
  inputs Design
  outputs
    Issue_Log
    Meeting_report
end
meeting 'Process Brainstorming'
  objective collection
  timing synchronous
  location local
  visibility public
  participants
    Leader
    Author
    Checker1
    Checker2
    Checker3
  roles
    Checker2 is scribe
```

```

        targets
            Code
            Design
        inputs Design
        outputs
            Issue_Log
            Meeting_report
    end
    rework 'Edit'
        participant Author
        targets Code
        inputs
            Issue_Log
            Design
    end
    follow_up 'Follow-up'
        participant Leader
        targets Code
        inputs
            Issue_Log
            Design
        outputs Follow_up_Report
    end
    exit 'Exit'
        participant Leader
        outputs Exit_criteria
    end
end
end
end

```

C.5 Asynchronous Inspection

```

inspection 'Asynchronous Inspection'
    declarations
        documents
            Code                product
            Design              source
            Issues1             list
            Comments1           list
            Actions1            list
            Issues2             list
            Comments2           list
            Actions2            list
            Issues3             list
            Comments3           list
            Actions3            list
            Issues4             list
            Comments4           list
            Actions4            list
        end
    end
end

```

```

        Consolidated_Issues    list
        Consolidated_Comments list
        Consolidated_Actions  list
        Unresolved_Issues     list
        Further_Issues        list
        Consolidation_report   report
        Follow_up_Report      report
        Master_Plan           plan
    end
    participants
        Moderator is moderator
            lists
                Issues1
                Comments1
                Actions1
        end
        Producer is author
            lists
                Issues2
                Comments2
                Actions2
        end
        Reviewer1 is inspector
            lists
                Issues3
                Comments3
                Actions3
        end
        Reviewer2 is inspector
            lists
                Issues4
                Comments4
                Actions4
        end
    end
end
process
    planning 'Setup'
        participants Moderator
        outputs Master_Plan
    end
    overview 'Orientation'
        location local
        participants
            Moderator
            Producer
            Reviewer1
            Reviewer2
        presenter Producer
        targets Code

```

```
end
meeting 'Private Review'
  objective detection
  timing asynchronous
  location local
  visibility private
  participants
    Moderator
    Producer
    Reviewer1
    Reviewer2
  targets Code
  inputs Design
end
meeting 'Public Review'
  objective collection
  timing asynchronous
  location local
  visibility public
  participants
    Moderator
    Producer
    Reviewer1
    Reviewer2
  roles
    Reviewer1 is reader
    Reviewer2 is scribe
  targets Code
  inputs Design
end
consolidation 'Consolidation'
  participant
    Moderator
  targets Code
  inputs
    Issues1
    Comments1
    Actions1
    Issues2
    Comments2
    Actions2
    Issues3
    Comments3
    Actions3
    Issues4
    Comments4
    Actions4
    Design
  outputs
    Consolidated_Issues
```

```

        Consolidated_Comments
        Consolidated_Actions
        Unresolved_Issues
        Consolidation_report
    end
meeting 'Group Review Meeting'
    objective collection
    timing synchronous
    location local
    visibility public
    participants
        Moderator
        Producer
        Reviewer1
        Reviewer2
    roles
        Reviewer1 is reader
        Reviewer2 is scribe
    targets Code
    inputs
        Unresolved_Issues
        Design
    outputs
        Further_Issues
end
rework 'Rework'
    participant Producer
    targets Code
    inputs
        Consolidated_Issues
        Consolidated_Comments
        Consolidated_Actions
        Further_Issues
        Design
end
follow_up 'Conclusion'
    participant Moderator
    targets Code
    inputs
        Consolidated_Issues
        Consolidated_Comments
        Consolidated_Actions
        Further_Issues
        Design
    outputs Follow_up_Report
end
end
end

```

C.6 Active Design Reviews

```

inspection 'Active Design Review'
  declarations
    documents
      Design          product
      AV_Questionnaire detection_aid
      AS_Questionnaire detection_aid
      C_Questionnaire detection_aid
      Defects1        list
      Defects2        list
      Defects3        list
      Follow_up_Report report
      Master_Plan     plan
    end
  responsibilities
    Assumption_Validity requires
      AV_Questionnaire
    end
    Assumption_Sufficiency requires
      AS_Questionnaire
    end
    Consistency requires
      C_Questionnaire
    end
  end
  participants
    Reviewer1 is inspector
      lists Defects1
      responsibility Assumption_Validity
    end
    Reviewer2 is inspector
      lists Defects2
      responsibility Assumption_Sufficiency
    end
    Reviewer3 is inspector
      lists Defects3
      responsibility Consistency
    end
    Designer1 is moderator
    end
    Designer2 is author
    end
  end
end
process
  planning 'Planning'
    participants Designer1
    outputs Master_Plan
  end

```

```
overview 'Overview'  
  location local  
  participants  
    Designer1  
    Designer2  
    Reviewer1  
    Reviewer2  
    Reviewer3  
  presenter Designer2  
  targets Design  
end  
meeting 'Review'  
  objective detection  
  timing asynchronous  
  location local  
  visibility private  
  participants  
    Reviewer1  
    Reviewer2  
    Reviewer3  
  targets Design  
end  
meeting 'Discussion 1'  
  objective collection  
  timing synchronous  
  location local  
  visibility public  
  participants  
    Reviewer1  
    Designer1  
  roles  
    Designer1 is scribe  
  targets Design  
  inputs AV_Questionnaire  
  outputs Defects1  
end  
meeting 'Discussion 2'  
  objective collection  
  timing synchronous  
  location local  
  visibility public  
  participants  
    Reviewer2  
    Designer2  
  roles  
    Designer2 is scribe  
  targets Design  
  inputs AS_Questionnaire  
  outputs Defects2  
end
```

```

meeting 'Discussion 3'
  objective collection
  timing synchronous
  location local
  visibility public
  participants
    Reviewer3
    Designer1
  roles
    Designer1 is scribe
  targets Design
  inputs C_Questionnaire
  outputs Defects3
end
rework 'Rework'
  participant Designer2
  targets Design
  inputs
    Defects1
    Defects2
    Defects3
end
follow_up 'Follow-up'
  participant Designer1
  targets Design
  inputs Defects1 Defects2 Defects3
  outputs Follow_up_Report
end
end
end

```

C.7 Phased Inspection

```

inspection 'Phased Inspection'
  declarations
    documents
      Code                product
      Design              source
      Defect_List1       list
      Defect_List2       list
      Defect_List3       list
      Defect_List4       list
      Question_List1     list
      Question_List2     list
      Question_List3     list
      Collected_Defect_List list
      Coding_Standard    standard
      Coding_Standard_Checklist detection_aid
      Reusability_Checklist detection_aid

```



```

        Master_Plan          plan
        Follow_up_Report    report
    end
    responsibilities
        Standards_Compliance requires
            Coding_Standard Coding_Standard_Checklist
        end
        Reusability requires
            Reusability_Checklist
        end
    end
    participants
        Moderator is moderator end
        Author is inspector end
        Inspector1 is inspector
            lists Defect_List1
            responsibility Standards_Compliance
        end
        Inspector2 is inspector
            lists Defect_List2 Question_List1
            responsibility Reusability
        end
        Inspector3 is inspector
            lists Defect_List3 Question_List2
            responsibility Reusability
        end
        Inspector4 is inspector
            lists Defect_List4 Question_List3
            responsibility Reusability
        end
    end
end
end
process
    planning 'Planning'
        participants Moderator
        outputs Master_Plan
    end
    meeting 'Phase 1'
        objective detection
        timing asynchronous
        location local
        visibility private
        participants
            Inspector1
        targets Code
        inputs Design
    end
    meeting 'Phase 2 Examination'
        objective examination
        timing asynchronous

```

```
location local
visibility private
participants
    Inspector2
    Inspector3
    Inspector4
targets Code
inputs Design
end
meeting 'Phase 2 Inspection'
objective detection
timing asynchronous
location local
visibility private
participants
    Inspector2
    Inspector3
    Inspector4
targets Code
inputs Design
end
meeting 'Phase 2 Reconciliation'
objective collection
timing synchronous
location local
visibility public
participants
    Inspector2
    Inspector3
    Inspector4
targets Code
inputs
    Defect_List2
    Defect_List3
    Defect_List4
    Design
outputs
    Collected_Defect_List
end
rework 'Rework'
participant Author
targets Code
inputs
    Collected_Defect_List
    Defect_List1
    Design
end
follow_up 'Follow-up'
participant Moderator
targets Code
```

```

        inputs
            Collected_Defect_List
            Defect_List1
            Design
        outputs Follow_up_Report
    end
end

```

C.8 N-Fold Inspection

```

inspection '3-Fold Code Inspection'
    declarations
        documents
            Code          product
            Design        source
            Defects1      list
            Defects2      list
            Defects3      list
            Defects4      list
            Defects5      list
            Defects6      list
            Defects7      list
            Defects8      list
            Defects9      list
            Team_A_Defects list
            Team_B_Defects list
            Team_C_Defects list
            Master_Defect_List list
            Team_A_Meeting_Rep report
            Team_B_Meeting_Rep report
            Team_C_Meeting_Rep report
            Follow_Up_Report report
            Master_Plan   plan
        end
        participants
            Coordinator is coordinator
            end
            Moderator1 is moderator
                lists Defects1
            end
            Moderator2 is moderator
                lists Defects2
            end
            Moderator3 is moderator
                lists Defects3
            end
            Inspector1 is inspector
                lists Defects4
            end
        end
    end

```

```

    Inspector2 is inspector
      lists Defects5
    end
    Inspector3 is inspector
      lists Defects6
    end
    Inspector4 is inspector
      lists Defects7
    end
    Inspector5 is inspector
      lists Defects8
    end
    Inspector6 is inspector
      lists Defects9
    end
    Author is author
    end
  end
end
process
  planning 'Planning'
    participants Coordinator
    outputs Master_Plan
  end
  overview 'Overview'
    location local
    participants
      Coordinator
      Moderator1
      Moderator2
      Moderator3
      Inspector1
      Inspector2
      Inspector3
      Inspector4
      Inspector5
      Inspector6
      Author
    presenter Author
    targets Code
  end
  n_fold '3 Fold Inspection'
    fold 'Team A Inspection'
      meeting 'Team A Preparation'
        objective examination
        timing asynchronous
        location local
        visibility private
        participants
          Moderator1

```

```

        Inspector1
        Inspector2
        targets Code
        inputs Design
    end
meeting 'Team A Inspection'
    objective collection
    timing synchronous
    location local
    visibility public
    participants
        Moderator1
        Inspector1
        Inspector2
    roles
        Inspector1 is reader
        Inspector2 is scribe
    targets Code
    inputs Design
    outputs
        Team_A_Defects
        Team_A_Meeting_Rep
    end
end # Team A inspection
fold 'Team B Inspection'
    meeting 'Team B Preparation'
        objective examination
        timing asynchronous
        location local
        visibility private
        participants
            Moderator2
            Inspector3
            Inspector4
        targets Code
        inputs Design
    end
    meeting 'Team B Inspection'
        objective collection
        timing synchronous
        location local
        visibility public
        participants
            Moderator2
            Inspector3
            Inspector4
        roles
            Inspector3 is reader
            Inspector4 is scribe
        targets Code

```

```

        inputs Design
        outputs
            Team_B_Defects
            Team_B_Meeting_Rep
    end # Team B Inspection
end
fold 'Team C Inspection'
    meeting 'Team C Preparation'
        objective examination
        timing asynchronous
        location local
        visibility private
        participants
            Moderator3
            Inspector5
            Inspector6
        targets Code
        inputs Design
    end
    meeting 'Team C Inspection'
        objective collection
        timing synchronous
        location local
        visibility public
        participants
            Moderator3
            Inspector5
            Inspector6
        roles
            Inspector5 is reader
            Inspector6 is scribe
        targets Code
        inputs Design
        outputs
            Team_C_Defects
            Team_C_Meeting_Rep
    end
end # Team C Inspection
collation 'Collation'
    timing synchronous
    location local
    participants
        Coordinator
        Moderator1
        Moderator2
        Moderator3
        Author
    roles
        Moderator1 is reader
        Moderator2 is scribe

```

