Towards Computer-Supported Collaborative Software Engineering

---

A thesis
submitted in partial fulfilment
of the requirements for the Degree
of
Doctor of Philosophy
in the
University of Canterbury
by
Carl Cook

Examining Committee

Professor John Grundy	Examiner
Associate Professor Nicholas Graham	Examiner
Doctor Neville Churcher	Supervisor, Head of Department
Associate Professor Andy Cockburn	Associate Supervisor

University of Canterbury
2007

المنارة للاستشارات

www.manaraa.com

المنارة للاستشارات
www.manaraa.com

To Amanda, with all my love.

www.manaraa.com

المنارة للاستشارات
www.manaraa.com

Abstract

Software engineering is a fundamentally collaborative activity, yet most tools that support software engineers are designed only for single users. There are many foreseen benefits in using tools that support real time collaboration between software engineers, such as avoiding conflicting concurrent changes to source files and determining the impact of program changes immediately.

Unfortunately, it is difficult to develop non-trivial tools that support real time Collaborative Software Engineering (CSE). Accordingly, the few CSE tools that do exist have restricted capabilities.

Given the availability of powerful desktop workstations and recent advances in distributed computing technology, it is now possible to approach the challenges of CSE from a new perspective. The research goal in this thesis is to investigate mechanisms for supporting real time CSE, and to determine the potential gains for developers from the use of CSE tools. An infrastructure, Caise, is presented which supports the rapid development of real time CSE tools that were previously unobtainable, based on patterns of collaboration evident within software engineering.

In this thesis, I discuss important design aspects of CSE tools, including the identification of candidate patterns of collaboration. I describe the Caise approach to supporting small teams of collaborating software engineers. This is by way of a shared semantic model of software, protocol for tool communication, and Computer Supported Collaborative Work (CSCW) facilities. I then introduce new types of synchronous semantic model-based tools that support various patterns of CSE. Finally, I present empirical and heuristic evaluations of typical development scenarios.

Given the Caise infrastructure, it is envisaged that new aspects of collaborative work within software engineering can be explored, allowing the perceived benefits of CSE to be fully realised.

المنارة للاستشارات

www.manaraa.com

المنارة للاستشارات
www.manaraa.com

Table of Contents

List of Figures viii

List of Tables xii

Chapter 1: Introduction 1

1.1 An Example Collaborative Development Scenario 4

1.1.1 Background 4

1.1.2 Conventional Tools 5

1.1.3 Collaborative Tools 6

1.2 Research Goals 7

1.3 Thesis Outline 8

Chapter 2: Background 9

2.1 The Process of Software Engineering 9

2.1.1 Overview 9

2.1.2 Software Engineering Methodologies and Processes . . 11

2.1.3 Software Engineering Artifacts 12

2.2 The Significance of Collaboration 15

2.2.1 Collaboration Defined 15

2.2.2 Project Life-Cycles 15

2.2.3 Conventional Support for Collaboration 16

2.2.4 The Progression of Software Engineering 19

2.3 Defining Real Time Collaborative Software Engineering 22

2.4 Research Related to Collaborative Software Engineering 23

2.4.1 Software Engineering Processes 24

2.4.2 Groupware and CSCW 25

2.4.3 Source Code Control Systems 27

2.4.4 Human Computer Interaction 28

2.4.5 Distributed Systems 29

المنارة للاستشارات

www.manaraa.com

2.4.6 Software Engineering Metrics and Visualisation 29
2.5 Previous Work Towards Collaborative Software Engineering . 30
2.5.1 Overview 30
2.5.2 Design Tools 31
2.5.3 Development Tools 33
2.5.4 Inspection Tools 37
2.5.5 Comparison to CAISE-Based Tools 38
2.6 Collaborative Software Engineering Barriers 40
2.6.1 Groupware Support 40
2.6.2 Building Industrial-Strength Tools 41

Chapter 3: Patterns of Collaboration 43
3.1 Motivation 43
3.1.1 The Patterns Language 44
3.1.2 An Example Pattern 45
3.2 Patterns of Interaction 48
3.3 Collaboration within Software Engineering 50
3.3.1 Modes of Collaboration 50
3.3.2 Current Facilities for Collaboration 51
3.3.3 Examples of Existing Collaboration Support 52
3.3.4 Types of Awareness 53
3.3.5 Atomic Elements of Collaboration 53
3.4 Candidate Patterns of Collaborative Software Engineering . . 54
3.4.1 Formal Identification of Patterns 55
3.4.2 A Patterns Map for Collaborative Software Engineering 55
3.4.3 Applying Patterns of Collaborative Software Engineering 61
3.4.4 Collaboration Antipatterns 63

Chapter 4: Supporting Collaborative Software Engineering 67
4.1 Tool Support for Collaborative Software Engineering 68
4.1.1 The Need for Better Communication Support 68
4.1.2 Common Tool Design Approaches 69
4.2 Considerations for Tool Developers 71
4.2.1 Tool Design 71

المنارة للاستشارات
www.manaraa.com

4.2.2 Requirements for Large-Scale Development 75
4.2.3 Threats to Tool Acceptance 77
4.2.4 Future Tool Design 78
4.3 Semantic Model-Based Software Engineering 78
4.3.1 Constructing a Semantic Model of Software 80
4.3.2 Sharing the Project Model 82
4.3.3 The Code Neighbourhood 83
4.4 Awareness Support 86
4.4.1 Types of Awareness 86
4.4.2 Media Richness 91
4.4.3 The Collaborative Spectrum 92

Chapter 5: The CAISE Framework **95**

5.1 The Need for a Better Tool Support 95
5.1.1 Motivation 96
5.1.2 Framework-Based Tool Support 98
5.2 Overview of the CAISE Framework 100
5.2.1 Architecture 101
5.3 Architectural Design 105
5.3.1 The Project Semantic Model 107
5.3.2 The CAISE Event Model 114
5.3.3 Artifact Modification 116
5.3.4 The CAISE Server 120
5.3.5 Collaborative Tool Support 124
5.3.6 Framework Extensibility 125
5.4 Related Work 126

Chapter 6: Using the CAISE Framework **128**

6.1 Overview of Current CAISE-Based Tools 128
6.2 CAISE-Based Tool Construction 130
6.2.1 Tool Construction Overview 130
6.2.2 Tool Services 131
6.2.3 CAISE Tool Widgets 133
6.2.4 The CAISE Tool Protocol 138

المنارة للاستشارات
www.manaraa.com

6.2.5 Building a New CAISE-Based Tool 140
6.2.6 Coding Examples 145
6.3 Example CSE Tools 156
6.3.1 Code Editors 158
6.3.2 Diagramming Tools 163
6.3.3 IDE Integration 167
6.3.4 Constructing Collaborative Widgets 167

Chapter 7: Evaluation of the CAISE Framework and Tools 169

7.1 Heuristic Evaluation 169
7.1.1 Heuristic Evaluation of Groupware 171
7.1.2 Heuristics for CSE Evaluations 172
7.2 Visualisation Tools 175
7.2.1 The Visualisation Pipeline 176
7.2.2 User Activity Visualisation 176
7.2.3 Artifact-Span Visualisation 178
7.3 User Evaluations 178
7.3.1 Evaluation Method 180
7.3.2 Evaluation Results 183
7.3.3 Threats to Validity 186
7.3.4 Discussion 187
7.4 Framework Performance 189
7.4.1 Memory Load 189
7.4.2 Network Load 191
7.4.3 Response Times 192
7.4.4 Feedback Information versus Number of Users 193

Chapter 8: CAISE in an Industrial Context 194

8.1 Managing Groups and Individuals 194
8.1.1 Working from a Source Code Repository 194
8.1.2 Partitioning of Projects 197
8.1.3 Compilation Crosstalk 200
8.1.4 Private Work 203
8.2 Large Software Projects 203

المنارة للاستشارات
www.manaraa.com

8.2.1 Tailoring Feedback 204
8.3 Areas of Enhancement 205
8.3.1 CSCW Floor Control Policies 205
8.3.2 Atomic Operations versus Refactoring 206

Chapter 9: Conclusions and Future Work **208**
9.1 Conclusions 208
9.2 Future Work 209
9.2.1 Areas of Investigation 210
9.2.2 Future Evaluations 211

References **227**

Acknowledgments **228**

Appendices **229**

Appendix A: The CAISE Server **230**
A.1 Overview 230
A.2 Language Support 230
A.2.1 Parsers 231
A.2.2 Source Code Formatters 232
A.3 Artifacts 233
A.3.1 The CAISE Document Buffer 234
A.3.2 Implementing Collaborative Undo 235
A.3.3 Tool Manager Plug-Ins 236
A.4 Server Applications 237
A.5 The CAISE Event Log 239
A.6 Project Administration 239
A.7 The Plug-Ins Interface 240
A.8 Interprocess Communication 241
A.8.1 Asynchronous Communication 241
A.8.2 Synchronous Communication 242
A.8.3 Selection of Distributed Communication Technologies . 242

المنارة للاستشارات
www.manaraa.com

Appendix B: Language Specification for Decaf 244
B.1 Overview 244
B.2 An Example Source File 244
B.3 An Example Grammar 244

Appendix C: CAISE Event Log DTD 247

Appendix D: CAISE Server Plug-Ins Specification 251
D.1 CAISEAnalyser 251
D.2 CAISEFeedback 252
D.3 CAISEFormatter 253
D.4 CAISEParser 253
D.5 CAISEServerApp 253
D.6 CAISEToolManager 254

Appendix E: IDE Integration 256

Appendix F: User Evaluation Design 260
F.1 Overview 260
F.2 Aim and Purpose 261
F.3 Evaluation Methodology 261
F.3.1 Participants 262
F.3.2 Physical Layout 262
F.3.3 Apparatus 263
F.3.4 Supporting a Minimal Code Repository Interface . . . 263
F.3.5 Tool Modes 264
F.3.6 Task Types 265
F.3.7 Order of Groups and Tasks 266
F.3.8 Training Manual 267
F.3.9 Evaluation Tasks 268
F.3.10 User Survey 268
F.3.11 Statistical Validity 268

Appendix G: User Evaluation Documents 273
G.1 Evaluation Documents 273

المنارة للاستشارات
www.manaraa.com

G.1.1 Training Manual 273
G.1.2 Training Tasks 274
G.1.3 Evaluation Tasks 275
G.1.4 Surveys 276
G.2 Source Code 276

Appendix H: Accompanying Resources 277

المنارة للاستشارات
www.manaraa.com

List of Figures

1.1 A typical conventional configuration for software development. 2
1.2 A software engineering scenario using CAISE-based CSE tools. 7

2.1 The lifecycle of software development [122]. 11
2.2 A typical revision history tree for a software project. 13
2.3 Artifact modifications within a software project. 14
2.4 The cost of correcting errors across the development lifecycle [102]. 19
2.5 Perceived evolution of software engineering. 21
2.6 The disciplines associated with CSE. 23
2.7 The MAUI Groupware toolkit [55]. 26
2.8 Microsoft's SharePoint desktop collaboration system [71]. . . . 27
2.9 A visualisation of class cohesion using the JST pipeline [60]. . 30
2.10 The Poseidon collaborative UML tool [11]. 32
2.11 The Rosetta web-based collaborative design document tool [48]. 33
2.12 The Moomba collaborative XP development environment [92]. 34
2.13 The Tukan collaborative code editor [100]. 35
2.14 A graph editing tool within the Eclipse Communication Framework [66]. 35
2.15 Borland's JBuilder IDE with pair-programming capabilities [12]. 36
2.16 The Augur inspection tool [41]. 37
2.17 The Palantír collaborative visualisation tool [98]. 38