```
        targets Code
        inputs
            Team_A_Defects
            Team_B_Defects
            Team_C_Defects
        outputs Master_Defect_List
    end
end # 3-Fold
rework 'Rework'
    participant Author
    targets Code
    inputs
        Master_Defect_List
        Design
    end
follow_up 'Follow-up'
    participant Coordinator
    targets Code
    inputs
        Master_Defect_List
        Design
    outputs Follow_Up_Report
end
end
end
```

Appendix D

Experiment Materials

D.1 Timetable

Each experiment was run over a period of ten weeks. Six weeks were used to train students in software inspection and the use of ASSIST, and to refresh their C++ knowledge. Four weeks were used to run the actual experiment. The following timetable was used:

- **Week 1** Individual inspection of `count.cc`.
- **Week 2** Individual inspection of `tokens.cc`, introducing checklist.
- **Week 3** Group meeting to collect the results of Week 2 individual inspection.
- **Week 4** Tutorial introduction to both individual inspection and group meeting using ASSIST with `simple_sort.cc`.
- **Week 5** Individual inspection of `series.cc` using ASSIST.
- **Week 6** Group meeting using ASSIST to collect the results of Week 5 individual inspection.
- **Week 7** Individual inspection of `analyse.cc`. Section 1 made use of ASSIST, while Section 2 performed the inspection on paper.
- **Week 8** Group meeting to collect the results of Week 7 individual inspection, using ASSIST or paper as appropriate.

- **Week 9** Individual inspection of `graph.cc`. Section 1 performed the inspection on paper, while Section 2 made use of ASSIST.
- **Week 10** Group meeting to collect the results of Week 9 individual inspection, using ASSIST or paper as appropriate.

D.2 C++ Checklist

1. General

- Is the functionality described in the specification fully implemented by the code?
- Is there any excess functionality implemented by the code but not described in the specification?
- Is the program interface implemented as described in the specification?
- Is there any dead code which cannot be reached in the program?

2. Variable Initialisation and Declarations

- Is the variable necessary for the operation of the program?
- Is the variable of an appropriate type?
- Is the variable correctly scoped, i.e. does it have the minimum visibility required?
- Is the variable correctly initialised before use?
- Is the variable correctly reinitialised as required?

3. Output Format

- Are there any spelling or grammatical errors in displayed output?
- Is the output complete?
- Is the output formatted correctly in terms of line stepping and spacing?

4. Files

- Are all files properly declared, opened and closed?
- Is a file not closed in the case of an error?
- Are EOF conditions detected and handled correctly?
- Is the file pointer reused without closing the file?

5. Dynamic Storage Allocation

- Is too much/too little space allocated?
- Are all fields of a dynamically allocated structure initialised?
- Is the link in the last node of a dynamic structure always set to NULL?

6. Arrays and Strings

- Check that all strings are identified by pointers and are NULL-terminated at all points in the program
- Is the index expression correct? Are there any off-by-one errors?
- Can array indexes ever go out-of-bounds?

7. Pointers

- Check that the pointer is initialised to NULL
- Check that the pointer is never unexpectedly NULL
- Can an uninitialised pointer ever be dereferenced?
- Is the pointer correctly dereferenced when required?
- Is pointer arithmetic ever used on non-array pointers?

8. If/Else

- Has a semicolon mistakenly been placed at the immediate right of the condition?
- Is the condition correct?
- Are both branches correctly enclosed in braces, as required?

9. Switch

- Is any case not terminated by break or return?
- Does every legal value have a corresponding case?
- Does the statement have a default branch?

10. For

- Has a semicolon mistakenly been placed at the immediate right of the header?
- Has the body been correctly enclosed in braces?
- Has the proper initialisation expression been supplied?

- Has the proper increment expression been supplied?
- Does the loop terminate? Check that the final value can be reached.
- Does the loop perform the correct number of iterations in all cases? Check for off-by-one errors.

11. Do/While

- Has a semicolon mistakenly been placed at the immediate right of the header?
- Is the condition correct?
- Is there an expression within the body which eventually causes termination of the loop? Is any counter incremented?
- Has the body been correctly enclosed in braces?

12. Function Calls

- Are parameters presented in the correct order?
- Are pointers and & used correctly?
- Is the correct function being called, or should it be a different function with a similar name?
- Is the correct value returned from the function?

13. Expressions

- Does operator precedence affect the correct evaluation of the expression, i.e., is there sufficient use of parenthesising to ensure correct evaluation of the expression?
- Can the denominator of a division ever be zero?
- Is integer arithmetic, especially division, ever used inappropriately, causing unexpected truncation/rounding?
- Is the comparison operator correct? Does the expression state exactly what is required?
- Is the boolean operator correct? Does the expression state exactly what is required?
- Do operands of the boolean operation have the value 0 or 1?
- Is an exact equality test used between two floating point numbers?

- Is the comparison between operands of incompatible types?
- If the test is an error-check, can the error condition actually be legitimate in some cases?
- Does the code rely on any implicit type conversions?
- Do any explicit type conversions lose required data?

D.3 Individual Defect Report Form

Complete this form in **BLOCK CAPITALS** in blue or black ink. Each defect description must be complete and accurate. Any description not satisfying the above criteria will be **IGNORED**.

Name:

Group:

Start time:

End time:

Defect No. 1 Time: Location:

Defect No. 2 Time: Location:

Defect No. 3 Time: Location:

Defect No. 4 Time: Location:

Defect No. 5 Time: Location:

Defect No. 6 Time: Location:

Defect No. 7 Time: Location:

Defect No. 8 Time: Location:

Defect No. 9 Time: Location:

Defect No. 10 Time: Location:

Defect No. 11 Time: Location:

Defect No. 12 Time: Location:

Defect No. 13 Time: Location:

Defect No. 14 Time: Location:

Defect No. 15 Time: Location:

Defect No. 16 Time: Location:

Defect No. 17 Time: Location:

Defect No. 18 Time: Location:

Defect No. 19 Time: Location:

Defect No. 20 Time: Location:

D.4 Master Defect Report Form

Complete this form in **BLOCK CAPITALS** in blue or black ink. Each defect description must be complete and accurate. Any description not satisfying the above criteria will be **IGNORED**.

Group:

Moderator:

Reader:

Start time:

Scribe:

Inspector (if present):

End time:

Defect No. 1 Time: Location:

Defect No. 2 Time: Location:

Defect No. 3 Time: Location:

Defect No. 4 Time: Location:

Defect No. 5 Time: Location:

Defect No. 6 Time: Location:

Defect No. 7 Time: Location:

Defect No. 8 Time: Location:

Defect No. 9 Time: Location:

Defect No. 10 Time: Location:

Defect No. 11 Time: Location:

Defect No. 12 Time: Location:

Defect No. 13 Time: Location:

Defect No. 14 Time: Location:

Defect No. 15 Time: Location:

Defect No. 16 Time: Location:

Defect No. 17 Time: Location:

Defect No. 18 Time: Location:

Defect No. 19 Time: Location:

Defect No. 20 Time: Location:

D.5 Training Program count .cc

D.5.1 Specification for program count .cc

Name

count – count lines, words, and characters

Usage

count filename [filename ...]

Description

count counts the number of lines, words, and characters in the named files. Words are sequences of characters that are separated by one or more spaces, tabs, or line breaks (carriage return).

If a file supplied as an argument does not exist, a corresponding error message is printed and processing of any other files continues. If no file is supplied as an argument, **count** reads from the standard input.

The computed values are given for each file (including the name of the file) as well as the sum of all values. If only a single file or if the standard input is processed, then no sum is printed. The output is printed in the order lines, words, characters, and either the file name or the word “total” for the sum. If the standard input is read, the fourth value (name) is not printed.

Options

None.

Examples

On a single file:

```
% count data
      84      462      3621 data
```

On multiple files:

```
% ./count file1 file2 file3
      3        24        120 file1
      5        49        196 file2
     17       175        787 file3
     25       248       1103 total
```

Author

Original by Erik Kamsties and Christopher Lott. C++ conversion and update by Fraser Macdonald.

D.5.2 Library functions used in count .cc

- `open()`
Opens the corresponding I/O stream.
- `char get(void)`
`get` inputs one character from the designated stream and returns this character as the result of the function call. If end-of-file on the stream is encountered, `get` returns EOF.
- `good()`
Returns true if the corresponding I/O stream is available for use.
- `width()`
Sets the field width and returns the previous width for this stream. The width setting applies only to the next stream insertion or extraction.

D.5.3 count .cc

```

1 #include <iostream.h>
2 #include <fstream.h>
3 #include <stdlib.h>
4
5 void main (int argc, char* argv[])
6 {
7     int    c, i, inword;
8     ifstream InputFile;
9     long   linect, wordct, charct;
10    long   tlinect = 1, twordct = 1, tcharct = 1;
11
12    i = 1;
13    do {
14        if (argc > 1) {
15            InputFile.open(argv[i]);
16            if (!InputFile.good()) {
17                cout << "can't open " << argv[i] << endl;
18                exit(1);
19            }
20        }
21        linect = wordct = charct = 0;
22        inword = 1;
23        while ((c = InputFile.get()) != EOF) {
24            ++charct;
25            if (c == '\n')
26                ++linect;
27            if (c == ' ' || c == '\t' || c == '\n')
```

```

28         inword = 0;
29     else if (inword == 0) {
30         inword = 1;
31         ++wordct;
32     }
33 }
34 cout.width(7);
35 cout << linect;
36 cout.width(7);
37 cout << wordct;
38 cout.width(7);
39 cout << charct;
40 if (argc > 1)
41     cout << " " << *argv << endl;
42 else
43     cout << endl;
44 InputFile.close();
45 tlinect += linect;
46 twordct += wordct;
47 tcharct += charct;
48 } while (++i < argc);
49 if (argc > 1) {
50     cout.width(7);
51     cout << linect;
52     cout.width(7);
53     cout << twordct;
54     cout.width(7);
55     cout << tcharct << " total" << endl;
56 }
57 exit(0);
58 }

```

D.5.4 Defects in count .cc

1. Defect in line 10: The variables are initialised with 1, should be with 0.
Causes failure: The sums are incorrect (off by one).
2. Defect in line 14: The variable “InputFile” is not initialised in the case that the input should be taken from “stdin”.
Causes failure: The program cannot read from stdin.
3. Defect in line 17: The error message is sent to “stdout” instead of “stderr”.
Causes failure: Error messages appear on the standard output (stdout) instead of the standard-error output (stderr).

4. Defect in line 18: Component is terminated with “exit (1)”, where “continue” should have been used.

Causes failure: If a file is not found, the program stops there instead of continuing on to other files; also, no sum is printed.

5. Defect in line 22: The variable “inword” is initialised with 1 instead of 0.

Causes failure: Depending on whether the first symbol in a file is whitespace, the program reports that files with n words have either n or $n - 1$ words.

6. Defect in line 41: `*argv` is used instead of `argv[i]`.

Causes failure: The program prints its own name instead of the file name when reporting the counts.

7. Defect in line 49: `Argc` is compared with 1, but should be compared with 2.

Causes failure: The program prints out sums even when only a single file was processed.

8. Defect in line 51: Instead of “`tlinect`” the variable “`linect`” is used.

Causes failure: The sums are not computed correctly. For example:

```
% ./count file2 file2 file2
      1      2      14 ./count
      1      2      14 ./count
      1      2      14 ./count
      1      7      43 total
```

D.6 Training Program `tokens.cc`

D.6.1 Specification for program `tokens.cc`

Name

`tokens` – sort and count alphanumeric tokens

Usage

tokens `[-ai] [-c chars] [-m count]`

Description

tokens reads its input from the standard input and counts all alphanumeric tokens. The tokens are then printed in alphabetic order, along with their counts. Options may be specified to tailor the output. If incorrect options are given, **tokens** will print a usage message.

Options

- “`-a`”: Allow only alphabetic characters in tokens (no digits 0–9).
- “`-c chars`”: Allow chars to be part of tokens.
- “`-i`”: The `-i` flag causes the program to ignore the difference between upper and lower case by mapping all input to lower case.
- “`-m count`”: The `-m` flag indicates the minimum count needed for the entry to be printed.

Examples

In its simplest form with no options:

```
% tokens
this is a_test
    1 a
    1 is
    1 test
    1 this
```

Allowing alphabetic characters only:

```
% tokens -a
test number 2
    1 number
    1 test
```

Using the “`-c`” option to allow “`+`” in tokens:

```
% tokens -c+
one on+ + +
    2 +
```



```
1 on+
1 one
```

Using “-i” to ignore the difference between lower and upper case characters, and using “-m” to set a minimum count of two:

```
% tokens -i -m2
Orange apple orange orange apple banana
2 apple
3 orange
```

Author

Original by Gary Perlman. C++ conversion and update by Fraser Macdonald.

D.6.2 Library functions used in `tokens.cc`

- `void assert(int expression)`
Tests the value of the supplied expression. If the value is 0, `assert` prints an error message and terminates program execution.
- `int atoi(const char* s)`
Converts the string given as argument to its `int` representation.
- `char get(void)`
`get` inputs one character from the designated stream and returns this character as the result of the function call. If end-of-file on the stream is encountered, `get` returns EOF.
- `getopt(int argc, char **argv, char *optstring)`
`extern char *optarg`
`extern int optind`
`getopt()` returns the next option letter in `argv` that matches a letter in `optstring`. `optstring` contains the option letters the command using `getopt()` will recognise; if a letter is followed by a colon, the option is expected to have an argument, or group of arguments, which must be separated from it by white space.
`optarg` is set to point to the start of the option argument on return from `getopt()`.
When all options have been processed, `getopt()` returns -1.

For example,

```
c = getopt(argc, argv, "b:fp:");
```

indicates that three options may be given to the program. The first is “-b”, which should have an argument, followed by “-f”, then “-p”, which should also have an argument (e.g. “-f20”). Each successive call to `getopt()` will return one of these letters, if they are present, and will set `optarg` to point to any argument that may be present for this option. When all options present have been parsed, `getopt()` will return -1.

See the entry for `getopt` in Section 3 of the man pages for more details.

- `int strcmp(const char *s1, const char *s2)`

Compares the two strings passed as arguments, returning an integer greater than, less than, or equal to zero when `s1` is respectively greater than, less than or equal to `s2`.

- `char *strdup(char *s)`

Returns a pointer to a new string which is a duplicate of the string pointed to by `s`.

- `int width(const int newwidth)`

Sets the field width and returns the previous width for this stream. The width setting applies only to the next stream insertion or extraction.

D.6.3 tokens.cc

```

1  #include <iostream.h>
2  #include <string.h>
3  #include <assert.h>
4
5  int    Ignore = 0;
6  int    Mincount = 0;
7  int    Alphabetic = 0;
8  char   MapAllowed[256];
9
10 typedef struct tnode
11 {
12     char   *contents;
13     int    count;
14     struct tnode *left;
15     struct tnode *right;
16 } TNODE;
17
18 void treeprint(TNODE *);
19 TNODE *install(char *, TNODE *);
20 char *getword(void);
21 void tokens(void);

```

```

22
23 int main(int argc, char **argv)
24 {
25     int c, errcnt = 0;
26     extern char *optarg;
27
28     while ((c = getopt(argc, argv, "ac:im:")) != EOF)
29         switch(c)
30             {
31             case 'a': Alphabetic = 0; break;
32             case 'c':
33                 while (*optarg)
34                     {
35                     MapAllowed[*optarg] = *optarg;
36                     optarg++;
37                     }
38                 break;
39             case 'i': Ignore = 1; break;
40             case 'm': Mincount = atoi(optarg); break;
41             }
42     if (errcnt)
43         {
44         cerr << "Usage: " << *argv << " [-i] [-c chars] [-m count]" << endl;
45         exit(1);
46         }
47     for (c = 'a'; c <= 'z'; c++)
48         MapAllowed[c] = c;
49     for (c = 'A'; c <= 'Z'; c++)
50         MapAllowed[c] = Ignore ? c : c - 'A' + 'a';
51     if (!Alphabetic)
52         for (c = '0'; c <= '9'; c++)
53             MapAllowed[c] = c;
54     tokens();
55     exit(0);
56 }
57
58 void treeprint(TNODE *tree)
59 {
60     if (tree != NULL)
61         {
62         treeprint(tree->left);
63         if (tree->count > Mincount) {
64             cout.width(7);
65             cout << tree->count;
66             cout << "\n" << tree->contents << endl;
67         }
68         treeprint(tree->right);
69     }
70 }
71

```

```
72 TNODE *install(char *string, TNODE *tree)
73 {
74     int cond;
75     assert(string != NULL);
76     if (tree == NULL)
77     {
78         if (tree = new TNODE)
79         {
80             tree->contents = strdup(string);
81             tree->count = 1;
82         }
83     }
84     else
85     {
86         cond = strcmp(string, tree->contents);
87         if (cond < 0)
88             tree->left = install(string, tree->left);
89         else if (cond == 0)
90             tree->count++;
91         else
92             tree->right = install(string, tree->right);
93     }
94     return(tree);
95 }
96
97 char *getword(void)
98 {
99     static char string[1024];
100    char *ptr = string;
101    int c, count = 0;
102    for (;;)
103    {
104        c = cin.get();
105        if (c == EOF)
106            if (ptr == string)
107                return(NULL);
108            else
109                break;
110        if (!MapAllowed[c])
111            if (ptr == string)
112                continue;
113            else
114                break; // end of word
115        *ptr++ = MapAllowed[c];
116    }
117    *ptr = NULL;
118    return(string);
119 }
120
121 void tokens(void)
```

```
122 {
123     TNODE *root = NULL;
124     char *s;
125     while (s = getword())
126         root = install(s, root);
127     treeprint(root);
128 }
```

D.6.4 Defects in `tokens.cc`

1. Defect in function “main”, line 27 (circa): The array “MapAllowed” is never initialised.
Failure: Non-alphanumeric symbols could be mistakenly accepted, depending on the contents of MapAllowed.
2. Defect in function “main”, line 31: The variable “Alphabetic” is given the value 0 instead of 1.
Failure: The argument “-a” has no effect.
3. Defect in function “main”, line 41 (circa): No default branch for the case statement.
Failure: Arguments other than those defined in the specification do not cause a usage message to be printed.
4. Defect in function “main”, line 44: The argument “-a” is not documented in the usage message.
Failure: The usage message says nothing about the “-a” argument.
5. Defect in function “main”, line 50: “-i” option not implemented correctly, the branches of the “?” operator are transposed.
Failure: Upper case and lower case characters are always distinguished, irrespective of the use of the “-i” option.
6. Defect in function “treeprint”, line 63: The symbol “>” should be “>=”.
Failure: If a boundary value n is given with the “-m” argument, the value $n + 1$ is used instead of n .
7. Defect in function “treeprint”, line 66: The escape sequence “\n” is used instead of “\t”.
Failure: The output is not formatted correctly. Each token should appear on a line with its count. As written, the program outputs the token, then the count on a new line.

8. Defect in function “install”, line 82 (circa): The left and right branches of the tree should be initialised to NULL.

Failure: No failure apparent, but this may be implementation dependent, i.e. depending on the definition of NULL. Also checklist violation.

9. Defect in function “install”, line 92: The function install is called with incorrect parameters. `tree->left` should be `tree->right`.

Failure: New tokens are inserted into the tree incorrectly. The output is therefore generally unreliable.

10. Defect in function “getword”, line 101: The variable count is declared but never used.

Failure: None apparent, but checklist violation.

11. Defect in function “getword”, line 103/117: The length of the input is not checked.

Failure: The program dumps core if a very long word is read.

D.7 Training Program `simple_sort.cc`

D.7.1 Specification for program `simple_sort.cc`

Name

`simple_sort` – sort a list of numbers entered by the user

Usage

`simple_sort`

Description

`simple_sort` starts by prompting for the number of items to be sorted. The program then reads in the list of numbers from the user, sorts them into ascending numerical order, then prints the sorted list.

Options

None.

Example

Sorting a list of ten numbers:

```
% simple_sort
Enter the number of data values: 10
Data item 0: 5
Data item 1: 6
Data item 2: 7
Data item 3: 8
Data item 4: 2
Data item 5: 3
Data item 6: 9
Data item 7: 1
Data item 8: 4
Data item 9: 10
```

Sorted list:

```
Data item 0: 1
Data item 1: 2
Data item 2: 3
Data item 3: 4
Data item 4: 5
Data item 5: 6
Data item 6: 7
Data item 7: 8
Data item 8: 9
Data item 9: 10
```

Restrictions

The number of elements which can be sorted is currently limited to 100.

Author

Fraser Macdonald.

D.7.2 simple_sort.cc

```

1  #include <iostream.h>
2  #include <stdlib.h>
3
4  const int TABLESIZE = 100;
5
6  void swap(int x, int y)
7  {
8      int temp = x;
9      x = y;
10     y = temp;
11 }
12
13 int max(int x, int y)
14 {if (x > y) return x; else return y;}
15
16 int main()
17 {
18     int size;
19     int table[TABLESIZE];
20
21     cout << "Enter the number of data values: ";
22     cin >> size;
23
24     if(size >= TABLESIZE) {
25         cout << "Too many elements, maximum is " << TABLESIZE << endl;
26         exit(1);
27     }
28     else {
29         for(int i = 0; i < size; i++){
30             cout << "Data item " << i << ": ";
31             cin >> table[i];
32         }
33         for(i = size - 1; i > 0; i--)
34             for(int j = 0; j <= i - 1; j++)
35                 if(table[j] > table[j+1])
36                     swap(table[j], table[j+1]);
37
38         cout << endl << "Sorted lits:" << endl;
39         for(i = 0; i < size; i++)
40             cout << "Data item " << i << ": " << table[i] << endl;
41     }
42     exit(0);
43 }

```

D.7.3 Defects in simple_sort.cc

1. Defect in function “swap”, line 6 : The parameters are passed by value, not by reference.

Failure: “swap” doesn't correctly swap the numbers, therefore the sort is not carried out correctly.

2. Defect in function “max”, line 13 : The function “max” is defined but never used.

Failure: None apparent, but checklist violation.

3. Defect in function “main”, line 24 : \geq should be $>$.

Failure: The program only accepts one less than the true maximum number of elements.

4. Defect in function “main”, line 38 : “list” is spelled incorrectly in the message.

Failure: The program displays incorrect output.

D.8 Training Program `series.cc`

D.8.1 Specification for program `series.cc`

Name

`series` – generate a series of numbers

Usage

`series` start end [stepsize]

Description

`series` prints the real numbers from start to end, one per line. `series` begins with start to which `stepsize` is repeatedly added or subtracted, as appropriate, to approach, possibly meet, but not pass end.

If all arguments are integers, only integers are produced in the output. The stepsize must be nonzero; if it is not specified, it is assumed to be 1. Negative step sizes are made positive. In all other cases, `series` prints an appropriate error message. If the wrong number of arguments are given, `series` prints a usage message.

`series` accepts numbers in several formats: integer, real (where either the whole or fractional part may be omitted) and exponential (an integer or real, suffixed with 'e' or 'E' followed by a (signed) integer exponent). Any number with a fractional part consisting only of zeroes is converted to an integer (e.g. 1.0000 is treated as 1). All numbers may optionally be prefixed by a plus or minus. Examples of acceptable numbers include:

```
+23
-2.4
26.0
92.
.348
1.0E3
34e-2
```

Example

To count from 1 to 100:

```
% series 1 100
1
2
3
....
98
99
100
```

To do the same, but backwards:

```
% series 100 1
100
99
98
...
3
2
1
```

To count from 1.5 to 4.5 in steps of 0.5

```
% series 1.5 4.5 0.5
1.5
2
2.5
```

```
3
3.5
4
4.5
```

Limitations

The reported number of significant digits is limited. If the ratio of the series range to the stepsize is too large, several numbers in a row will be equal.

The maximum length of a series is limited to the size of the maximum integer that can be represented on the machine in use. Exceeding this value has undefined results.

Author

Original by Gary Perlman. C++ conversion and update by Fraser Macdonald.

D.8.2 Library functions used in `series.cc`

- `double atof(char *nptr)`
Converts the initial portion of the string pointed to by `nptr` to double representation. The function returns the converted value.
- `double fabs(double x)`
Computes the absolute value of number `x`
- `int isdigit(char c)`
Returns non-zero integer if the character `c` is a decimal digit.
- `int isspace(char c)`
Returns non-zero integer if the character `c` is a standard whitespace character. The standard whitespace characters are: space(` `), form feed (`\f`), newline (`\n`), carriage return (`\r`), horizontal tab (`\t`) and vertical tab (`\v`).

D.8.3 `series.cc`

```
1 #include <stdlib.h>
2 #include <ctype.h>
3 #include <iostream.h>
4
5 const int IS_NOT = 0;
```

```
6  const int    IS_INT   = 1;
7  const int    IS_REAL  = 2;
8  const int    IS_EXP   = 3;
9  const double FZERO    = 10e-10;
10
11 int fzero(double x);
12 int isinteger (char *string);
13 int number(char *string);
14
15 void main (int argc, char **argv)
16 {
17     long NumItems = 0;
18     double Value = 0.0;
19     double Start = 0.0;
20     double End = 0.0;
21     double Step = 1.0;
22     char *startstr = argv[1];
23     char *endstr = argv[2];
24     char *stepstr = argv[3];
25     int NumArgs = argc;
26
27     switch (NumArgs)
28     {
29     case 3:
30         if (! number(startstr)) {
31             cerr << "Argument #1 not a number: " << startstr << endl;
32             exit(1);
33         }
34         if (! number(startstr)) {
35             cerr << "Argument #2 not a number: " << endstr << endl;
36             exit(1);
37         }
38         if (! number(stepstr)) {
39             cerr << "Argument #3 not a number: " << stepstr << endl;
40             exit(1);
41         }
42         break;
43     case 2:
44         if (! number(startstr)) {
45             cerr << "Argument #1 not a number: " << endstr << endl;
46             exit(1);
47         }
48         if (! number(endstr)) {
49             cerr << "Argument #2 not a number: " << endstr << endl;
50             exit(1);
51         }
52         break;
53     default:
54         cerr << "Usage: " << argv << " start end [stepsize]" << endl;
55         exit(1);
```

```
56     }
57
58     Start = atof(startstr);
59     End = atof(endstr);
60     if (NumArgs == 3) {
61         Step = fabs(atof(stepstr));
62         if (! fzero(End - Start) && fzero(Step))
63             cerr << "stepsize must be non-zero" << endl;
64         exit(1);
65     }
66
67     if (fzero(End - Start))
68         NumItems = 2;
69     else
70         NumItems = (long) (End - Start / Step + 1.0 + FZERO);
71
72     for (int Item = 0; Item < NumItems; Item++) {
73         Value = Start + Step * (double) Item;
74         if (fzero(Value))
75             cout << 0.0 << endl;
76         else
77             cout << Value << endl;
78     }
79
80     exit(0);
81 }
82
83 int fzero (double x)
84 {
85     return (fabs (x) < FZERO);
86 }
87
88 int isinteger (char *string)
89 {
90     return (number(string) == IS_INT);
91 }
92
93 int number (char *string)
94 {
95     int    answer = IS_REAL,
96         before = 0,
97         after = 0;
98     char   *ptr = NULL;
99
100    while (isspace(*string))
101        string++;
102    if (*string == NULL)
103        return (IS_NOT);
104    if (*string == '+' || *string == '-') {
105        string++;
```