3.1 An example of the Action/Reaction candidate pattern of CSE. 47
3.2 The CSE patterns map. 56

4.1 A UML class diagram for a simplistic semantic model of software. 79
4.2 The combined code neighbourhood for two developers, using UML notation. 84
4.3 Media Richness Theory: reducing ambiguity by media selection. 92

المنارة للاستشارات
www.manaraa.com

4.4 The collaborative spectrum of software engineering. 93

5.1 A general schematic representation of the CAISE framework. . 99
5.2 Artifact modification within the CAISE framework. Internal to the framework is a constantly-updated semantic model, which represents the authoritative structure of the software project, and is used to provide accurate, fine-grained feedback information to participating tools. 101
5.3 The CAISE framework in the context of the collaborative spectrum. 104
5.4 An illustration of the CAISE framework and participating tools. 105
5.5 Relationships between key components of the CAISE framework. 106
5.6 A semantic model of object-oriented software [60], in UML notation. 108
5.7 The three conceptual layers of the CAISE framework. 112
5.8 The CAISE event model. 115
5.9 Key types of actions within the CAISE framework. 117
5.10 Schematic view of an artifact modification within CAISE. . . . 119

6.1 The CAISE *User Tree* widget, supporting a user-centric project view. 134
6.2 The CAISE *Change Graph* project management widget. 135
6.3 The CAISE *Users Pane*, providing voice and text communication. 135
6.4 The CAISE *Build Pane* with adjustable levels of collaborative awareness. 136
6.5 The *Talk Button* CAISE collaborative widget. 137
6.6 The CAISE *Collaborative Text Pane* with remote highlighting and telecursors. 137
6.7 The recommended threading model within a CAISE-based tool. 139
6.8 Initialising a CAISE-based CSE tool. 145
6.9 Adding widgets to a CAISE-based tool. 146
6.10 Downloading a CAISE artifact. 147
6.11 Downloading the semantic model and an auxiliary artifact. . . . 148
6.12 Implementing a key listener within a collaborative text editor. 149
6.13 Sending a local tool action event to the CAISE server. 150

المنارة للاستشارات
www.manaraa.com

6.14 Modifying an auxiliary artifact. 151
6.15 Processing events thrown by the CAISE server. 152
6.16 Updating the local view of the Java code editor based on framework events. 153
6.17 Updating the view for the UML class diagrammer. 154
6.18 Implementing a custom feedback plug-in. 155
6.19 Implementing a tool manager plug-in. 157
6.20 A CAISE-based collaborative code editor for Java. 159
6.21 A Decaf collaborative code editor. 161
6.22 A code age collaborative text editor. 162
6.23 The event sequence for updating a code age display. 163
6.24 A collaborative UML class diagrammer. 164
6.25 A Decaf collaborative code editor. 166

7.1 Visualising CAISE event log data. 176
7.2 Treemaps showing events in a CAISE session. 177
7.3 Temporal analysis of user activity within a CAISE-based project. 178
7.4 Artifacts accessed within a CAISE-based project. 179
7.5 The graphical interface of the program under modification during the evaluation sessions. 182
7.6 Mean task completion times for within file tasks. 184
7.7 Mean task completion times for between file tasks. 184
7.8 Lines of code versus server memory usage. 190

8.1 A typical revision trunk for a collaborative software project. . 195
8.2 A syntax-directed code repository interface. 196
8.3 A simplistic example of a well partitioned software project. . . 198
8.4 Various configurations for group work using CSE tools. 199
8.5 The various modes of collaborative view when compiling from within a CAISE tool. 201
8.6 The likelihood of build failures: collaborative versus conventional modes of work. 202

A.1 The CAISE server architecture. 231
A.2 Typical role of a CAISE-based parser. 232

المنارة للاستشارات
www.manaraa.com

A.3 User positions within a CAISE document buffer. 234
A.4 A server application which inspects the semantic model of a CAISE-based software project. 237
A.5 The code listing for a simple server application. 238
A.6 The plug-ins configuration panel of the Project Manager Panel. 240

B.1 An example source file for the Decaf language. 245
B.2 The full grammar for the Decaf language. 246

E.1 The Together Architect IDE operating within CAISE. 257
E.2 The Together IDE operating alongside a CAISE-based editor. . 259

F.1 Evaluation layout of CSE tools experiment. 263

المنارة للاستشارات
www.manaraa.com

List of Tables

2.1 Feature matrix of existing CSE tools. 39

3.1 Synchronous versus asynchronous development: typical tasks within each quadrant. 50

5.1 Event types within the CAISE framework. 116

6.1 Key methods of the CAISE tool API. 133

7.1 Summary of the subjective measures for tasks: NASA-TLX workload ratings. Possible values range from 1 (low) to 20 (high). 185

7.2 Summary of the subjective measures for overall preference. Possible values range from 1 (low) to 20 (high). 185

7.3 Post-session user comments. 186

F.1 Task types, tool modes and order of tasks. 266

للاستشارات المنارة
www.manaraa.com

Chapter I

Introduction

Software Engineering (SE) is undoubtedly a collaborative process. Developers within a SE project work together during all phases of the software development lifecycle.

In any discipline, teamwork is difficult due to the need for constant communication and coordination of tasks. Collaboration in software development teams is further complicated by the task of maintaining a range of products, each with multiple versions.

SE tools typically have poor support for collaboration. Instead of being designed upon collaborative processes central to SE, tools are based on a single-user view of the development lifecycle.

Common problems that current SE tools fail to address include *transactional conflicts* where concurrent changes to the project conflict with each other semantically, and *merge conflicts* [70] where concurrent changes to the same source file conflict lexically. Both of these problems stem from poor awareness of other users' actions, and an inability to synchronise the work of developers at a fine-grained level.

These problems are worsened by the typical copy, modify and merge idiom of conventional source code management systems [9], where long intervals of isolated development are common, increasing the difficulties of the source code integration and build process. A typical configuration for conventional software development is presented in Figure 1.1, where notification of program modifications is facilitated predominantly through source code repository systems, and tool support for communication between developers is low.

It is argued that the better communication and collaboration is supported, the better the SE process will be in terms of productivity and quality

www.manaraa.com

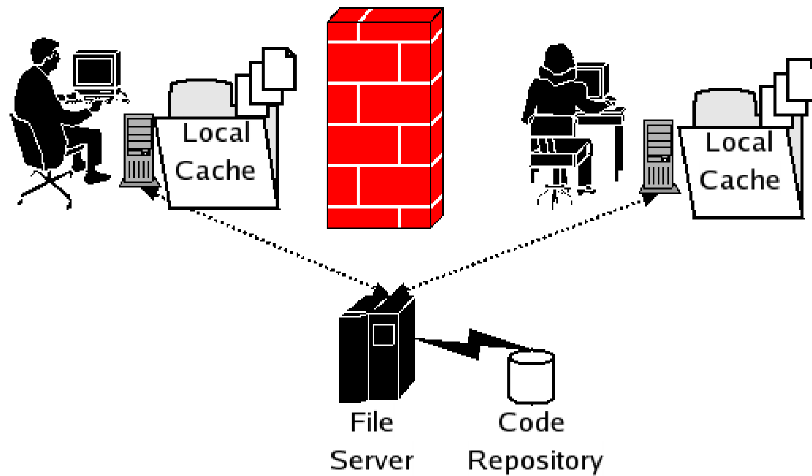


Figure 1.1: A typical conventional configuration for software development.

of the final software product [99]. However, as the field of Collaborative Software Engineering (CSE) is in its infancy, many of these claims are yet to be verified empirically. Only once quality real time tools exist will researchers be able to realise the full potential and benefits of CSE.

Support for collaboration, both synchronous and asynchronous, exists in other fields not directly related to SE. Within an office environment, it is possible to share and collaboratively edit documents with fellow workers to some extent. Teleconferencing facilities are also commonplace today, as are real time virtual meetings where shared virtual whiteboards are used to communicate ideas within a group of distributed personnel.

The successful application of asynchronous and synchronous collaboration within other fields of research suggests that the practice of SE can also become more collaboration-centric. Unfortunately, tools to support CSE are difficult to design; SE involves aspects that are challenging to accommodate using current Computer Supported Collaborative Work (CSCW) technology. In particular, SE is based around numerous complicated artifacts such as source files, where the document syntax is expansive and rigid, and many relationships between artifacts exist. Adding CSCW features to existing single-user tools, while an intuitive first step towards new CSE tools, does not necessarily scale or provide the level of improvement envisaged.

www.manaraa.com

As an example of typical CSCW challenges within CSE, what should be done when a user is half way through typing a method body into a text editor and a user in a UML diagrammer renames that particular method? Are all code changes lost? If not, how can the code change be preserved? It is not surprising, therefore, that most collaborative features for both commercial tools and research prototypes restrict collaboration to pair-programming and token-passing floor control mechanisms.

Any CSE tool of a realistic scale has complex issues to address, such as user interface design, CSCW floor control and management, varying levels of collaboration requirements, varying expectations between developers within a group, support of multiple artifact types, and potentially multiple views of artifacts. There are also technical aspects to address such as concurrency control and distributed system design, along with the standard SE technicalities such as parsing, semantic modelling and source code management. Accordingly, only a few research prototypes such as Poseidon for UML [11] have evolved into professional tools.

While it is certainly possible to implement collaboration-enhanced SE tools, the single significant barrier to the success of tools may be the poor ratio of tool power to development effort. Even once a good quality CSE tool has been developed, there is no guarantee that it will gain widespread adoption due to the varying requirements of software engineers; this mistake has been made within related areas of CSCW research [19].

The purpose of the research presented in this thesis is to investigate mechanisms to support real time CSE, and to determine the benefits for software engineers when such tools are used. As a secondary objective, this research is aimed at reducing the barrier of high CSE tool construction costs by providing a framework that enables many new types of CSE tools to be developed rapidly. The research premise is that given a framework to support challenging aspects of tool construction such as group management, artifact sharing and semantic analysis of source code, it is more feasible to develop tools in order to explore new aspects of CSE.

The research in this thesis focuses on creating a framework and prototype widgets for CSE tools. The framework, Caise (a Collaborative Architecture for Iterative Software Engineering), supports the development of

المنارة للاستشارات
www.manaraa.com

CSE tools that operate both synchronously and asynchronously. These new types of CSE tools can be designed to avoid the problems associated with conventional approaches to software development by increasing programmer communication and awareness of others' actions.

The CAISE framework allows isolated programmers to work collaboratively without sacrificing communication. CAISE-based synchronous CSE tools achieve this by keeping all programmers within a group synchronised in real time, at the same time providing customisable user awareness and project state information to individual tools. The CAISE framework is best suited to a small group of developers who wish to work collaboratively and in close contact on an entire software project.

The CAISE framework provides an infrastructure with the potential to support the entire SE process. CAISE-based tools can be constructed that provide more than just shared editing of basic software artifacts. Collaborative compilation, testing and debugging of software projects is also possible to implement using the services of CAISE. Comprehensive inter-developer communication facilities can also be constructed.

Many tools have been produced by way of this framework, including collaborative editors and diagrammers, reusable CSE widgets that can be added to any existing SE tool, new types of user activity visualisations based on fine-grained logging and project structure information, and custom CSE tools such as real time project management agents and observers.

1.1 An Example Collaborative Development Scenario

To illustrate the concept of the CAISE framework, an all-too-common SE scenario is presented. First, an example is given of a coding conflict and resolution between two developers using conventional tools. This is followed by another example of the same scenario, but this time with the support of CAISE-based CSE tools.

1.1.1 Background

Bob and Alice are working on the same project, developing a graph editor with a simple user interface written in Java. The user interface code is

المنارة للاستشارات
www.manaraa.com

contained within one file named `GUI.java`, and the code to perform file saving is in a file called `Persistence.java`. Two tasks require completion: replacing the `save(int fileType)` method call with a call to the method `saveXML()` from the file `GUI.java`, and adding a new method named `save()` to `Persistence.java` that saves the current file using the system default format.

1.1.2 Conventional Tools

Using conventional text editors and a code repository such as CVS, Bob and Alice will both take separate copies of the current version of the code from the repository (version 1.1, for example) and start working independently. Alice has chosen to edit the GUI code, and she replaces the `save(int fileType)` method call with `saveXML()`. At that point, Alice re-compiles her code, tests her working copy of the program to ensure that it operates, and then checks her files back into the CVS repository. This marks `GUI.java` as version 1.2.

At the same time, Bob starts working on his task of adding the new `save()` method to the file `Persistence.java`. Not only does he complete this method, but he also deletes the existing `saveXML()` method, as according to the current code base (version 1.1), no calls to this method are made. Bob re-compiles his program, and it compiles and runs without error. He then checks his files back into the repository, which marks `Persistence.java` as version 1.2.

Both Bob and Alice then leave for the day, knowing that they have successfully made improvements to the latest version of the graphing tool. Unfortunately, when they arrive to work the following day, they are advised that the nightly rebuild from the code repository failed, due to an unresolved call to the `saveXML()` method.

After scheduling a meeting, Bob and Alice realise the source of the problem that they have unintentionally created. To resolve it, they decide that to save in XML, the new `save()` method is to be called after setting the system file format to XML. Therefore, Alice checks out the entire code base again (version 1.2), changes the call in `GUI.java` from `saveXML()` to `save()` after specifying the system file format, and re-compiles the program. After

المنارة للاستشارات
www.manaraa.com

verifying that the program works, `GUI.java` is checked back into the central repository as version 1.3, and the project is now up-to-date.

When working very closely with each other, Bob and Alice might immediately notice that their given tasks have the potential to conflict unless care is taken, and that some negotiation is necessary to successfully modify the existing code base. In normal coding practice, however, conflicts of this nature are common, particularly as the number of concurrent developers increases. The protocols that the programming team employ, such as frequency of code integration and degree of communication during development, govern the number of conflicts encountered and the level of effort required to merge conflicting modifications.

1.1.3 Collaborative Tools

Using Caise-based real time CSE tools for the above task, developer interaction can be very different. Even though Bob and Alice could set out working on separate tasks using different types of development tools, as soon as Bob and Alice place their cursors in code that is semantically related, the Caise framework will send notification to their tools that a semantic relationship exists. A Caise-based CSE tool that responds to this type of feedback is illustrated in Figure 1.2.

Using the current coding scenario, when Bob moves his cursor over the `saveXML()` method with the intention of deleting it, his tool will be able to inform him that this method is now called from a method within the file GUI.java, which is currently being edited by Alice. This information is shown in the feedback panel, presented in the lower half of Figure 1.2. Even if Bob did choose to delete this method, both he and Alice will be notified immediately that the program has just been broken due to an unresolved method call.

This is only a simple example of the capabilities of Caise-based tools. Chapter 6 presents in more detail some typical Caise-based tools which provide many different types of feedback mechanisms, such as telecursors, real time metrics, chat facilities, and widgets that indicate the current state of the project and its associated artifacts such as source files. In the above

www.manaraa.com

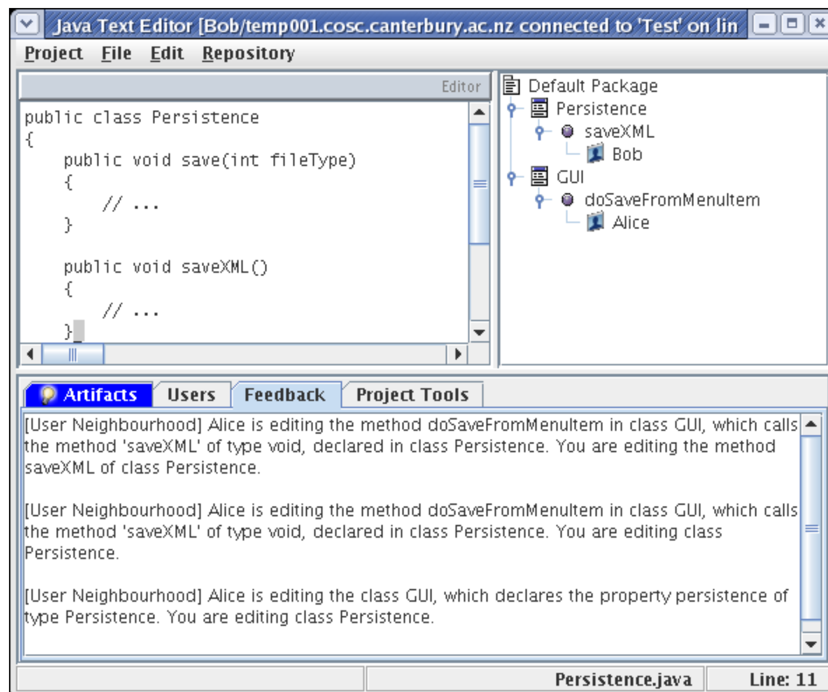


Figure 1.2: A software engineering scenario using CAISE-based CSE tools.

simple scenario, a clear illustration is given of the fundamental premise of the CAISE framework: through real time monitoring of user activity and semantic analysis of the underlying software, it is possible to provide collaboration support far richer than currently available within conventional SE tools.

1.2 Research Goals

The areas of SE, CSCW and distributed communication are well-researched. Synchronous CSE, however, is an emerging field of research, and the body of knowledge is relatively small. At the time of writing, no holistic frameworks to support CSE at a fine-grained level have been demonstrated, either at a theoretical or practical level. The motivation for the research presented in this thesis, therefore, is to provide such a framework that supports the development and run-time requirements of CSE tools, and to evaluate example tools that the framework can typically support.

المنارة للاستشارات
www.manaraa.com

The research components of thesis are:

- A background study of CSE and related areas of research
- Identification of primary design considerations for CSE tools, including the identification of candidate patterns of CSE
- The presentation of a framework for CSE tool construction that supports the identified patterns of CSE
- A demonstration of several tools to support CSE, based upon the Caise collaborative framework
- Evaluation of the Caise framework and associated CSE tools

1.3 Thesis Outline

In Chapter 2, an overview of work towards CSE is given. Existing CSE tools are reviewed, as well as supporting technologies such as Groupware, distributed systems and configuration management. In Chapter 3, patterns of collaboration related to SE are discussed. In Chapter 4, the requirements for CSE tools to support such patterns of collaboration are presented.

In Chapter 5, the Caise framework is discussed and a full description of the Caise framework is given, including implementation details. This chapter also includes a discussion of how the facilities provided by the underlying framework support CSE tools. In Chapter 6, the application of Caise-based tools is illustrated. Demonstrations of how different tools are used within collaborative development settings are provided. This includes a discussion on how CSE tools are developed within the Caise framework.

In Chapter 7, an evaluation of the Caise framework and supporting tools is presented. In this chapter, heuristic evaluations for such tools are also introduced. In Chapter 8, open problems for the Caise framework are discussed. This chapter also discusses how the Caise framework can be used within an industrial context.

Final conclusions are made in Chapter 9. Directions for future work are provided, both for the Caise framework and associated collaborative tools.

المنارة للاستشارات
www.manaraa.com

Chapter II

Background

In this chapter, the background related to CSE is presented. In Section 2.1, the process of SE is discussed, including common methodologies and developer roles. The significance of collaboration within SE is addressed in Section 2.2. A definition of what the field of CSE encompasses is given in Section 2.3, including a description of what CSE tools aims to provide.

Research closely related to CSE, such as distributed computing and Groupware systems, is summarised in Section 2.4. A detailed discussion on previous work towards support for CSE is presented in Section 2.5. This chapter concludes in Section 2.6 with a description of current barriers to the support of CSE, including an outline of the key difficulties in implementing CSE tools.

2.1 The Process of Software Engineering

This section outlines the process of SE as a prelude to Chapter 3: Patterns of Collaboration.

2.1.1 Overview

SE is the result of a maturity within the field of software development. The seminal point of this maturity was the 1968 NATO conference [78], where a progression from unorganised, unstructured hacking to well-planned engineering during all phases of development was promoted.

Around this time, far too many projects were failing as the size and scopes of projects increased along with the complexity of source files and related libraries. From Dijkstra's ACM Turing Award lecture, an analogy is made for the necessity of SE processes [35]:

www.manaraa.com

> "As the power of available machines grew by a factor of more than a thousand, society's ambition to apply these machines grew in proportion, and it was the poor programmer who found his job in this exploded field of tension between ends and means. The increased power of the hardware, together with the perhaps even more dramatic increase in its reliability, made solutions feasible that the programmer had not dared to dream about a few years before. And now, a few years later, he *had* to dream about them and, even worse, he had to transform such dreams into reality! Is it a wonder that we found ourselves in a software crisis?"

Edsger W. Dijkstra,
1972

SE specifies how to approach the development of large projects. This includes aspects such as the development methodology followed, the gathering of project requirements, the specification of appropriate test cases and user acceptance tests, deadlines and project milestones, contingency plans, and the design and development phases. While no SE methodology can guarantee a successful outcome, adhering to the accepted codes of best practice helps minimise the effects of factors such as changing project requirements, incomplete specifications and unforeseen development delays.

The Software Development Life-Cycle

A powerful model to abstractly represent the process of software development is presented by Zelkowitz, Shaw, and Gannon [122] in Figure 2.1. This model shows how SE takes a real-world problem, derives a solution based on abstract analysis, and delivers a final and tangible working product in the form of software.

As Zelkowitz's model is a general one, it does not make reference to the specific development factors that must come into consideration when engineering software. Core factors of any SE project include source files and versioning, programming languages used, product versions and branches, teams for each development phase, and the development methodologies followed. These aspects are discussed further within the context of CSE in Chapter 4.