```

106         if (isdigit(*string) && *string != '.')
107             return (IS_NOT);
108     }
109     if (isdigit(*string)) {
110         before = 1;
111         while (isdigit(*string))
112             string++;
113     }
114     if (*string == '.') {
115         string++;
116         ptr = string;
117         while (*ptr == '0')
118             ptr++;
119         while (isspace(*ptr))
120             ptr++;
121         if (*ptr == NULL)
122             return (IS_INT);
123         answer = IS_REAL;
124         if (isdigit(*string)) {
125             after = 1;
126             while (isdigit(*string))
127                 string++;
128         }
129     }
130     if (!before && !after)
131         return (IS_NOT);
132     if (*string == 'E' || *string == 'e') {
133         answer = IS_EXP;
134         string++;
135         if (*string == '+' || *string == '-')
136             string++;
137         if (!isdigit(*string))
138             return (IS_NOT);
139         while (isdigit(*string))
140             string++;
141     }
142     while (isspace(*string))
143         string++;
144     return(answer);
145 }

```

D.8.4 Defects in program series.cc

1. Defect in line 25: NumArgs is initialised to argc instead of argc - 1

Failure: With one or two arguments, a segmentation Defect occurs. Any other number of arguments always produces a usage message.

2. Defect in function `main()`, line 34: The variable `startstr` is used in the if-condition instead of `endstr`.

Failure: The program does not recognise a non-numeric second argument as an error.

3. Defect in function `main()`, line 45: The variable `endstr` is used in the output instead of `startstr`.

Failure: Although a non-numeric first argument is recognised as an error, the corresponding error message shows the second argument.

4. Defect in function `main()`, line 54: `argv` should be `*argv`

Failure: Instead of the program name, the program prints the address of its name.

5. Defect in function `main()`, line 62: Mismatch with specification.

Failure: The specification states that `stepsize` should always be non-zero. In fact, the program accepts zero for `stepsize` if the difference between the start and end values is also zero.

6. Defect in function `main()`, line 62–64: Missing brackets for the branch of the if statement.

Failure: If three arguments are given, the program always terminates without any input, since `exit(0)` is always executed.

7. Defect in function `main()`, line 68: The variable `NumItems` is set to 2 instead of 1.

Failure: If the distance between the first and second parameter is evaluated to zero, then two lines are produced as output although only one was expected.

8. Defect in function `main()`, line 70: The calculation `End - Start` should be enclosed in parentheses.

Failure: The value assigned to `NumItems` is incorrect in all cases except where the step size is 1, since the calculation is performed in the wrong order (division is performed first).

9. Defect in function `main()`, line 72–78: Treatment of the case “`end < start`” was forgotten.

Failure: No output is produced in the case that the starting value is greater than the ending value.

10. Defect in function `main()`, line 88–91: Function `isinteger` is defined but never used.

Failure: No failure, but checklist violation.

11. Defect in function `number()`, line 95: `answer` is initialised to `IS_REAL`, should be `IS_INT`.

Failure: `number()` will only return `IS_INT` if the number given is of the form 1.0, 51.000, etc. Normal integers will be mistakenly classified as reals.

12. Defect in function `number()`, line 106: The call to `isdigit(*string)` should be `!isdigit(*string)`

Failure: `number()` will not recognise numbers starting with signs, such as +1.4, -23, etc. The exceptions are anything of the form +.2, -.982, etc., i.e. any number with a point straight after the sign.

13. Defect in function `number()`, line 144: The string should be checked for any remaining characters after a number has been parsed.

Failure: If a number is terminated by non-numeric characters, `number()` does not return `IS_NOT`.

D.9 Experiment Program `analyse.cc`

D.9.1 Specification for program `analyse.cc`

Name

`analyse` – perform simple analysis on survey data

Usage

`analyse` file

Description

`analyse` performs simple statistical analysis on a file containing survey responses. Each response is an integer from 0 to 9 (inclusive). The program calculates four statistics: mean, median, mode and standard deviation.

For the mean, the calculation is presented, along with the answer. For the median, both the unsorted and sorted arrays of responses are printed (formatted in rows of up to twenty numbers), followed by the value of the median. The median is the central value of the ordered data set. If the number of elements in the data set is even, the median is the average of the two most central values. For the mode, a histogram of frequencies of each response is drawn, followed by the value of the mode and its frequency. The mode is the most frequently occurring response. If more than one response shares the highest frequency, the lowest valued response is chosen. For the standard deviation, the answer is simply presented. It is calculated according to the formula $\sqrt{\frac{S}{N}}$, where S is the sum of the squares of the differences between each data element and the mean, and N is the number of elements in the data set.

The input file consists of a list of integer responses in the range 0 to 9, one to each line. The first line of the file contains an integer indicating the number of responses in the file. This value must be greater than zero, otherwise an error message is printed. If no filename is given, the program prints a usage error. If the file cannot be accessed, an appropriate error message is printed.

Example

A typical data file might be:

```
8
6
5
8
9
5
8
5
1
```

The first line indicates that eight responses are included in the file. This is then followed by each of the eight responses. The output for this file is:

```
% analyse data
Processing data

End of file data
```

Mean = $47/8 = 5.875$

The unsorted array is

6 5 8 9 5 8 5 1

The sorted array is

1 5 5 5 6 8 8 9

Median is 5.5

Response	Frequency	Histogram
		1 1 2 2 5 0 5 0 5
0	0	
1	1	*
2	0	
3	0	
4	0	
5	3	***
6	1	*
7	0	
8	2	**
9	1	*

Mode is 5, occurring 3 times.

Standard deviation is 2.36841

Restrictions

The histogram output will only properly display a maximum frequency of 25 responses.

Author

Written by Fraser Macdonald, based on an example from *C: How to Program* by Deitel and Deitel.

D.9.2 Library functions used in `analyse.cc`

- `int eof(void)`
Returns 1 if end-of-file has been encountered in the corresponding stream, otherwise returns 0.
- `double fabs(double x)`
Computes the absolute value of floating point number `x`.
- `int good(void)`
Returns 1 if the corresponding I/O stream is available for use, otherwise returns 0.
- `int open(char* s)`
Opens the corresponding I/O stream.
- `int setw(int x)`
Sets the field width and returns the previous width for this stream.
- `double sqrt(double x)`
Computes the non-negative square root of `x`. An error occurs if the argument is negative.

D.9.3 `analyse.cc`

```
1 #include <iostream.h>
2 #include <iomanip.h>
3 #include <fstream.h>
4 #include <math.h>
5
6 void Mean (int Array[], int Size);
7 void Median (int Array[], int Size);
8 void Mode(int Array[], int Size);
```

```
9 void StandardDeviation(int Array[], int Size);
10 void BubbleSort(int Array[], int Size);
11 void PrintArray(int Array[], int Size);
12
13 int main(int argc, char **argv)
14 {
15     int Item = 0,
16         Size = 0,
17         Count = 0;
18     int *Responses = NULL;
19     ifstream InputFile;
20     char *Filename = NULL;
21
22     if (argc != 2) {
23         cerr << "Usage: " << argv[1] << " file" << endl;
24         exit(1);
25     }
26     Filename = argv[1];
27     InputFile.open(Filename);
28     if (!InputFile.good()) {
29         cerr << "File error." << endl;
30         InputFile.close();
31         exit(1);
32     }
33     cout << "Processing " << Filename << endl;
34     InputFile >> Size;
35     if (Size > 0) {
36         cerr << "One or more responses required." << endl;
37         InputFile.close();
38         exit(1);
39     }
40     else
41         if (!(Responses = new int [Size])) {
42             cerr << "Memory allocation failure." << endl;
43             exit(1);
44         }
45     while (!InputFile.eof()) {
46         InputFile >> Item;
47         Responses[Count++] = Item;
48     }
49     InputFile.close();
50     Mean(Responses, Size);
51     Median(Responses, Size);
```

```

52     Mode(Responses, Size);
53     StandardDeviation(Responses, Size);
54     cout << endl << "End of file " << Filename << endl << endl;
55     exit(0);
56 }
57
58 void Mean(int Array[], int Size)
59 {
60     int Total = 0;
61
62     for (int j = 0; j < Size; j++)
63         Total += Array[j];
64     cout << "Mean = " << Total << "/" << Size << " = "
65         << Total / Size << endl << endl;
66 }
67
68 void Median(int Array[], int Size)
69 {
70     int Median;
71
72     BubbleSort(Array, Size);
73     cout << "The sorted array is" << endl;
74     PrintArray(Array, Size);
75     Median = Array[Size / 2];
76     cout << "Median is " << Median << endl << endl;
77 }
78
79 void Mode(int Array[], int Size)
80 {
81     int Rating, j, h,
82         Largest = 0,
83         ModeValue = 0;
84     int Frequency[10];
85
86     for (j = 0; j < Size; j++)
87         Frequency[Array[j]]++;
88
89     cout << "Responce";
90     cout << setw(11) << "Frequency";
91     cout << setw(19) << "Histogram" << endl << endl;
92     cout << setw(54) << "1    1    2    2" << endl;
93     cout << setw(54) << "5    0    5    0    5" << endl << endl;
94

```

```

95     for (Rating = 0; Rating <= 9; Rating++) {
96         cout << setw(8) << Rating;
97         cout << setw(11) << Frequency[Rating] << "          ";
98         if (Frequency[Rating] > Largest)
99             Largest = Frequency[Rating];
100            ModeValue = Rating;
101            for (h = 1; h < Frequency[Rating]; h++)
102                cout << "*";
103            cout << endl;
104        }
105
106        cout << endl << "Mode is " << ModeValue << ", occurring "
107            << Largest << " times." << endl << endl;
108    }
109
110    void StandardDeviation(int Array[], int Size)
111    {
112        double Total = 0.0,
113            Mean = 0.0,
114            StdDev = 0.0;
115
116        for (int j = 0; j < Size; j++)
117            Total += (double) Array[j];
118        Mean = Total / (double) Size;
119
120        for (j = 0; j < Size; j++)
121            Total = Total + fabs((double) Array[j] - Mean)
122                * fabs((double) Array[j] - Mean);
123        StdDev = sqrt(Total / (double) Size);
124        cout << "Standard deviation is " << StdDev << endl << endl;
125    }
126
127    void BubbleSort(int Array[], int Size)
128    {
129        int Pass, Hold, j;
130
131        for (Pass = 1; Pass < Size; Pass++)
132            for (j = 0; j < Size - 1; j++)
133                if (Array[j] > Array[j+1]) {
134                    Hold = Array[j];
135                    Array[j] = Array[j+1];
136                    Array[j+1] = Hold;
137                }

```

```

138 }
139
140 void PrintArray(int Array[], int Size)
141 {
142     for (int j = 0; j < Size; j++) {
143         if (j % 20 == 0) cout << endl;
144         cout << setw(2) << Array[j];
145     }
146     cout << endl << endl;
147 }

```

D.9.4 Defects in program analyse.cc

1. Defect in function `main()`, line 23: `argv[1]` should be replaced by `argv[0]`
Failure: Program does not print out its own name as part of the usage message. The name printed is undefined.
2. Defect in function `main()`, line 35: The test for an illegal value of `Size` is incorrect.
Failure: Program will not detect illegal values of `Size`, and correct values will cause the program to fail with an error message.
3. Defect in function `main()`, line 45–48: If the file contains more data elements than the number declared at the start, the elements are written into unallocated memory.
Failure: None apparent, but unallocated memory is overwritten.
4. Defect in function `main()`, line 54: End of file message occurs in incorrect position.
Failure: The end of file message is printed after all calculations have been printed, instead of the correct time.
5. Defect in function `Mean()`, line 65: The calculation of the mean requires an explicit cast to `float` of both operands.
Failure: The calculation of the mean is always truncated.
6. Defect in function `Median()`, line 71: The specification states that the unsorted array should be printed out when calculating the median. This is not done.
Failure: Output incomplete
7. Defect in function `Median()`, line 75: The specification states that if there is an even number of responses then the median should be the average of the two most central values, but the program always uses a single central value.

Failure: Incorrect median for data sets with an even number of elements.

8. Defect in function `Mode()`, line 84: The array `Frequency` is not initialised before use.

Failure: Checklist violation, behaviour undefined.

9. Defect in function `Mode()`, line 89: `'Response'` spelled incorrectly.

Failure: Incorrect label printed.

10. Defect in function `Mode()`, line 98–100: Brackets missing around the contents of the `if` statement.

Failure: The mode is always set to 9.

11. Defect in function `Mode()`, line 101: The terminating condition of the `for` loop is incorrect, printing out one less than the required number of asterisks.

Failure: Incorrect histogram display.

12. Defect in function `StandardDeviation()`, line 119: `Total` is not re-initialised before being used in calculating the standard deviation.

Failure: Calculation of standard deviation is incorrect.

D.10 Experiment Program `graph.cc`

D.10.1 Specification for program `graph.cc`

Name

`graph` – draw a graph

Usage

`graph` file

Description

Given an input file of ordered pairs (x, y) of either positive or negative integers as input, the program displays the list of points read in and plots them on a grid with a horizontal x-axis and a vertical y-axis which are appropriately labeled, and have 'tick' marks every five units.

A plotted point on the grid appears as an asterisk (*), and the grid is scaled to fit into an area with a maximum width of 40 characters and a maximum height of 20 characters.

Vertical scaling is handled as follows. The total height of the graph is calculated as the difference between the largest y value (or zero if the largest is negative) and the smallest y value (or zero if the smallest value is positive). If this height is less than the maximum height of the graph, no scaling is carried out and the graph is plotted with vertical spacing of one line per integral unit (e.g., the point (3, 6) should be plotted on the sixth line above the origin; two lines above the point (3, 4)). Note that the origin (point (0, 0)) corresponds to the intersection of the axes (the x-axis is referred to as the O^{th} line). The origin is represented by a '+' and the graph is drawn to ensure that the origin and axes always appear.

If the height is greater than the maximum height of the graph, the scale for vertical spacing is set to the maximum possible height divided by the total height required. This scaling factor is then applied to every point on the graph and the result rounded appropriately to ensure the point lies within the correct interval. For example, if the the graph was required to display the points (1, 1) and (1, 99) the total height is 100 (since the origin must also be displayed). The scaling factor is then $20/100 = 0.2$. (1, 1) is displayed on the 0th line (which covers the interval 0 to 4) and (1, 99) is displayed on the 19th line (which covers 95–99). Negative coordinate values are treated in a similar way. Horizontal scaling is handled similarly.

If two or more of the points to be plotted would show up as the same asterisk in the grid, the number of points occurring on that grid position appears instead of the asterisk. Points whose asterisks will lie on an axis or other marker show up in place of that item.

The input file consists of list of integer coordinates, with each x-coordinate followed by the corresponding y-coordinate. If no filename is given, the program prints a usage error. If the file cannot be accessed, an appropriate error message is printed. If an odd number of coordinates are present in the file, an appropriate error message is printed.

Example

A typical data file might be:

```
-100
-100
0
0
4
19
```

```

5
20
99
99
49
49
48
48

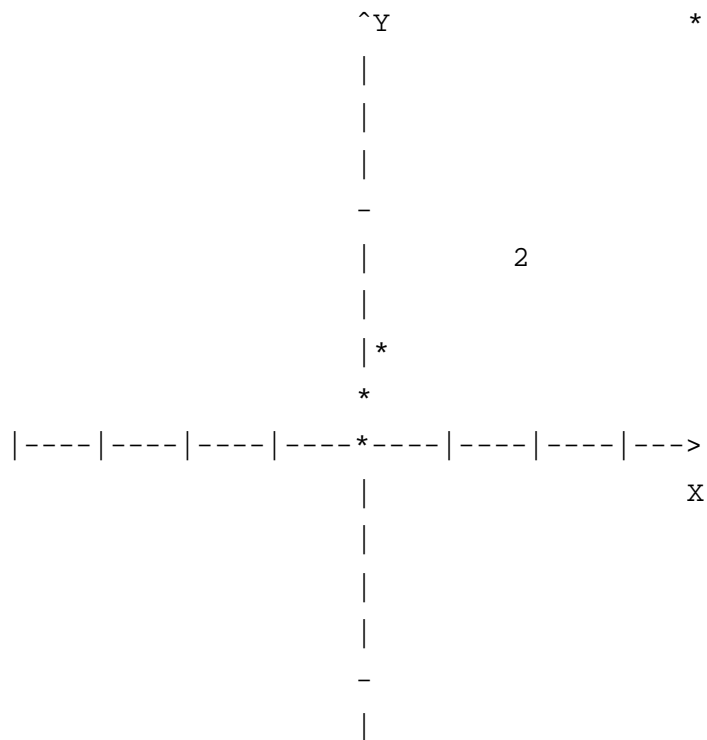
```

The printed output from this file is:

```

% graph data
(48, 48)
(49, 49)
(99, 99)
(5, 20)
(4, 19)
(0, 0)
(-100, -100)

```



```

          |
          |
          |
*         -

```

Restrictions

The program will only correctly deal with up to nine overlapping points, since numbers beyond this occupy more than a single grid position.

Author

Written by Fraser Macdonald, based on a specification from Basili and Selby's *Comparing the effectiveness of software testing techniques*, IEEE Transactions on Software Engineering, 13(12), December 1987.

D.10.2 Library functions used in graph.cc

- `int abs(int x)`
Computes the absolute value of integer `x`.
- `int good(void)`
Returns 1 if the corresponding I/O stream is available for use, otherwise returns 0.
- `int open(char* s)`
Opens the corresponding I/O stream.

D.10.3 graph.cc

```

1 #include <iostream.h>
2 #include <fstream.h>
3 #include <stdlib.h>
4
5 const int GWIDTH = 40;
6 const int GHEIGHT = 40;
7
8 typedef struct pointnode {
9     int X;
10    int Y;
11    struct pointnode *Next;
12 } PointNode;
13

```

```

14 typedef PointNode *PointNodePtr;
15
16 void InsertPoint(int XValue, int YValue, PointNodePtr PointList);
17 void PrintPointList(PointNodePtr PointList);
18 void PlotGraph(PointNodePtr PointList, float XShift, float YShift,
19               float XScale, float YScale, char Output[GWIDTH][GHEIGHT]);
20 void DrawGraph(PointNodePtr PointList);
21
22 int main(int argc, char **argv)
23 {
24     PointNodePtr PointList = NULL;
25     ifstream InputFile;
26     char *Filename = NULL;
27     int X = 0, Y = 0;
28
29     if (argc != 2) {
30         cerr << "Usage: " << argv[0] << " file" << endl;
31         exit(1);
32     }
33     Filename = argv[1];
34     InputFile.open(Filename);
35     if (InputFile.good()) {
36         cerr << "File error on " << Filename << endl;
37         InputFile.close();
38         exit(1);
39     }
40     while (InputFile >> X)
41         if (InputFile >> Y)
42             InsertPoint(X, Y, PointList);
43     else {
44         cerr << "Error: no Y value for X = " << X << endl;
45         InputFile.close();
46         exit(1);
47     }
48     InputFile.close();
49     DrawGraph(PointList);
50     exit(0);
51 }
52
53 void InsertPoint(int XValue, int YValue, PointNodePtr PointList)
54 {
55     PointNodePtr NewNode = NULL;
56
57     if (NewNode = new PointNode) {
58         NewNode->X = XValue;
59         NewNode->Y = YValue;
60         NewNode->Next = PointList;
61         PointList = NewNode;
62     }
63     else

```

```

64     cerr << "Error allocating memory." << endl;
65 }
66
67 void PrintPointList(PointNodePtr PointList)
68 {
69     PointNodePtr Current = PointList;
70
71     while(Current != NULL) {
72         cout << "(" << Current->Y << ", " << Current->X << ")" << endl;
73         Current = Current->Next;
74     }
75     cout << endl;
76 }
77
78 void PlotGraph(PointNodePtr PointList, float XShift, float YShift,
79               float XScale, float YScale, char Output[GWIDTH][GHEIGHT])
80 {
81     int x = 0, y = 0;
82     PointNodePtr Current = PointList;
83
84     for (y = 0; y < GHEIGHT; y++)
85         Output[(int)XShift][y] = '|';
86     Output[(int)XShift][GHEIGHT - 1] = '^';
87     Output[(int)XShift + 1][GHEIGHT - 1] = 'Y';
88
89     for (x = 0; x < GWIDTH; x++)
90         Output[x][(int)YShift] = '-';
91     Output[GWIDTH - 1][(int)YShift] = '>';
92     Output[GWIDTH - 1][(int)YShift - 1] = 'X';
93     Output[(int)XShift][(int)YShift] = '+';
94
95     while (Current != NULL) {
96         x = (int)((float) Current->X * XScale + XShift);
97         y = (int)((float) Current->Y * YScale + YShift);
98         switch(Output[x][y]) {
99             case '-' : case '|' :
100             case '+' : case ' ' : Output[x][y] = '*'; break;
101             case '*' : Output[x][y] = '2'; break;
102             default : Output[x][y] = Output[x][y] + 1; break;
103         }
104         Current = Current->Next;
105     }
106 }
107
108 void DrawGraph(PointNodePtr PointList)
109 {
110     int     SmallestX = 0, LargestX = PointList->X,
111           SmallestY = 0, LargestY = PointList->Y,
112           Width = 0, Height = 0,
113           x = 0, y = 0;

```