Evolution within SE processes is becoming increasingly important. Due to large frameworks and libraries, component-based SE, and programming

المنارة للاستشارات
www.manaraa.com

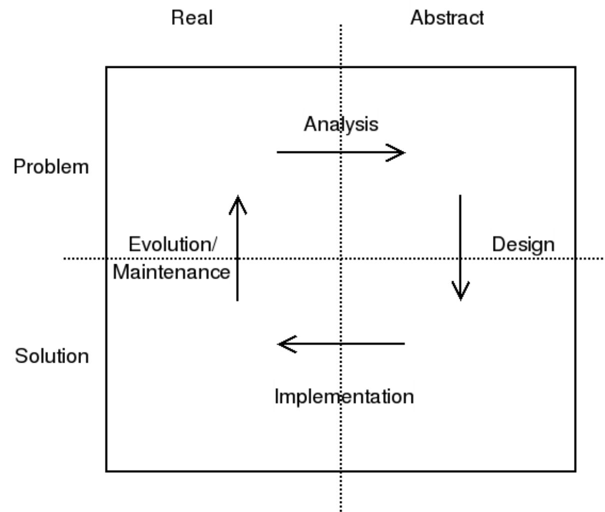


Figure 2.1: The lifecycle of software development [122].

languages that encourage reuse and 'programming by difference', it is commonplace for applications to be refactored, redesigned, reused and even merged with other products. Figure 2.1 has therefore been enhanced with an additional evolution phase, which indicates that implementations may simply represent the end of one iteration of the cycle.

Some types of SE artifacts are specific to certain phases of Zelkowitz's model. For example, requirements documentation is typically formed exclusively during the analysis phase, as it is likely to be used only as reference material for the remaining phases. For other artifacts such as class diagrams, they might be used for several or all phases of the model during any given iteration.

2.1.2 Software Engineering Methodologies and Processes

Many SE processes and methodologies exist and are in mainstream use today. Processes range from the more prescriptive, such as the Waterfall process [96], to the highly adaptive, such as eXtreme Programming (XP) [7]. Hybrid approaches also exist, such as the Rational Unified Process (RUP) [62], where both prescriptive and adaptive practices are followed. While all currently used processes predate CSE tools, the XP process does accommodate pair-

المنارة للاستشارات
www.manaraa.com

programming, where groups of two programmers work together on the same input device.

As well as processes and methodologies, many types of programming languages are in use today. These include procedural programming where the program follows a predefined sequence of instructions, Object Oriented (OO) programming where small units of code react to events, and functional programming where complex results are obtained through the chaining of multiple function calls.

The processes and types of programming languages employed within a project directly affect the software in various ways, including the ability to adapt to changing requirements, the degree of clarity when determining project milestones, and the partitioning of development teams.

2.1.3 Software Engineering Artifacts

Artifacts are central to the SE process. Here, a discussion is presented on how collaboration during artifact modification is supported by conventional SE tools and practices.

Project Branches and File Versions

For any realistically-sized software project, it is likely that multiple versions of its founding source files are stored within a source code repository system. This is regardless of the number of programmers employed or development methodology followed. By branching, as illustrated in Figure 2.2, minor modifications to files within a previous release of an application can be made immediately upon request, regardless of the compilation state of the current version of the project. Such changes can then be integrated into the main product trunk once the main version is in a compilable and stable state, instead of attempting to rush the construction of the current project version which also incorporates the newly requested modifications.

Branching, where a complete set of project files is duplicated for an alternate stream of development, does not necessarily have to be undertaken within a software project. Within a single development trunk, however, source code repository systems will still checkpoint sequential versions of

المنارة للاستشارات
www.manaraa.com

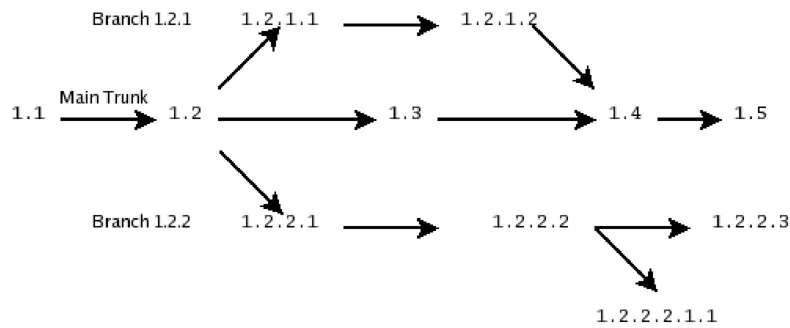


Figure 2.2: A typical revision history tree for a software project.

all project files; typically this is done automatically upon each commit of modified files back into the central repository. Accordingly, it is important to realise that source code repository systems have the ability to produce a previous version of any file, possibly replacing the current version if required.

Given that multiple branches of a project may exist, and that each file within a branch has a potentially large number of previous versions, the focus is now turned to the lifecycle of a file in the context of a single version.

File Merging

Any given version of a single file may be modified by several developers concurrently. When using conventional SE tools, files are typically shared through the idiom of copy, modify and merge [9]; each user obtains a copy of the current version of the file, makes their modifications, and once all changes have been made, the set of modified files are merged into one single newer version.

A single version of a file may undergo complicated concurrent changes. Using Figure 2.3 to illustrate, a change in area A poses no likely modification problems. For regions B and C, however, it is highly likely that any concurrent changes will conflict when merging takes place. Even if the syntax of the two changes do not directly interfere with each other, it is likely that the lexical closeness of these changes will still cause a *merge conflict*, where a new version of the file is not able to be formed automatically.

The process of file merging is normally performed on a character–by–

www.manaraa.com

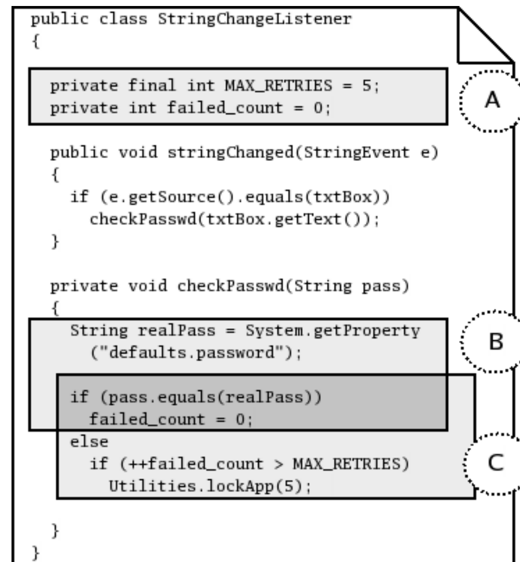


Figure 2.3: Artifact modifications within a software project.

character basis, where no effort is made to analyse the syntax or semantics of the modified source files. Once concurrent modifications for a source file have been merged into a new version, the resultant file is committed into the main repository and distributed to all developers.

Merging is not a trivial process, however. Even the very latest automatic file merging tools fail to resolve all but the most simple of file merging tasks [70]. This leaves the task of correcting changes that conflict to the developers, which can be a painstaking and time consuming process.

Transactional Conflicts

It is evident that the lifecycle of any given artifact within a project is complex. Independent of within-file merge conflicts, a further problem exists during the concurrent modification of a set of files. This issue, which can be termed a *transactional conflict*, occurs when a modified file, while syntactically valid, causes the project build to inevitably fail due to a broken code dependency.

An example of a transactional conflict, where the project became unintentionally incomplete at a semantic level, was presented in Section 1.1. Within conventional tools, users will not detect transactional conflicts until all the

المنارة للاستشارات
www.manaraa.com

new versions of the modified files are merged with the central repository and a project rebuild is attempted.

Transactional conflicts can be considered at the scope of project level rather than at file level. When discussing artifact lifecycles, however, it is important to realise that transactional conflicts have an impact on artifacts—the artifacts involved require subsequent modification to enable the project to build again, even if this means reverting to a previous version of the file.

2.2 The Significance of Collaboration

SE is undoubtedly a collaborative process. It involves the concurrent editing of multiple artifacts by many teams of software engineers through a series of development stages, across multiple product lines. The artifacts of software, such as source code, will be heavily shared during the project's design, implementation, testing, refactoring and maintenance.

2.2.1 Collaboration Defined

Before discussing the significance of collaboration in detail, a definition of collaboration within the context of this thesis is defined. For many SE purposes, collaboration is a term that can be used loosely to indicate any form of interaction between related software producing organisations. In terms of this thesis, however, CSE is defined as the practice of developing a specific software product within a team of closely related developers.

Working collaboratively, therefore, encompasses regular meetings, division of labour, regular code integration, and ongoing communication and interaction between programmers during the development lifecycle. Developers within a project can be co-located or distributed, but they will work on the same set of artifacts—possibly the same artifacts at the same time.

2.2.2 Project Life-Cycles

The nature of project lifecycles implies that human interaction is necessary and unavoidable, both within and between groups of developers. The leverage of expertise from others is important, which is normally facilitated by face-to-face meetings. Regardless of the location of individuals, all developers must

المنارة للاستشارات
www.manaraa.com

communicate with each other on a regular basis to communicate, coordinate and collaborate in order to analyse, design, implement and test the software under development.

The ability to collaborate during software development is essential even when working on a well-structured single-person project. Collaboration does not just involve connecting people; collaboration between files, versions and tools is equally as important. A programmer in isolation can still benefit from tools that understand the actions of other tools, and recognise the inter-relationships within constantly-changing source files and other artifacts. The ability to collaborate between tools, even by a single developer, is becoming increasingly important as the size and complexity of software projects grow.

Within team development, groups of software engineers need to be able to share files, modify them, re-integrate them with the main source code repository and resolve any errors that result from concurrent conflicting changes. To avoid significant complications when re-integrating code with the global repository, each developer needs to be aware of the impact that his or her changes will have on other parts of the system. Particular attention must be paid to the parts that are being edited by other developers at that point in time.

To assess the impact of changes before they happen, the actions and intentions of others must be identified. The sooner these aspects are made known to the developer the better; if programming conflicts between developers can be detected early they can be resolved early [99], or avoided altogether.

2.2.3 Conventional Support for Collaboration

At present the support for collaboration within SE is limited. Conventional SE tools appear to be designed around the premise that developers are co-located, well aware of the current actions of other programmers, and use social protocols such as source file ownership and regular integration periods to prevent significant modification conflicts [51].

Unfortunately, SE practice today often falls beyond the desired characteristics listed above. Developers are often distributed rather than co-located, are not constantly coordinated with other users within the group, and often

المنارة للاستشارات
www.manaraa.com

work on copies of the same source files concurrently—particularly within the open source community where developers have a great deal of freedom in the construction of software.

Face-to-face communication is the richest way to communicate with others [103]. Accordingly, within a close team of co-located developers, face-to-face communication on a daily basis is both desirable and commonplace. For development scenarios where face-to-face communication is not possible, mailing lists and email correspondence are typical substitutes [51].

Face-to-face meetings occur within most co-located software development teams on a daily basis, regardless of the SE process followed. Such meetings are typically held to discuss problems resulting from the nightly build, or perhaps to assign tasks to pairs within the XP process. Conference calls within distributed teams is common practice, typically upon completion of milestones or immediately prior to a critical phase such as a re-division of labour. Instant messaging between developers regardless of physical location is also commonplace today for low-level interaction, such as determining the current activities and intentions of co-workers.

For very well coordinated teams, communication through electronic means and careful use of a code repository system to implement file sharing has been shown to produce some successful projects [51]. There are, however, many trade-offs to this approach including the need for expert software engineers, a slow product delivery rate, a closed development group and a long learning period for any newcomers accepted as code contributors.

For most development groups, however, it is problematic deriving information about the actions and intentions of others from single-user development tools, email correspondence and mailing lists. Source control systems, such as CVS, provide some features to alert users to conflicting actions such as a file being checked-out by other developers. The authors of CVS state however: *"CVS is no substitute for communication"* [9]. This is particularly true for non-expert groups, where developers often spend large amounts of their time addressing problems that are only exposed during the code integration stage.

المنارة للاستشارات
www.manaraa.com

It appears that communication and collaboration are unduly restricted within most teams of software engineers. This is evident from the integration problems common to most projects as documented previously [86]. The research within this thesis is based on the perception that the current level of support for collaboration within conventional tools is unacceptably low.

The research in this thesis favours the approach of detecting errors as early if possible—to the point where they may in some circumstances be avoided altogether—which is not possible with conventional SE tools. Furthermore, the current repository-based model of copy, modify and merge is troublesome; conflicting changes made by others are only exposed upon code integration, and feedback information is restricted to code repository response codes and build reports.

It is possible to envisage CSE tools where the facilitation of discussions is commonplace, and such discussions are initiated as changes are being made. At present, this is difficult to achieve due to very limited tool support; conventional tools can provide feedback related to the current developer's modifications, but they do not account for any changes being made concurrently by other users within the project.

Figure 2.4 illustrates the cost of correcting errors as they occur during different phases of the development lifecycle [102]. The vertical axis on this figure represents the relative cost of correcting errors per phase, which could be measured in dollars or man-months. More recent research of contemporary software development concludes that "uncorrected defects become exponentially more costly with each phase in which they are unresolved" [118]. Given that it is important to discover errors as early as possible, again it is apparent why tool support at present is inadequate—small but critical changes are often not discussed until problems are exposed during final testing, and current code integration practices mean that even easy-to-detect errors such as merge and transactional conflicts may go unnoticed for days.

المنارة للاستشارات
www.manaraa.com

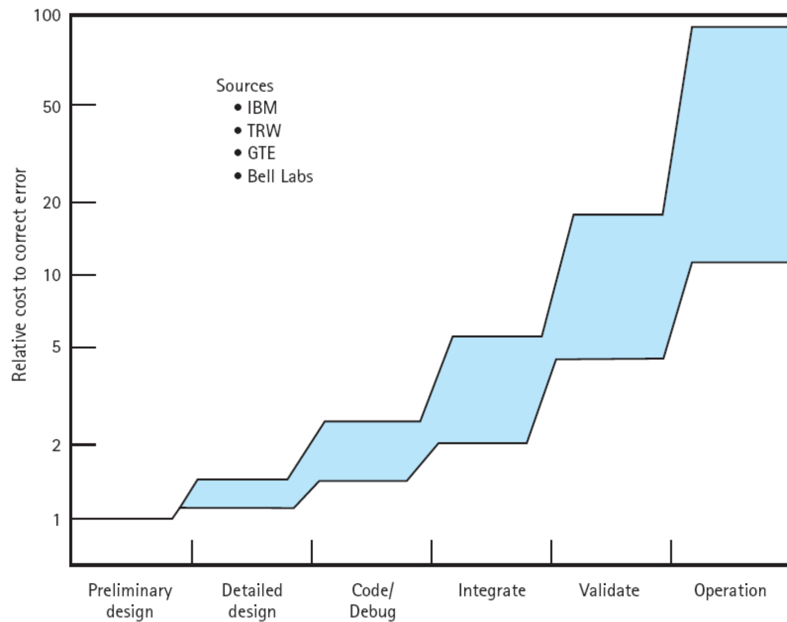


Figure 2.4: The cost of correcting errors across the development lifecycle [102].

2.2.4 The Progression of Software Engineering

To support the progression of SE, tools must support higher levels of code understanding. Without such support, development is likely to become stifled as programs and programming languages become even more complicated and inter-related.

Projects will continue to rise in complexity as software becomes increasing heterogeneous and libraries continue to add more globally accessible, complicated classes. While many professional Integrated Development Environments (IDEs) provide comprehensive support for code and library browsing within a single instance of a set of source files, no facilities are dedicated to the analysis of the relationships between developers as they work within their project work-spaces.

To increase programmer productivity, the code repository idiom of copy, modify and merge must be replaced with a more efficient means of file sharing, particularly in cases where this idiom is used as a basis for communication between developers. Even if developers work on separate source files during

المنارة للاستشارات
www.manaraa.com

a development phase, the isolated nature of current software development tools implies that conflicting changes between files are still likely to be made, and will not be detected until the integration stage.

The progression of SE is the motivation behind the research of this thesis. A possible path of progression for SE is presented in Figure 2.5. Tools are proposed that operate on a shared semantic model of software, where modifications are made to the corresponding single, shared and central repository of artifacts in real time. Given such a shared semantic model and real time access, it is possible to immediately calculate the impact of proposed or actual changes and provide user proximity information between sets of collaborating developers. Instead of making course-grained commits to a code repository, perhaps at the level of multiple files at a time after several days' worth of development, systems can be envisaged where code modifications are real time atomic operations without any need for duplication of artifacts.

Collaboration of this proposed nature is beginning to take place within the field of SE. The XP process of daily meetings and regular rotation of tasks to leverage and share knowledge provides a degree of collaboration unobtainable by most other development methodologies. Additionally, the XP practice of pair-programming supports the shared development philosophy promoted in this thesis, albeit restricted by the limited capabilities of current technology for collaborative programming.

Agile Methods [24] are also becoming increasingly popular within SE teams. The Agile process focuses on producing deliverables frequently by having groups of limited numbers that work on small units of development. This type of development process is well-aligned with the principles of CSE and their associated patterns of collaboration (as described in Chapter 3). Therefore, as the popularity of Agile Methods increases, so too will the demand for Agile Methods-aligned CSE development tools.

A key aim of the research presented in this thesis is to promote collaboration-based SE through better tool support. From the perspective of Agile Methods and XP, the research in this thesis may allow these team-focused processes to scale from small subsets of collaborative developers to a completely collaborating team. This can only be achieved, however, once enabling tools and technologies are available.

المنارة للاستشارات
www.manaraa.com

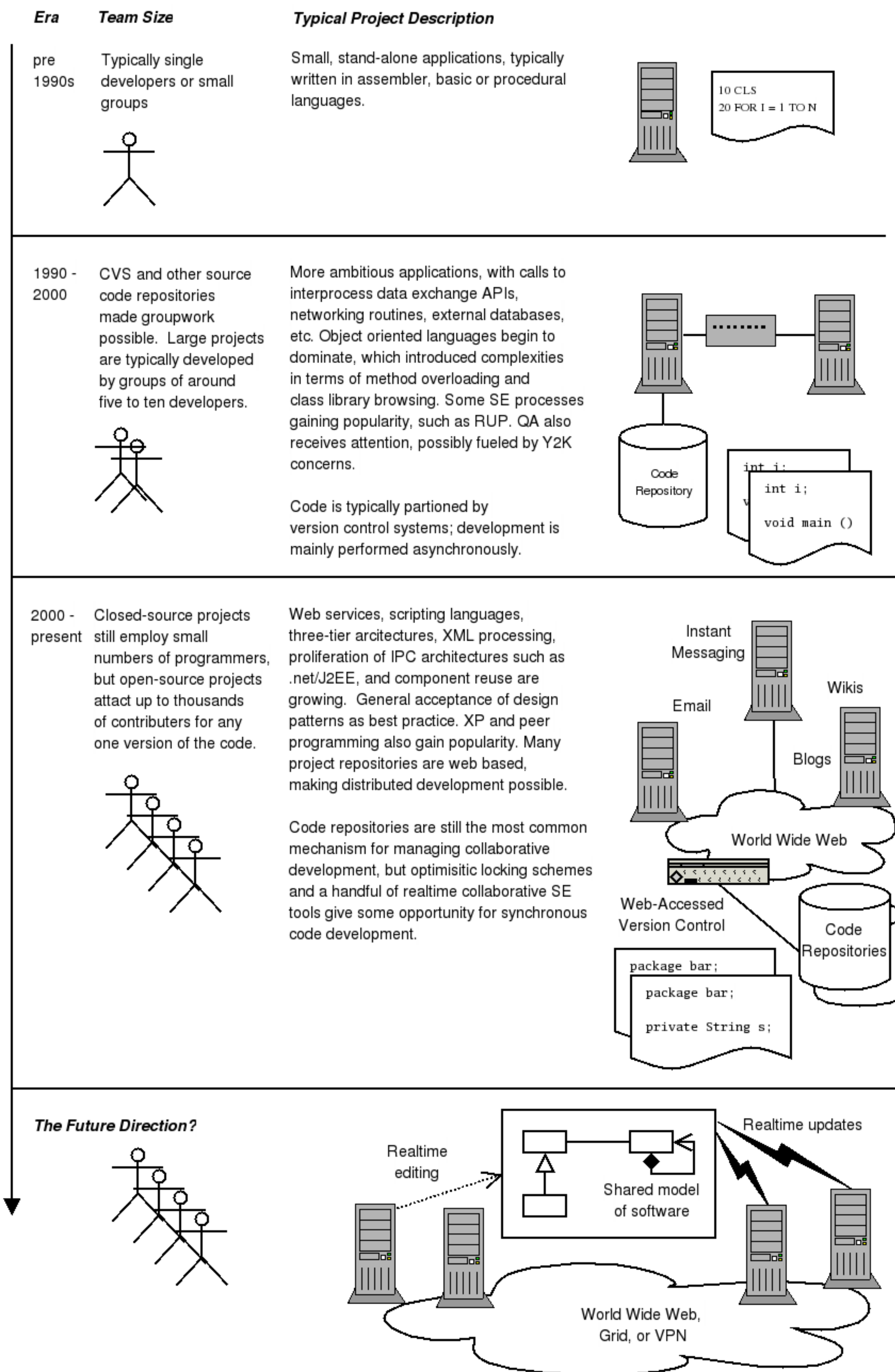


Figure 2.5: Perceived evolution of software engineering.

www.manaraa.com

2.3 Defining Real Time Collaborative Software Engineering

Almost all modern SE methodologies involve several distinct groups of people. They work on many artifacts with different types of tools, producing multiple versions of software products. SE is unavoidably collaborative. Accordingly, research into real time CSE investigates ways of lending computer-based support to assist programmer and tool communication, management of artifacts, and coordination of tasks.

Practical support for CSE at the moment is little more than tool support for CVS plus simple communication facilities such as email and instant messaging. As described previously, tools for CSE can be envisaged that have the potential to raise collaboration to a higher level. Such tools make the separation of users less obvious, giving an impression of working on one shared software project.

In defining CSE, research and development efforts should not be restricted only to real time collaboration. Often programmers will not work at the same time, due to other obligations or even time zone differences. Any CSE tool or architecture must be able to support both synchronous and asynchronous modes of development to be genuinely useful.

Similarly, the research in this thesis advocates the development of tools and architectures that allow CSE to be practiced in both co-located and distributed settings. Development by two programmers working on the same workstation should be supported if that is what a given methodology requires, such as pair-programming. Large degrees of physical separation should also be no obstacle for CSE. The ultimate goal of CSE research could aim to make collaborative work no different whether workstations are separated by an office partition or a hemisphere.