```

114 float XScale = 1.0, YScale = 1.0,
115       XShift = 0.0, YShift = 0.0;
116 PointNodePtr Current = PointList;
117 char Output[GWIDTH][GHEIGHT];
118
119 while (Current != NULL) {
120     if (Current->X < SmallestX) SmallestX = Current->X;
121     if (Current->X > LargestX) LargestX = Current->X;
122     if (Current->Y < SmallestY) SmallestY = Current->Y;
123     if (Current->Y > LargestY) LargestY = Current->Y;
124     Current = Current->Next;
125 }
126
127 Width = LargestX - SmallestX + 1;
128 Height = LargestY - SmallestY + 1;
129 if (Width > GWIDTH) XScale = (float) Width / (float) GWIDTH;
130 if (Height > GHEIGHT) YScale = (float) Height / (float) GHEIGHT;
131 if (SmallestX < 0) XShift = (float) abs(SmallestX) * XScale;
132 if (SmallestY < 0) YShift = (float) abs(SmallestY) * YScale;
133
134 PlotGraph(PointList, XShift, YShift, XScale, YScale, Output);
135
136 for (y = 0; y < GHEIGHT; y++) {
137     for (x = 0; x < GWIDTH; x++)
138         cout << Output[x][y];
139     cout << endl;
140 }
141 }

```

D.10.4 Defects in program graph.cc

1. Defect in function `main()`, line 6: Wrong value given to `GHEIGHT`.
Failure: The program prints a graph 40 by 40, instead of 40 by 20.
2. Defect in function `main()`, line 35: The test for a correctly opened file should have a `!` in front of it.
Failure: The program always fails if the file is correctly opened.
3. Defect in function `main()`, line 48: Call to `PrintPointList` missing.
Failure: The list of points is not displayed before the graph is drawn, as is required by the specification.
4. Defect in function `InsertPoint()`, line 16, 53: `PointList` passed by value instead of by reference.
Failure: `PointList` is never updated with the inserted pointed.

5. Defect in function `PrintPointList()`, line 72: X and Y values of `Current` transposed.
Failure: The printed points appear with the Y coordinate followed by the X coordinate.
6. Defect in function `PlotGraph()`, line 83: The array `Output` is not initialised.
Failure: The output from the program is unpredictable, since the point drawing routine depends on the array being initialised with spaces.
7. Defect in function `PlotGraph()`, line 84, 89: The code does not deal with inserting tick marks in the graph.
Failure: The printed graph does not have the tick marks required by the specification.
8. Defect in function `PlotGraph()`, line 92: One is subtracted from the Y coordinate of the position for the X axis label.
Failure: The array index may go out of bounds if `YShift` is zero.
9. Defect in function `PlotGraph()`, line 98: Conditions for `switch` statement incomplete.
Failure: If a point occurs on the graph over the axis labels or arrows, the point is not drawn correctly, since these cases are not explicitly tested for in the `switch`.
10. Defect in function `DrawGraph()`, line 110, 111: `LargestX` and `LargestY` are given the wrong initial values.
Failure: If all coordinates are negative, and the graph has to be scaled, the axes are not drawn. These should be set to 0 initially.
11. Defect in function `DrawGraph()`, line 129, 130: Incorrect calculation of scaling factors.
Failure: If scaling is required the program usually terminates abnormally as the output array is indexed out of bounds.
12. Defect in function `DrawGraph()`, line 136: The loop control for printing the vertical axis proceeds from bottom to top instead of top to bottom.
Failure: The graph is printed out upside down.

D.11 Questionnaires

D.11.1 Questionnaire 1

Section 1: C++ Browser

1. Indicate how much you used each of the following features of the C++ browser during individual preparation by marking the appropriate box.

Feature -----	Frequency of use -----
Window split	Never[] Once[] Twice[] Many times[]
Inspected code indication (reduced font)	Never[] Once[] Twice[] Many times[]
Line/column number indicator	Never[] Once[] Twice[] Many times[]
Coverage indication (percentage)	Never[] Once[] Twice[] Many times[]
New annotation button	Never[] Once[] Twice[] Many times[]
Delete annotation button	Never[] Once[] Twice[] Many times[]
Show annotation button	Never[] Once[] Twice[] Many times[]
Cycle annotation button	Never[] Once[] Twice[] Many times[]
Find facility	Never[] Once[] Twice[] Many times[]
Cross-referencing facility	Never[] Once[] Twice[] Many times[]

2. Overall, how did you find navigating around the document you were inspecting?

(1 = very difficult, 2 = difficult, 3 = average, 4 = easy, 5 = very easy)

1[] 2[] 3[] 4[] 5[]

3. Was the size of the text font used...

Too small[] Just right[] Too large[]

4. Was the default window size used...

Too small[] Just right[] Too large[]

5. Did the use of a reduced font to indicate coverage of the code affect your ability to read the code?

Still readable[] Unreadable[]

Section 2: List Browser

6. Indicate how much you used each of the following features of the list browser in ASSIST during individual preparation by marking the appropriate box.

Feature	Frequency of use
-----	-----
Show item	Never[] Once[] Twice[] Many times[]
New item	Never[] Once[] Twice[] Many times[]
Cut item	Never[] Once[] Twice[] Many times[]
Copy item	Never[] Once[] Twice[] Many times[]
Paste item	Never[] Once[] Twice[] Many times[]
JumpTo item	Never[] Once[] Twice[] Many times[]

7. How did you find creating comments?

(1 = very difficult, 2 = difficult, 3 = average, 4 = easy, 5 = very easy)

1[] 2[] 3[] 4[] 5[]

8. Was the default window size used...

Too small[] Just right[] Too large[]

Section 3: General

9. Indicate how much you used each of the following features of ASSIST during individual preparation by marking the appropriate box.

Feature -----	Frequency of use -----
Checklist	Never[] Once[] Twice[] Many times[]
C++ reference	Never[] Once[] Twice[] Many times[]

10. Describe the strategy you used for inspecting the code. For example, sequential, bottom up, top down, etc.

11. Overall, how easy to use was the list browser/C++ browser combination?
(1 = very difficult, 2 = difficult, 3 = average, 4 = easy, 5 = very easy)

1[] 2[] 3[] 4[] 5[]

12. Did you prefer creating comments from within the C++ browser or the list browser, or did you have no preference?

C++ browser[] List browser[] No preference[]

13. Overall, did you find that the number of windows used by ASSIST reduced your inspection efficiency, had no effect on your inspection efficiency, or improved your inspection efficiency?

Improved[] No effect[] Reduced[]

14. Compared with paper-based inspection, do you feel that computer-based inspection is more efficient, less efficient, or about equally efficient?

Computer-based is less efficient[] Equal[] Computer-based is more efficient[]

15. Are there any facilities which you feel ASSIST could provide to enhance the individual preparation phase?

16. Please use this space to detail any problems you had when using ASSIST for

individual inspection, or any other comments you may have.

[End of questionnaire]

D.11.2 Questionnaire 2

Section 1: C++ Browser

1. Please indicate how much you used each of the following features of the ASSIST C++ browser during the group meeting by marking the appropriate box.

Feature -----	Frequency of use -----
Window split	Never[] Once[] Twice[] Many times[]
Inspected code indication (reduced font)	Never[] Once[] Twice[] Many times[]
Line/column number indicator	Never[] Once[] Twice[] Many times[]
Coverage indication (percentage)	Never[] Once[] Twice[] Many times[]
New annotation button	Never[] Once[] Twice[] Many times[]
Delete annotation button	Never[] Once[] Twice[] Many times[]
Show annotation button	Never[] Once[] Twice[] Many times[]
Cycle annotation button	Never[] Once[] Twice[] Many times[]
Find facility	Never[] Once[] Twice[] Many times[]
Cross-referencing facility	Never[] Once[] Twice[] Many times[]

2. ANSWER ONE QUESTION ONLY

If you WERE THE READER, please indicate how you found using the focus system to guide your team through the document.

(1 = very difficult, 2 = difficult, 3 = average, 4 = easy, 5 = very easy)

1[] 2[] 3[] 4[] 5[]

If you WERE NOT THE READER, please indicate how happy you were with the reader's control of the focus?

(1 = unhappy, 2= slightly unhappy, 3 = neither happy or unhappy, 4 = fairly happy, 5 = very happy)

1[] 2[] 3[] 4[] 5[]

Section 2: List Browser

3. Please indicate how much you used each of the following features of the list browser in ASSIST during the group meeting by marking the appropriate box.

Feature	Frequency of use
-----	-----
Show item	Never[] Once[] Twice[] Many times[]
New item	Never[] Once[] Twice[] Many times[]
Cut item	Never[] Once[] Twice[] Many times[]
Copy item	Never[] Once[] Twice[] Many times[]
Paste item	Never[] Once[] Twice[] Many times[]
JumpTo item	Never[] Once[] Twice[] Many times[]
Propose item	Never[] Once[] Twice[] Many times[]
Update item (scribe only)	Never[] Once[] Twice[] Many times[]

4. How easy/difficult to use did you find the defect proposal system?

(1 = very difficult, 2 = difficult, 3 = average, 4 = easy, 5 = very easy)

1[] 2[] 3[] 4[] 5[]

5. How much did the voting mechanism help resolve issues?

(1 = hindered, 2 = no effect, 3 = helped)

1[] 2[] 3[]

Section 3: General

6. Indicate how much you used each of the following features of ASSIST during the group meeting by marking the appropriate box.

Feature	Frequency of use
-----	-----
Checklist	Never[] Once[] Twice[] Many times[]
C++ reference	Never[] Once[] Twice[] Many times[]

7. Did you feel that using ASSIST had any effect on your group discussion? E.g. you may have been able to discuss issues more quickly (a positive effect) or you may have found it more difficult to discuss issues (a negative effect). (1 = large negative effect, 2 = small negative effect, 3 = no effect, 4 = small positive effect, 5 = large positive effect)

1[] 2[] 3[] 4[] 5[]

8. Describe in detail the effect ASSIST had on your group meeting (if any).

9. Compared with a paper-based group meeting, do you feel that a computer-based group meeting is more efficient, less efficient, or about equally efficient?

Computer-based is less efficient[]
 Computer-based is equally efficient[]
 Computer-based is more efficient[]

10. Are there any facilities which you feel ASSIST could provide to enhance the group meeting phase?

11. Please use this space to detail any problems you had when using ASSIST for the group meeting, or any other comments you may have.

[End of questionnaire]

D.11.3 Questionnaire 3

Section 1 - Individual Inspection

1. With regard to length, do you think that the analyse.cc code was...

Too long[] Just right[] Too short[]

2. Do you think that the analyse.cc code was...

Too complex[] Just right[] Too simple[]

3. Approximately how much of analyse.cc did you understand?

0-20% [] 21-40% [] 41-60% [] 61-80% [] 81-100% []

4. Do think that two hours was sufficient time to find defects in analyse.cc?

Not enough time [] Just right [] Too much time []

Section 2 - Group Meeting

5. Was your understanding of analyse.cc changed at the group meeting?

Understanding confounded []
No change in understanding
[] Understanding increased []

6. How many defects were discovered during the meeting that were not identified during individual inspection?

0 [] 1-2 [] 3-5 [] >5 []

7. How many defects do you think that your meeting lost, i.e. defects which were identified during individual inspections but not recorded during the group meeting?

0 1-2 3-5 >5

8. What percentage of all the defects in the document do you estimate your group found?

0-20% 21-40% 41-60% 61-80% 81-100%

9. How many of the group's reported defects did you not agree with?

0 1-2 3-5 >5

10. Which of the following best describes the relative contribution of the individuals in your group?

- All contributed approximately the same
 One person contributed noticeably more than the rest
 One person contributed noticeably less than the rest
 One person contributed noticeably more, and one person contributed noticeably less

11. Which of the following objectives were achieved by your group meeting? (you may mark more than one)

- Merging defect lists
 Additional defect detection
 Group bonding/improving team spirit
 Education of weak group members
 Ensuring adequate individual inspection

 Section 3 - ASSIST-based Inspection (ASSIST users ONLY)

12. Did you feel ASSIST was an impediment to any aspects of your performance? If so, please state which aspects and why.

13. Did you feel ASSIST improved any aspects of your performance? If so, please state which aspects and why.

14. Overall, how would you rate the usability of ASSIST for inspection?
 (1 = extremely usable, 2 = fairly usable, 3 = average, 4 = fairly unusable,
 5 = totally unusable)

1[] 2[] 3[] 4[] 5[]

[End of questionnaire]

D.11.4 Questionnaire 4

Section 1 - Individual Inspection

1. Approximately how much of graph.cc did you understand?

0-20% [] 21-40% [] 41-60% [] 61-80% [] 81-100% []

2. Do think that two hours was sufficient time to find defects in graph.cc?