Within this thesis, real time CSE is defined as: *A field of research that investigates platforms and technologies to support the development and use of synchronous SE tools for a range of tasks, programming languages, SE processes, physical settings and team sizes. The goal is to explore and realise the perceived benefits and the full potential of CSE.*

المنارة للاستشارات
www.manaraa.com

Identifying Related Research Fields

After considering the main aspects of team development, it is apparent that for the successful implementation of CSE tools, knowledge of several related disciplines of computer science is required. While an understanding of core SE aspects is important, other topics that must be addressed include human factors and usability, source code control systems, distributed systems, Groupware and CSCW, and software visualisation. As illustrated in Figure 2.6, CSE forms an intersection of these overlapping areas.

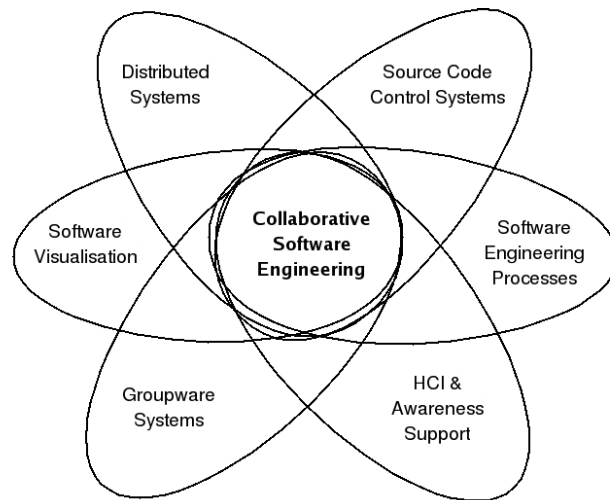


Figure 2.6: The disciplines associated with CSE.

2.4 Research Related to Collaborative Software Engineering

The following section presents a discussion on the areas of related fields of research identified within the diagram of Figure 2.6. A claim is not made that CSE research is at the core of each related discipline, but an assertion of the research in this thesis is that each of the above areas is key in supporting and enabling successful CSE tools and frameworks. References to pertinent papers from all related research fields will be made in the subsequent chapters of this thesis. For a comprehensive listing and further discussion, please refer to the annotated bibliography contained in the accompanying resources disc.

المنارة للاستشارات
www.manaraa.com

2.4.1 Software Engineering Processes

SE is a very practical area within the discipline of computer science. As such, the theories related to SE are often produced empirically by observation of practicing software engineers and the induction of facts, rather than by deduction and proofs.

Due to the large range of tasks within SE and the considerable amount of variance in methodologies, team dynamics and duties to be performed, it is difficult to perfectly map what happens within SE projects to a set of rules and processes. Consequently, many of the highly cited articles related to SE are chapters within books that provide an overview of an entire process, rather than scientific conference and journal papers that focus on highly specific topics.

There are many tools available to the software engineer. At one end of the spectrum, well designed simple tools, such as text editors, have widespread use and popularity, particularly for hobbyists or individuals working on small software projects. At the other end of the spectrum, more complex Computer Aided Software Engineering (CASE) tools that incorporate code editors, diagrammers, source code control interfaces, workflow components and debuggers are available. Tools and technologies of this nature, such as FIELD [94], PCTE [13], Eclipse [83] and Visual Studio [72] are often well suited to large teams of software engineers, where numerous procedures and methodologies that require tool support are likely to be in place.

Conventional Software Development Environments

Over the last two decades, many IDEs have been used within software development teams. Examples of popular environments today include Microsoft's Visual Studio [72], Borland's JBuilder [12] and Eclipse [83]. One of the earliest IDEs which is considered to be seminal within the field of CSE is FIELD [94], due to its ability to support numerous distinct SE tools via a clearly defined message passing interface. While not originally cited as a CSE platform, FIELD was one of the first projects to address tool interoperability and basic analysis of source code.

A similar research project was the Portable Common Tool Environment

المنارة للاستشارات
www.manaraa.com

(PCTE), which again defined interfaces for sharing data between SE tools and underlying components of the host operating system [13]. PCTE is based on relational databases, using fine-grained relational data models as the authoritative source of tool information interchange.

2.4.2 Groupware and CSCW

SE is a team activity; cooperation between individuals is essential. Groupware, the class of software that enables local and remote users to collaborate via networked computers, provides only limited support for SE. As the interaction between software engineers is often complex, and SE is predominantly artifact based, few computer supported cooperative tools for SE exist.

Groupware and CSCW are areas that clearly lend themselves to CSE tools and research. With Groupware technologies, user interfaces for simple tools can be replicated and shared in real time by multiple users. In theory, such technology can be directly applied to assist the development of multi-user SE tools.

Conventional applications constructed through Groupware are typically tools such as shared white-board editors, chat facilities, and map and graph browsers. The CSE researcher must be aware, however, that for complicated applications such as SE tools, Groupware has its limitations that are of particular concern for CSE tools. These limitations are discussed in Section 2.6.1.

Programmable Toolkits

Programmable Groupware toolkits are used to rapidly construct CSCW applications using common multi-user components. An example of such a toolkit, MAUI [55], is presented in Figure 2.7. Essentially, programmable Groupware toolkits are all that developers require to facilitate communication between a group of distributed applications unless highly specific networking or collaborative features are to be supported.

Typical components within Groupware toolkits include shared text editors, chat facilities, sketch-pads, voting facilities and mechanisms to support group membership, connection and disconnection. Some programmable

المنارة للاستشارات
www.manaraa.com

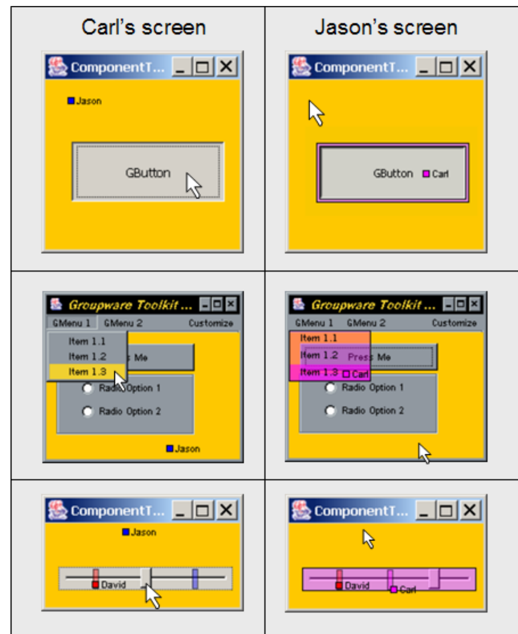


Figure 2.7: The MAUI Groupware toolkit [55].

toolkits, such as GroupKit [95], also provide comprehensive floor control policies; for the remainder, all concurrency control must be implemented by the application developer.

Desktop Systems

Desktop CSCW systems are designed to support general work-flow and collaboration without the need for customisation or special tool development. They are monolithic systems that allow common applications, documents and data sets to be shared. A key distinction between desktop systems, such as Lotus Notes, and other types of Groupware is that desktop systems do not normally support synchronous collaboration.

An example of a desktop system is Microsoft's SharePoint Server [71], as illustrated in Figure 2.8. In this figure, a Word document is being edited by a member of an office team, with options to check it back in to a document store, receive alerts when others edit it, and retrieve previous versions. The file sharing mechanism employed is very similar to CVS.

www.manaraa.com

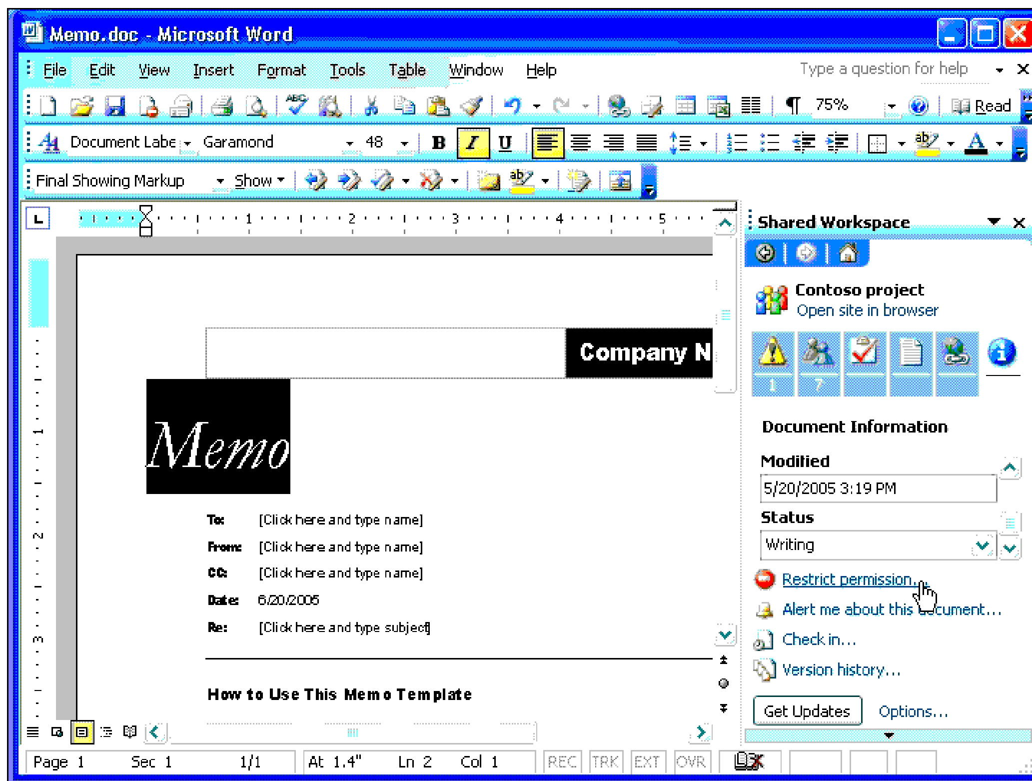


Figure 2.8: Microsoft's SharePoint desktop collaboration system [71].

Desktop systems are useful for general work-flow contexts such as document editing and project scheduling, but do not lend themselves directly to most SE tasks. This is because SE artifacts are highly constrained in terms of semantics, undergo frequent concurrent modifications throughout the team, and have many inter-relationships between documents. Desktop systems do however provide a good illustration of how CSCW technologies can be applied to facilitate computerised structured teamwork.

2.4.3 Source Code Control Systems

Source code control systems such as CVS [9] and SubVersion [25] are core to SE. These systems enable the versioning, branching and management of SE artifacts to ease the burden of producing multiple versions of software products derived from the efforts of potentially hundreds of developers.

المنارة للاستشارات
www.manaraa.com

Even for single user projects, the benefits gained from source code control systems, including the ability to roll-back changes, make the use of such systems warranted and valuable. Source code control systems also attempt to keep source files coordinated as they evolve by allowing regular check-ins and project builds. Without such facilities, it is possible for individual coding efforts to skew the project into several separate and hard-to-consolidate directions.

Source code control systems address the fact that many people may be working on the same code base, and that often several developers will want to work on the same source file. To facilitate controlled file sharing, two schemes are typically employed: file locking or file copying and subsequent merging—and both approaches have their advantages and disadvantages. Regardless of the issues surrounding the use of source code control systems, they are a fundamental component for most SE tools, both collaborative and single user.

2.4.4 Human Computer Interaction

The area of Human Computer Interaction (HCI) is active with large volumes of research papers being generated each year. Such papers typically present small-scale evaluations where problems are isolated into a simplistic form. This is a scientifically correct and well-accepted practice, but unfortunately for the purposes of CSE research, such studies can at times be too trivial. Research into SE requires an acceptance that tools are complex and artifacts are numerous, and that simplification can yield results that are not significant in 'real world' terms.

CSE tools can be more ambitious in design and harder to evaluate than the tools studied within HCI. Many HCI papers that address CSE only assess programming within isolated environments, such as spread-sheeting tools [77]. Therefore, the CSE researcher needs to be aware that HCI studies may not necessarily scale to realistic SE scenarios—often the papers are only useful for general guidance related to CSE experimental design.

The effectiveness of awareness mechanisms is an area of great importance to CSE tools. Some results from previous HCI studies will be useful for

المنارة للاستشارات
www.manaraa.com

CSE tool development, but CSE researchers will certainly have to extend the current base of HCI knowledge as CSE progresses.

2.4.5 Distributed Systems

In terms of providing facilities for interprocess communication, Groupware technology can offer basic distributed communication support. For the transport of application specific data, or for where more complex and efficient systems are to be supported, then a distributed systems technology may be the only answer to the support of CSE tools.

Distributed systems aim to make the boundaries between computers invisible, a term often referred to as global or ubiquitous computing. Distributed systems provide facilities to support client/server, pair-to-pair and grid computing architectures. For the development of CSE tools, distributed systems allow tools to communicate with each other, send and receive custom data, access peripheral servers, and make calls to remote functions and methods.

Distributed systems technology has made significant advances in the last few years, particularly with the introduction of .Net [20], J2EE [65], SOAP [74] and related web services, all of which are explained in the accompanying annotated bibliography. Due to these advances, it is possible to implement very comprehensive collaborative features within CSE tools, with functions more advanced than those typically supported in conventional Groupware toolkits. Additionally, with the advent of the Internet and wireless networking, the physical boundaries of computer networks are diminishing, allowing CSE tools to be supported far further than just the local network.

2.4.6 Software Engineering Metrics and Visualisation

The field of software metrics and visualisation concentrates on the analysis and extraction of useful metrics from software projects. Findings are then presented back to engineers in a way that is useful and minimises information overloading. An example system, JST [60] provides a transparent pipeline which uses a standard language grammar and source files as the input, and provides rich program visualisations and associated software metrics as a result. A class cohesion visualisation from JST is presented in Figure 2.9.

المنارة للاستشارات
www.manaraa.com

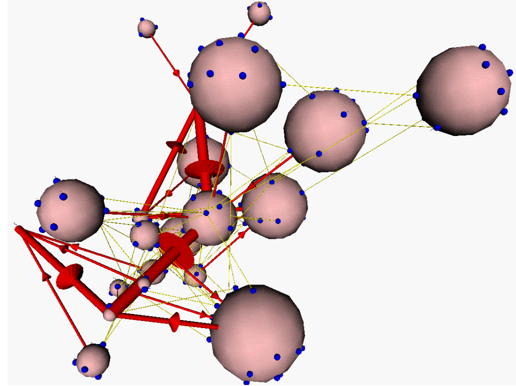


Figure 2.9: A visualisation of class cohesion using the JST pipeline [60].

For CSE research, the field of software metrics and visualisation is immediately interesting: the metrics gathering and reporting are a core function of most CSE tools. Additionally, many CSE tools now employ visualisations of user activity as their main mode of feedback. As SE becomes more collaborative, visualisations may become even more important to the developers of CSE tools; the additional dimension of multiple users and their interactions with artifacts over time provides rich information for the software team.

It should be noted, however, that the most effective modes of visualisation for both standard and collaborative SE tools are yet to be fully investigated by software visualisation and HCI researchers [53]. Alternatives to common interfaces such as explorer panes and tree views are likely to be necessary as richer types of information are displayed.

2.5 Previous Work Towards Collaborative Software Engineering

This section presents an overview of previous work towards CSE. For a more in-depth listing and discussion of the related literature, please refer to the annotated bibliography contained in the accompanying resources disc.

2.5.1 Overview

Research into CSE is progressing rapidly. Driving factors for this current surge of research include the advent of industrial-strength open source IDEs,

المنارة للاستشارات
www.manaraa.com

a solidification of standards for distributed computing, significant advances in processing speeds and memory capacities, and more powerful, interoperable programming languages. Reliable high-speed networking reduces the boundaries between remote developers, programming frameworks such as .Net and J2EE provide efficient access to rich information related to any given software project, and new collaborative features can be incorporated into IDEs through open Application Programmer Interfaces (APIs).

SE encompasses a wide range of tasks ranging from requirements outlining to code debugging, and researchers are now beginning to develop prototype CSE tools for every conceivable task. At the time of writing, a handful of commercial CSE tools also exist for relatively simple SE tasks, and the research world is constantly publishing new and novel architectures, tools and perspectives related to CSE. Researchers have implemented collaboration-based prototype tools using existing SE frameworks, Groupware toolkits, and manually from a blank starting point.

Tools are now available which support real time modelling, design and management of software. Development tools, however, are typically based upon conventional SE tools and technologies. For example, as developers check-in or check-out source code from a central repository, users can be alerted to possible conflicts. Only a few real time editing and diagramming tools exist, where the conventional model of copy, modify and merge is replaced with fully synchronous file sharing with multiple view support.

The remainder of this section highlights each category of tool. At the end of this section, a feature matrix is presented that compares these existing CSE tools with Caise-based tools.

2.5.2 Design Tools

CSE design tools have some or all focus on supporting collaboration during the design of SE artifacts. CSE design tools focus on work-flow, communication and basic source file generation rather than on low-level coding. Tools within this category typically support the design of relatively simple and low-detailed artifacts such as class, sequence and Class-Responsibility-Collaborators (CRC) diagrams.

www.manaraa.com

Other Unified Modelling Language (UML) diagrams such as state transition diagrams and use-case diagrams appear too complex to be supported by CSE design tools at present, although a few limited commercial implementations of such tools have been recently released, such as Poseidon for UML Enterprise Edition [11], presented in Figure 2.10. Poseidon supports shared UML modelling, with locking if required and a conflict detection and resolution facility. Additionally, while not shown in the current screen-shot, Poseidon also supports a shared white-board facility and instant messaging between developers.

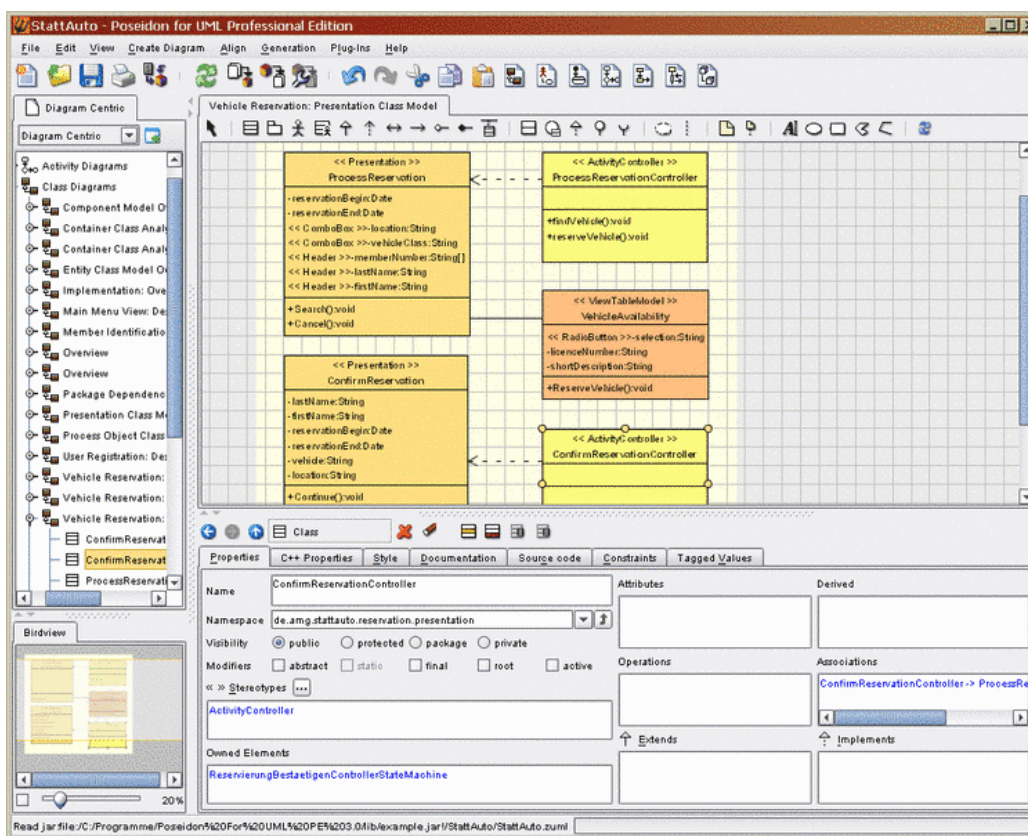


Figure 2.10: The Poseidon collaborative UML tool [11].

For web-based shared UML editing, Rosetta [48] is a well known research prototype. The Rosetta architecture allows editing of HTML-based software design documents from the Internet, with embedded UML diagrams. An

المنارة للاستشارات
www.manaraa.com

editor applet allows collaborative editing of UML diagrams, as presented in Figure 2.11. Rosetta also supports code conformance tests, where source code is compared against its design documentation for possible inconsistencies.

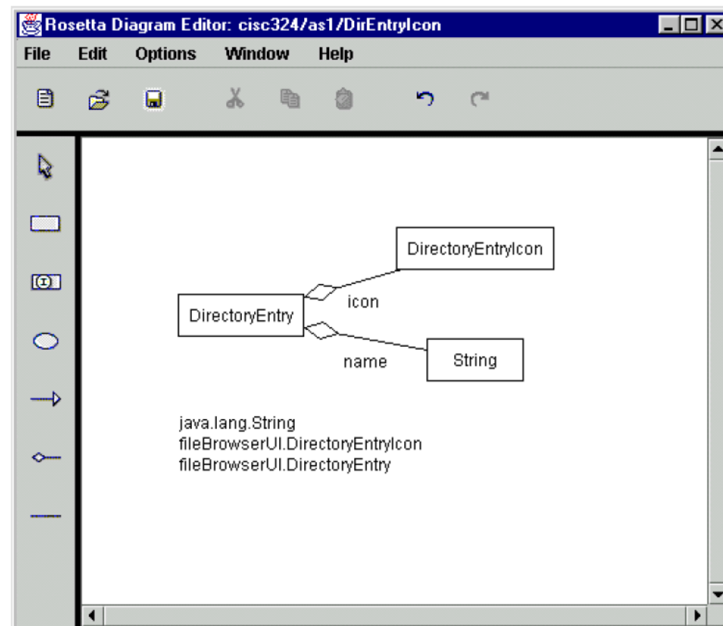


Figure 2.11: The Rosetta web-based collaborative design document tool [48].

2.5.3 Development Tools

Many specific and ambitious prototype tools exist to accommodate a range of development tasks. For distributed eXtreme Programming a new framework called Moomba has been released [92]. The Moomba environment for distributed XP is presented in Figure 2.12. Moomba facilitates the daily activities of XP in a collaborative manner, where user stories and other XP artifacts can be shared and modified by multiple uses. Moomba also supports a fully-featured IDE for shared collaborative editing, which includes syntax highlighting, code completion, build and collaborative debugging support.