Not enough time [] Just right [] Too much time[]

3. In terms of complexity, how did graph.cc compare with analyse.cc?

Much more complex []

Slightly more complex []

Of similar complexity []

Slightly less complex []

Much less complex []

Section 2 - Group Meeting

4. Was your understanding of graph.cc changed at the group meeting?

Understanding confounded []

No change in understanding []

Understanding increased []

5. How many defects were discovered during the meeting that were not identified during individual inspection?

0 [] 1-2 [] 3-5 [] >5 []

6. How many defects do you think that your meeting lost, i.e. defects which were identified during individual inspections but were accidentally not

recorded during the group meeting?

0 1-2 3-5 >5

7. How many of the group's reported defects did you not agree with?

0 1-2 3-5 >5

8. Which of the following best describes the relative contribution of the individuals in your group?

- All contributed approximately the same
 One person contributed noticeably more than the rest
 One person contributed noticeably less than the rest
 One person contributed noticeably more, and one person contributed noticeably less

9. Which of the following objectives were achieved by your group meeting?
 (you may mark more than one)

- Merging defect lists
 Additional defect detection
 Group bonding/improving team spirit
 Education of weak group members
 Ensuring adequate individual inspection

10. In comparison with your previous group meeting (involving analyse.cc), which of the following do you think are true:

- The group performed better
 The group performed about the same
 The group performed worse

Please give reasons:

 Section 3 - ASSIST-based Inspection (ASSIST users ONLY)

11. Did you feel ASSIST was an impediment to any aspects of your performance?
 If so, please state which aspects and why.

12. Did you feel ASSIST improved any aspects of your performance? If so, please state which aspects and why.

13. Overall, how would you rate the usability of ASSIST for inspection?
(1 = extremely usable, 2 = fairly usable, 3 = average, 4 = fairly unusable, 5 = totally unusable)

1[] 2[] 3[] 4[] 5[]

Section 4 - General

14. How well do you think you understand software inspections?

Completely [] Well [] Reasonably Well [] Not too sure [] Not at all []

15. Overall, do you think your knowledge of C/C++ was adequate for the tasks set?

Inadequate [] Adequate []

If you feel your knowledge was inadequate, please explain further:

16. Overall, did you feel you performed better during individual inspection paper-based inspection or ASSIST, or were you equally effective with both methods?

Performed better with paper-based []
Performed equally well with both methods []
Performed better with ASSIST []

If you felt you performed better with a particular method, please explain:

17. Did you feel you made more, less or about the same amount of use of your checklist during an ASSIST inspection as compared to a paper-based inspection?

Used checklist more with paper-based
About the same for both paper-based and ASSIST
Used checklist more with ASSIST

18. Overall, did you feel your group performed better during the meeting using paper-based inspection or ASSIST, or were you equally effective with both methods?

Performed better with paper-based
Performed equally well with both methods
Performed better with ASSIST

If you felt you performed better with a particular method, please explain:

19. Some people believe it is easier to work with paper-based documents than screen-based documents, while others believe it is easier to work with screen-based documents than paper-based documents. From your experience of paper-based and tool-assisted inspection, which do you prefer?

Prefer screen-based No preference Prefer paper-based

Please give reasons:

20. Overall, what did you find to be the most difficult aspect of this part of the course?

21. Any other comments about this part of the course?

[End of questionnaire]

Appendix E

Raw Data

E.1 Comparing Paper-based and Tool-based Software Inspection

E.1.1 Experiment 1

Group no.	Subjects
1	1, 2, 3
2	4, 5, 6
3	7, 8, 9
4	10, 11, 12
5	13, 14, 15
6	16, 17, 18
7	19, 20, 21, 22
8	23, 24, 25
9	26, 27, 28
10	29, 30, 31
11	32, 33, 34
12	35, 36, 37
13	38, 39, 40
14	41, 42, 43

Table E.1: Allocation of subjects to groups.

This section presents the raw defect detection data collected from the first experiment. Table E.1 shows the allocation of subjects to groups. Table E.2 presents the defect detection data for the individual inspection of `analyse.cc` in Experiment 1. Each row contains the data for one subject and shows the technique used by that subject and which defects that subject detected. A summary score, indicating the number of correct defects and the total

Subject	Method	Defect Number												Total
		1	2	3	4	5	6	7	8	9	10	11	12	
1	T	X	X		X	X	X	X	X	X		X	X	10/11
2	T		X		X	X	X	X		X		X	X	7/10
3	T	X	X		X		X	X	X		X	X	X	9/10
4	T	X	X	X		X	X	X		X		X	X	9/12
5	T		X		X	X	X	X		X	X	X	X	9/14
6	T		X		X	X	X	X		X	X	X	X	6/10
7	T		X		X	X	X	X		X	X	X	X	9/14
8	T	X	X		X	X	X		X	X	X	X	X	9/13
9	T		X			X	X		X	X				5/9
10	T		X			X	X	X		X	X	X		7/10
11	T	X	X		X	X		X			X	X		7/7
12	T		X		X	X	X		X		X	X		6/13
13	T		X		X		X	X		X	X	X		7/13
14	T	X	X		X		X	X		X	X	X	X	9/9
15	T	X	X		X			X	X	X	X	X	X	9/11
16	T	X	X				X	X				X	X	6/10
17	T	X	X				X	X		X		X		6/10
18	T	X	X	X	X	X	X	X		X	X	X		10/14
19	T	X	X		X	X	X	X		X			X	8/11
20	T	X	X		X		X	X	X		X	X		8/15
21	T	X	X		X		X	X		X	X	X		8/13
22	T		X		X		X	X			X			5/9
23	P	X	X			X	X	X	X	X	X	X	X	10/14
24	P		X				X	X		X	X			4/11
25	P	X	X	X			X	X		X		X	X	8/13
26	P	X	X				X	X		X		X	X	7/13
27	P	X	X		X	X	X	X				X	X	8/10
28	P	X	X		X	X	X		X	X		X		7/9
29	P	X	X	X	X	X	X	X	X	X	X	X		10/14
30	P	X	X				X	X		X	X			7/14
31	P	X	X		X	X	X	X		X		X		8/14
32	P	X	X		X		X	X	X			X	X	8/11
33	P	X	X		X		X	X		X	X			7/12
34	P	X	X		X		X	X	X	X				7/12
35	P	X	X		X	X	X	X	X	X	X	X	X	11/17
36	P	X	X				X	X			X	X		6/8
37	P		X				X	X		X	X	X	X	7/11
38	P	X	X		X		X	X		X			X	7/12
39	P	X	X	X	X	X	X	X	X	X	X	X	X	12/14
40	P		X		X			X	X		X	X	X	6/11
41	P	X	X		X		X	X		X	X	X	X	9/15
42	P		X				X	X		X		X		5/5
43	P	X	X			X	X	X		X	X	X	X	9/11

Table E.2: Raw data for individual phase of analyse.cc inspection. Method is either (P)aper or (T)ool. An 'X' indicates a defect reported by the individual. The total is given as the total number of correct defects out of the total number submitted.

Group	Method	Defect Number												Total	Gains	Losses
		1	2	3	4	5	6	7	8	9	10	11	12			
1	T	X	X		X	X	X	X	X	X	X	X	X	11/12	0	0
2	T	X	X	X	X	X	X	X	<u>X</u>	X	X	X	X	12/14	1	0
3	T	X	X		X	X	X	X		X	X	X	X	10/14	0	0
4	T	X	X		X	X	X	X	X	X	X	X	X	11/15	0	0
5	T	X	X		X		X	X	X	X	X	X	X	10/13	0	0
6	T	X	X	-	X	X	X	X	<u>X</u>	X	X	X	X	11/16	1	1
7	T	X	X		X	X	X	X	X	X	X	X	X	11/17	0	0
8	P	X	X	X		X	X	X	X	X	X	X	X	11/15	0	0
9	P	X	X		X	X	X	X	X	X		X	X	10/12	0	0
10	P	X	X	X	X	X	X	X	X	X	X	X		11/15	0	0
11	P	X	X		X	<u>X</u>	X	X	X	X	X	X	X	11/16	1	0
12	P	X	X		X	X	X	X	X	X	X	X	X	11/13	0	0
13	P	X	X	X	X	X	X	X	X	X	X	X	X	12/16	0	0
14	P	X	X		X	-	X	X		X	X	X	X	9/10	0	1

Table E.3: Raw data for group meeting with analyse.cc. Method is either (P)aper or (T)ool. An 'X' indicates a defect reported by the group. A '-' represents a meeting loss. An underlined 'X' represents a meeting gain. The total is given as the total number of correct defects out of the total number submitted.

number of defects submitted, is also given. Table E.3 gives defect detection data for the group phase. It has the same format as the table for individuals, but also contains data on meeting gains and losses.

Defect detection data for the individual and group phases of the `graph.cc` inspection are given in Table E.4 and Table E.5 respectively. Their format is identical to that of the tables showing the data for the `analyse.cc` inspection.

Subject	Method	Defect Number												Total
		1	2	3	4	5	6	7	8	9	10	11	12	
1	P	X	X	X		X	X	X			X	X	X	9/10
2	P		X	X		X	X	X						5/8
3	P	X	X	X			X	X		X		X		7/8
4	P	X	X		X	X	X	X		X		X		8/12
5	P	X	X	X		X	X	X				X		7/11
6	P	X	X	X		X		X			X	X		7/9
7	P	X	X	X		X		X	X	X	X		X	10/13
8	P	X	X	X		X		X				X		6/9
9	P	X	X	X	X	X						X		6/13
10	P	X	X	X		X								4/9
11	P	X	X	X		X	X	X						6/8
12	P	X	X	X	X	X			X				X	7/8
13	P	X	X	X		X				X	X			6/7
14	P	X	X	X				X			X	X		6/9
15	P	X		X	X	X	X	X	X				X	8/8
16	P	X	X	X		X	X					X		6/11
17	P		X	X		X					X			4/11
18	P	X	X			X	X	X		X	X	X		8/12
19	P	X	X	X		X				X		X		6/10
20	P	X	X	X		X	X	X				X		7/12
21	P	X	X	X		X		X				X		6/8
22	P	X	X	X		X		X		X		X		7/11
23	T	X	X		X	X		X			X	X	X	8/12
24	T	X	X			X				X		X		5/8
25	T	X	X			X		X				X		5/12
26	T	X	X	X		X	X	X				X		7/7
27	T	X	X	X	X	X		X				X		7/9
28	T	X	X			X		X				X		4/10
29	T	X	X	X		X	X	X		X	X	X	X	10/10
30	T	X	X	X										3/9
31	T	X	X			X		X				X		5/9
32	T	X	X	X		X						X		5/11
33	T		X	X			X		X					4/9
34	T	X	X	X		X	X	X		X	X	X		9/10
35	T	X	X	X	X	X		X				X		7/9
36	T	X	X	X		X					X	X		6/7
37	T	X	X	X								X		4/9
38	T	X	X	X		X	X	X				X		7/11
39	T	X	X		X	X	X	X			X	X	X	9/10
40	T	X					X	X				X		4/8
41	T	X	X	X		X					X	X		6/7
42	T	X				X						X		3/7
43	T	X	X	X		X		X		X	X	X		8/10

Table E.4: Raw data for individual phase of graph .cc inspection.

Group	Method	Defect Number												Total	Gains	Losses
		1	2	3	4	5	6	7	8	9	10	11	12			
1	P	X	X	X		X	X	X		X	X	X	X	10/10	0	0
2	P	X	X	X	X	X	X	X	X	X	-	X	X	11/15	2	1
3	P	X	X	X	-	X		X	X	X	X	X	X	10/12	0	1
4	P	X	X	X	-	X	X	X	X			X	X	9/10	1	1
5	P	X	X	X	X	X	X	X	X	X	X	-	X	11/14	0	1
6	P	X	X	X		X	X	X		X	-	X		8/12	0	1
7	P	X	X	X		X	X	X		X		X		8/10	0	0
8	T	X	X	X	X	X		X		X	X	X	X	10/12	1	0
9	T	X	X	X	-	X	-	X				X	X	7/8	1	2
10	T	X	X	X		X	X	X		X	-	X	X	9/11	0	1
11	T	X	X	X		X	X	X	-	X	-	X		8/15	0	2
12	T	X	X	X	X	X	X	X			X	X		9/12	1	0
13	T	X	X	X	X	X	X	X			X	X	X	10/12	0	0
14	T	X	X	X		X	X	X		X	X	X		9/9	1	0

Table E.5: Raw data for group meeting with graph . cc.

E.1.2 Experiment 2

Group no.	Subjects
1	1, 2, 3
2	4, 5, 6
3	7, 8, 9
4	10, 11, 12
5	13, 14, 15
6	16, 17, 18
7	19, 20, 21
8	22, 23, 24
9	25, 26, 27
10	28, 29, 30
11	31, 32, 33
12	34, 35, 36
13	37, 38, 39
14	40, 41, 42
15	43, 44, 45
16	46, 47, 48, 49

Table E.6: Allocation of subjects to groups.

The raw data collected from the second experiment is presented in the section. Table E.6 shows the allocation of subjects to groups. Table E.7 summarises the defect detection data for the individual inspection of `analyse.cc`. Its form is identical to that of Table E.2. Table E.8 shows the raw defect detection data for the group phase. Its format is identical to that of Table E.3

Defect detection data for the individual and group phases of the `graph.cc` inspection is shown in Table E.9 and Table E.10 respectively.

Subject	Method	Defect Number												Total
		1	2	3	4	5	6	7	8	9	10	11	12	
1	T	X	X	X	X	X	X	X	X	X	X	X		10/13
2	T		X		X		X	X		X		X		6/13
3	T		X				X	X				X	X	4/8
4	T	X	X		X		X	X		X	X	X	X	9/11
5	T	X	X		X		X	X		X	X	X	X	9/10
6	T		X		X		X			X	X			5/9
7	T		X		X	X	X			X	X	X	X	8/9
8	T		X		X		X	X		X	X			6/11
9	T	X	X		X	X	X			X	X	X		8/10
10	T	X	X	X	X			X			X	X		7/10
11	T	X	X					X			X	X		5/8
12	T		X					X	X		X	X		5/19
13	T		X					X		X	X			3/8
14	T		X		X			X	X	X	X		X	6/10
15	T		X		X			X	X	X				5/12
16	T	X	X					X	X	X		X	X	8/10
17	T		X		X	X	X	X		X	X		X	8/12
18	T													0/0
19	T	X	X					X	X		X			6/7
20	T	X	X					X	X			X	X	5/11
21	T		X		X			X	X		X			6/10
22	T	X	X	X	X			X	X		X	X	X	9/14
23	T		X		X			X			X	X		5/12
24	T	X	X		X			X			X	X		7/8
25	P	X	X		X			X	X		X	X	X	9/12
26	P				X			X	X		X		X	5/12
27	P		X		X			X	X		X		X	6/14
28	P	X	X	X				X	X		X	X		7/11
29	P	X	X					X	X		X	X		7/13
30	P	X	X		X			X	X		X		X	8/11
31	P	X	X	X	X			X	X	X			X	8/15
32	P		X					X						2/4
33	P	X	X		X			X		X	X	X		7/17
34	P	X	X					X	X		X	X	X	7/12
35	P	X	X		X			X	X		X	X	X	9/9
36	P		X		X			X		X				4/12
37	P	X	X	X	X			X		X	X	X		9/16
38	P	X	X					X	X	X		X		7/17
39	P	X	X		X			X	X	X	X		X	8/13
40	P		X					X	X		X	X		6/15
41	P	X	X		X			X	X	X		X	X	9/14
42	P	X	X		X			X		X				5/14
43	P	X	X		X			X		X		X	X	7/13
44	P		X		X			X	X			X		5/12
45	P		X		X			X	X	X	X		X	7/14
46	P	X			X			X	X	X	X			6/12
47	P		X					X	X		X			4/12
48	P	X	X		X			X	X	X	X	X	X	8/10
49	P	X	X		X			X	X		X	X		7/18

Table E.7: Raw data for individual phase of analyse.cc inspection. Note that subject 18 did not take part in this phase, and is discounted from the analysis.

Group	Method	Defect Number												Total	Gains	Losses
		1	2	3	4	5	6	7	8	9	10	11	12			
1	T	X	X	X	X	X	X	X	X	X	X	X	X	12/15	0	0
2	T	X	X		X		X	X		X	X	X	X	9/10	0	0
3	T	X	X		X		X	X		X	X	X	X	9/14	0	0
4	T	X	X	-	X		X	X		X	X	X		8/12	0	1
5	T	X	X		X		X	X		X	X		-	7/13	1	1
6	T	X	X		X	X	X	X	X	X	X	X	X	11/15	0	0
7	T	X	X		X		X	X		X	X	X	X	9/14	0	0
8	T	X	X	X	X	X	X	X		X	X	X	X	11/14	1	0
9	P	X	X	X	X	X	X	X		X	X	X	X	11/16	2	0
10	P	X	X	-	X	X	X	X		X	X	X	X	10/12	1	1
11	P	X	X	X	X		X	X	X	X	X	X	X	11/15	0	0
12	P	X	X		X		X	X		X	X	X	X	9/12	0	0
13	P	X	X	X	X		X	X	X	X	X	X	X	11/15	0	0
14	P	X	X		X	X	X	X	X	X	X	X	X	11/18	1	0
15	P	-	X		X	X	X	X		X	X	X	X	9/17	1	1
16	P	X	X		X		X	X	X	X	X	X	X	10/17	0	0

Table E.8: Raw data for group meeting with analyse .cc.

Subject	Method	Defect Number												Total
		1	2	3	4	5	6	7	8	9	10	11	12	
1	P	X	X	X		X		X			X	X	X	8/11
2	P	X	X	X		X					X			5/7
3	P	X	X			X	X				X	X		6/10
4	P	X	X	X				X						4/6
5	P	X	X	X				X	X					5/7
6	P	X	X	X				X				X		5/11
7	P	X	X	X		X		X		X		X		7/9
8	P	X	X	X				X				X		5/9
9	P	X	X	X	X	X		X	X			X		8/10
10	P		X	X		X		X		X		X		6/7
11	P	X	X	X										3/7
12	P	X	X	X						X	X	X		6/9
13	P	X	X	X		X		X				X		6/9
14	P	X	X	X			X	X			X			6/6
15	P	X	X	X				X			X	X		6/11
16	P	X	X			X	X	X		X		X		7/8
17	P	X	X	X				X		X		X		6/9
18	P	X	X			X						X		4/9
19	P	X	X	X		X		X			X	X		7/10
20	P	X	X	X				X			X	X		6/10
21	P		X	X		X						X		3/6
22	P	X	X	X		X		X		X		X	X	8/10
23	P	X	X	X	X	X		X		X		X	X	9/10
24	P	X	X	X				X		X		X		6/8
25	T		X	X		X		X		X	X	X	X	8/10
26	T	X	X	X							X	X		4/9
27	T	X	X	X		X		X			X	X		7/13
28	T	X	X	X				X		X	X	X		7/9
29	T	X	X			X		X				X		5/8
30	T	X	X	X				X		X		X		6/9
31	T	X				X		X		X	X			5/6
32	T	X	X	X		X		X			X	X		7/9
33	T		X	X				X						2/4
34	T	X	X	X				X		X			X	6/7
35	T	X	X	X				X			X	X		6/7
36	T	X	X			X						X		5/11
37	T	X	X	X	X	X		X		X		X		8/12
38	T	X	X	X		X					X	X		6/12
39	T	X	X	X			X	X				X	X	7/11
40	T	X	X	X				X						4/10
41	T	X	X	X				X			X	X		6/7
42	T	X	X			X				X		X		5/11
43	T	X	X	X		X		X			X	X		7/7
44	T	X	X	X		X		X			X	X		7/8
45	T	X	X	X				X			X	X		6/12
46	T	X	X	X			X					X	X	6/10
47	T	X	X	X			X				X	X		6/9
48	T	X	X	X		X		X				X		6/10
49	T	X	X			X		X				X		5/10

Table E.9: Raw data for individual phase of graph .cc inspection.

Group	Method	Defect Number												Result	Gains	Losses
		1	2	3	4	5	6	7	8	9	10	11	12			
1	P	X	X	X	<u>X</u>	X	X	X			X	X	X	10/11	1	0
2	P	X	X	X		<u>X</u>		X	X		<u>X</u>	X	<u>X</u>	9/11	3	0
3	P	X	X	X	-	X		X	X	X		X	<u>X</u>	9/11	1	1
4	P	X	X	X		X		X	X	X	X	X		8/12	0	0
5	P	X	X	X		X	-	X			X	X		7/9	0	1
6	P	X	X	X		X	X	X		X		X	<u>X</u>	9/11	1	0
7	P	X	X	X		X		X	X	<u>X</u>	X	X		8/11	1	0
8	P	X	X	X	-	X	<u>X</u>	X	<u>X</u>	X	<u>X</u>	X	X	11/14	3	1
9	T	X	X	X		X		X		X	X	X	X	9/12	0	0
10	T	X	X	X		X		X		X	X	X	<u>X</u>	9/14	1	0
11	T	X	X	X		X		X		X	X	X	<u>X</u>	9/12	1	0
12	T	X	X	X		-		X		X	X	X	X	8/9	0	1
13	T	X	X	X	-	X	X	X		X	X	X	X	10/12	0	1
14	T	X	X	X		X		X		X	X	X		8/10	0	0
15	T	X	X	X		X		X			X	X	<u>X</u>	8/10	1	0
16	T	X	X	X		X	X	X		-	X	X	X	8/13	0	1

Table E.10: Raw data for group meeting with graph . cc.

E.2 Automated Defect List Collation

E.2.1 Experiment 1

Table E.11 shows the average percentage of correct defects in defect lists generated with various content and threshold settings. Table E.12 shows the average percentage of duplicates in defect lists generated with various content and threshold settings. The defect lists used are those from the tool users from the first experiment comparing paper-based and tool-based inspection.

Contents	Threshold																		
	0.05	0.10	0.15	0.20	0.25	0.30	0.35	0.40	0.45	0.50	0.55	0.60	0.65	0.70	0.75	0.80	0.85	0.90	0.95
0.05	8.93	8.93	8.93	8.93	8.93	8.93	10.53	14.36	22.40	23.80	32.51	48.58	94.68	100.0	100.0	100.0	100.0	100.0	100.0
0.10	8.93	8.93	8.93	8.93	8.93	8.93	10.53	14.95	21.75	25.30	33.16	58.36	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.15	8.93	8.93	8.93	8.93	8.93	8.93	10.53	14.95	22.40	27.80	41.05	65.56	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.20	8.93	8.93	8.93	8.93	8.93	8.93	10.53	14.95	21.60	29.31	44.16	78.85	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.25	8.93	8.93	8.93	8.93	8.93	8.93	10.53	14.95	21.60	30.43	46.60	90.42	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.30	8.93	8.93	8.93	8.93	8.93	8.93	10.36	17.20	21.60	31.73	51.70	95.10	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.35	8.93	8.93	8.93	8.93	8.93	8.93	10.36	16.49	20.89	34.39	61.45	97.84	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.40	8.93	8.93	8.93	8.93	8.93	8.93	11.25	16.49	21.78	35.28	70.36	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.45	8.93	8.93	8.93	8.93	8.93	8.93	11.25	17.92	21.78	44.68	79.56	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.50	8.93	8.93	8.93	8.93	8.93	8.93	10.53	18.63	24.38	49.72	85.20	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.55	8.93	8.93	8.93	8.93	8.93	8.93	10.53	17.19	27.35	56.56	87.86	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.60	8.93	8.93	8.93	8.93	8.93	8.93	9.64	16.29	30.90	63.04	90.52	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.65	8.93	8.93	8.93	8.93	8.93	8.93	10.36	16.95	34.59	67.97	94.68	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.70	8.93	8.93	8.93	8.93	8.93	8.93	10.36	17.35	38.70	73.02	96.16	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.75	8.93	8.93	8.93	8.93	8.93	8.93	10.36	18.24	44.09	76.71	98.64	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.80	8.93	8.93	8.93	8.93	8.93	8.93	11.00	19.14	45.53	77.31	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.85	8.93	8.93	8.93	8.93	8.93	8.93	11.00	21.57	51.47	81.27	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.90	8.93	8.93	8.93	8.93	8.93	8.93	11.00	21.57	56.63	84.76	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.95	8.93	8.93	8.93	8.93	8.93	8.93	12.55	25.84	62.31	86.43	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0

Table E.11: Average percentage of correct defects in auto-collated lists for varying threshold and contents factors, using Experiment 1 defect lists.

Contents	Threshold																		
	0.05	0.10	0.15	0.20	0.25	0.30	0.35	0.40	0.45	0.50	0.55	0.60	0.65	0.70	0.75	0.80	0.85	0.90	0.95
0.05	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	7.61	100.0	100.0	100.0	100.0	100.0	100.0
0.10	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	51.75	100.0	100.0	100.0	100.0	100.0	100.0
0.15	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.59	80.81	100.0	100.0	100.0	100.0	100.0	100.0
0.20	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.44	89.50	100.0	100.0	100.0	100.0	100.0	100.0
0.25	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	7.17	91.81	100.0	100.0	100.0	100.0	100.0	100.0
0.30	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	18.98	92.76	100.0	100.0	100.0	100.0	100.0	100.0
0.35	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.79	32.76	94.92	100.0	100.0	100.0	100.0	100.0	100.0
0.40	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.12	47.70	96.51	100.0	100.0	100.0	100.0	100.0	100.0
0.45	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	4.07	54.70	97.30	100.0	100.0	100.0	100.0	100.0	100.0
0.50	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	7.79	65.74	97.30	100.0	100.0	100.0	100.0	100.0	100.0
0.55	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.44	14.66	72.27	97.30	100.0	100.0	100.0	100.0	100.0	100.0
0.60	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.30	26.07	78.69	97.30	100.0	100.0	100.0	100.0	100.0	100.0
0.65	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2.15	34.75	82.77	97.70	100.0	100.0	100.0	100.0	100.0	100.0
0.70	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	4.44	41.64	85.93	97.70	100.0	100.0	100.0	100.0	100.0	100.0
0.75	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.79	7.55	47.70	87.99	97.70	100.0	100.0	100.0	100.0	100.0	100.0
0.80	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.48	13.77	51.43	89.11	97.70	100.0	100.0	100.0	100.0	100.0	100.0
0.85	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.95	20.38	56.25	89.83	97.70	100.0	100.0	100.0	100.0	100.0	100.0
0.90	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2.54	24.84	61.12	90.30	97.70	100.0	100.0	100.0	100.0	100.0	100.0
0.95	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	3.91	27.52	65.57	90.30	97.70	100.0	100.0	100.0	100.0	100.0	100.0

Table E.12: Average percentage of duplicates in auto-collated lists for varying threshold and contents factors, using Experiment 1 defect lists.

E.2.2 Experiment 2

Table E.13 shows the average percentage of correct defects in defect lists generated with various content and threshold settings. Table E.14 shows the average percentage of duplicates in defect lists generated with various content and threshold settings. The defect lists used are those from the tool users from the second experiment comparing paper-based and tool-based inspection.

Contents	Threshold																		
	0.05	0.10	0.15	0.20	0.25	0.30	0.35	0.40	0.45	0.50	0.55	0.60	0.65	0.70	0.75	0.80	0.85	0.90	0.95
0.05	8.35	8.35	8.35	8.35	8.35	8.35	9.13	11.45	17.19	31.13	46.15	70.03	98.61	100.0	100.0	100.0	100.0	100.0	100.0
0.10	8.35	8.35	8.35	8.35	8.35	8.35	9.13	11.50	17.19	32.02	46.94	74.36	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.15	8.35	8.35	8.35	8.35	8.35	8.35	9.13	11.50	18.45	32.72	51.79	84.04	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.20	8.35	8.35	8.35	8.35	8.35	8.35	9.13	12.20	21.16	34.73	54.30	87.19	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.25	8.35	8.35	8.35	8.35	8.35	8.35	9.13	12.76	21.81	34.58	59.36	95.62	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.30	8.35	8.35	8.35	8.35	8.35	8.35	9.13	12.76	23.59	38.99	64.78	97.54	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.35	8.35	8.35	8.35	8.35	8.35	8.35	9.13	12.76	23.03	42.42	72.17	98.81	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.40	8.35	8.35	8.35	8.35	8.35	8.35	9.13	12.76	24.92	43.59	81.81	99.38	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.45	8.35	8.35	8.35	8.35	8.35	8.35	9.13	11.87	28.92	50.35	86.21	99.38	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.50	8.35	8.35	8.35	8.35	8.35	8.35	9.13	12.96	32.79	55.64	92.52	99.38	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.55	8.35	8.35	8.35	8.35	8.35	8.35	9.13	16.14	34.36	59.28	94.00	99.38	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.60	8.35	8.35	8.35	8.35	8.35	8.35	9.13	15.94	38.86	66.91	94.57	99.38	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.65	8.35	8.35	8.35	8.35	8.35	8.35	9.13	16.63	41.69	72.78	95.89	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.70	8.35	8.35	8.35	8.35	8.35	8.35	9.13	18.00	42.21	75.75	95.19	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.75	8.35	8.35	8.35	8.35	8.35	8.35	9.13	20.36	45.24	80.48	95.71	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.80	8.35	8.35	8.35	8.35	8.35	8.35	10.70	19.58	46.83	83.00	96.28	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.85	8.35	8.35	8.35	8.35	8.35	8.35	12.16	24.89	49.73	85.77	96.85	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.90	8.35	8.35	8.35	8.35	8.35	8.35	12.16	27.06	53.92	86.47	96.33	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.95	8.35	8.35	8.35	8.35	8.35	8.35	14.61	31.45	56.66	86.47	96.33	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0

Table E.13: Average percentage of correct defects in auto-collated lists for varying threshold and contents factors, using Experiment 2 defect lists.

Contents	Threshold																		
	0.05	0.10	0.15	0.20	0.25	0.30	0.35	0.40	0.45	0.50	0.55	0.60	0.65	0.70	0.75	0.80	0.85	0.90	0.95
0.05	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.62	1.85	10.97	41.55	100.0	100.0	100.0	100.0	100.0	100.0
0.10	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.62	1.59	13.41	71.45	100.0	100.0	100.0	100.0	100.0	100.0
0.15	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.62	3.57	15.68	92.28	100.0	100.0	100.0	100.0	100.0	100.0
0.20	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.52	5.52	20.34	96.31	100.0	100.0	100.0	100.0	100.0	100.0
0.25	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.52	5.52	28.85	98.25	100.0	100.0	100.0	100.0	100.0	100.0
0.30	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.63	8.63	41.78	98.25	100.0	100.0	100.0	100.0	100.0	100.0
0.35	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	10.45	52.46	98.25	100.0	100.0	100.0	100.0	100.0	100.0
0.40	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.25	13.15	63.49	98.25	100.0	100.0	100.0	100.0	100.0	100.0
0.45	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.07	17.52	70.75	98.25	100.0	100.0	100.0	100.0	100.0	100.0
0.50	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.70	21.93	77.26	98.25	100.0	100.0	100.0	100.0	100.0	100.0
0.55	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	4.14	32.20	83.28	98.25	100.0	100.0	100.0	100.0	100.0	100.0
0.60	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	5.04	37.70	86.17	98.25	100.0	100.0	100.0	100.0	100.0	100.0
0.65	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	5.92	43.30	87.84	98.25	100.0	100.0	100.0	100.0	100.0	100.0
0.70	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.25	9.23	52.80	91.12	98.25	100.0	100.0	100.0	100.0	100.0	100.0
0.75	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.67	13.52	57.08	91.23	98.25	100.0	100.0	100.0	100.0	100.0	100.0
0.80	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	3.04	23.64	61.55	90.60	98.25	100.0	100.0	100.0	100.0	100.0	100.0
0.85	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2.56	26.11	65.34	91.64	97.80	100.0	100.0	100.0	100.0	100.0	100.0
0.90	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	3.34	25.94	70.25	93.10	97.80	100.0	100.0	100.0	100.0	100.0	100.0
0.95	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	7.30	27.62	72.56	93.10	97.80	100.0	100.0	100.0	100.0	100.0	100.0

Table E.14: Average percentage of duplicates in auto-collated lists for varying threshold and contents factors, using Experiment 2 defect lists.