Moomba is the successor to Tukan, a CSE tool for SmallTalk editing [100]. The Tukan system is presented in Figure 2.13. Tukan supports editing of source files, but code changes are not propagated to other users; instead

المنارة للاستشارات
www.manaraa.com

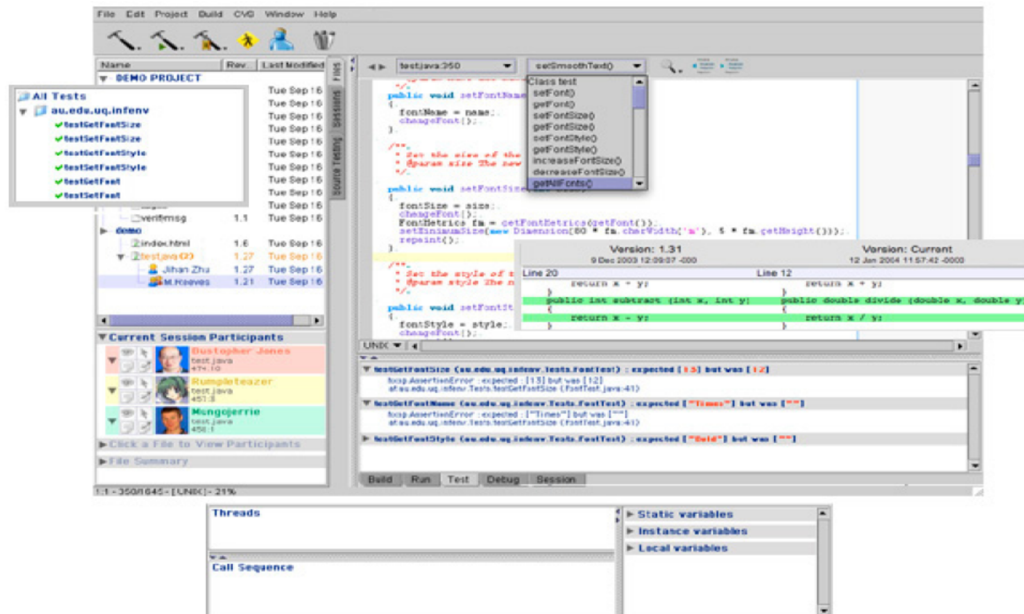


Figure 2.12: The Moomba collaborative XP development environment [92].

Tukan provides real time awareness of other users' presence and their perceived potential to make conflicting changes. In Figure 2.13, Tukan's collaborative code indicators are visible, which convey degree of interest (DOI) information and potential configuration issues between programmers.

In the last year many of the major commercial IDEs have also taken significant steps towards code-level real time collaboration. Of the five Java IDEs that have the largest market shares, two of them now support shared development facilities, and all five environments are promising more to come in the next major releases.

Eclipse [83] is arguably the most popular development environment for Java, and has the support of many of the industry's largest corporations. While Eclipse itself does not support code-level collaboration, a new subproject called the Eclipse Communication Framework [66] aims to allow the Eclipse code repository and project model to be shared and collaboratively edited. The API to perform basic sharing is available now, along with some prototype client applications. Such an application is presented in Figure 2.14, where a shared graph editing tool is hosted within the Eclipse IDE.

www.manaraa.com

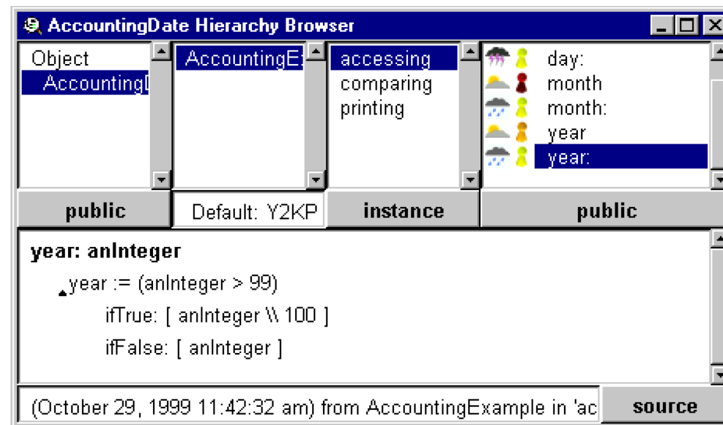


Figure 2.13: The Tukan collaborative code editor [100].

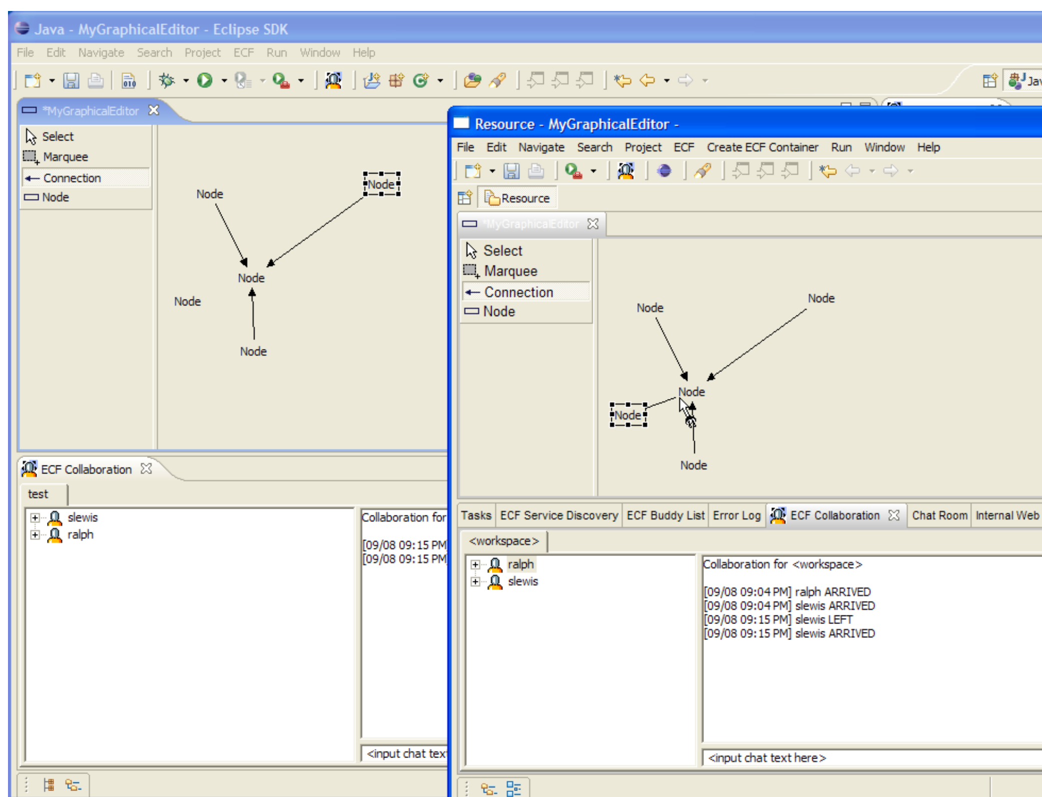


Figure 2.14: A graph editing tool within the Eclipse Communication Framework [66].

المنارة للاستشارات
www.manaraa.com

Borland's JBuilder [12], as presented in Figure 2.15, is another of the main IDEs in the Java development market. It supports real time remote refactoring, distributed views of UML diagrams, and chat channels. At the time of writing, the latest version incorporated a shared pair-programming code editor and collaborative debugging capabilities, although this has been implemented with a rather restricted token-passing floor control policy where only one user can make modifications at a time.

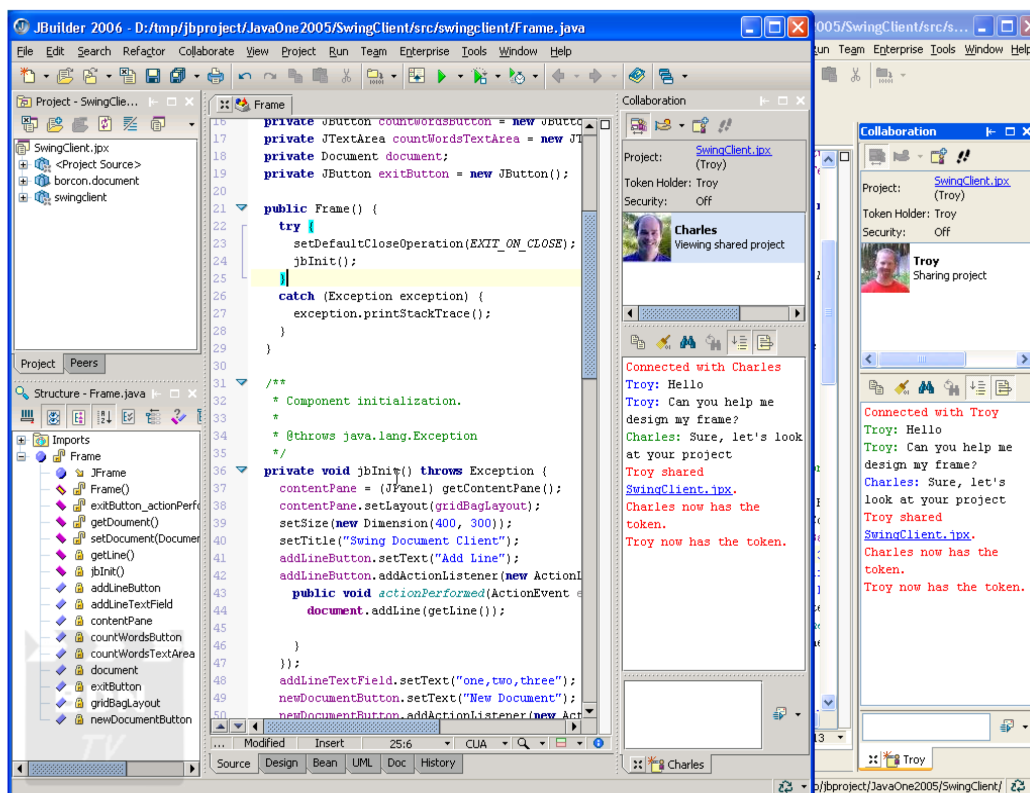


Figure 2.15: Borland's JBuilder IDE with pair-programming capabilities [12].

Similarly, Sun's JSE [115] now supports a collaborative code editor, with more plans on the way for the next release. There are also collaborative plug-ins available for Oracle and Rational's IDEs, bringing them into the market for code-level collaborative development tools.

www.manaraa.com

2.5.4 Inspection Tools

CSE inspection tools typically support one of two functions: allowing users to collaboratively inspect code and designs as a group, or allowing single users to inspect code and designs that have been collaboratively developed. Inspection tools differ from management tools in that their key role is the inspection and investigation of SE artifacts for the benefit of future development and refinement, as opposed to management tools that are more concerned with group coordination, high-level design and artifact control.

An example of a popular inspection tool is Augur [41], as illustrated in Figure 2.16. Augur is a comprehensive tool for inspecting and exploring software development activity. Augur consists of a data gathering architecture based on semantic analysis of source code repositories and a set of visualisation tools. These tools allow developers to monitor their activity and explore the distribution of their combined activities over time and artifacts.

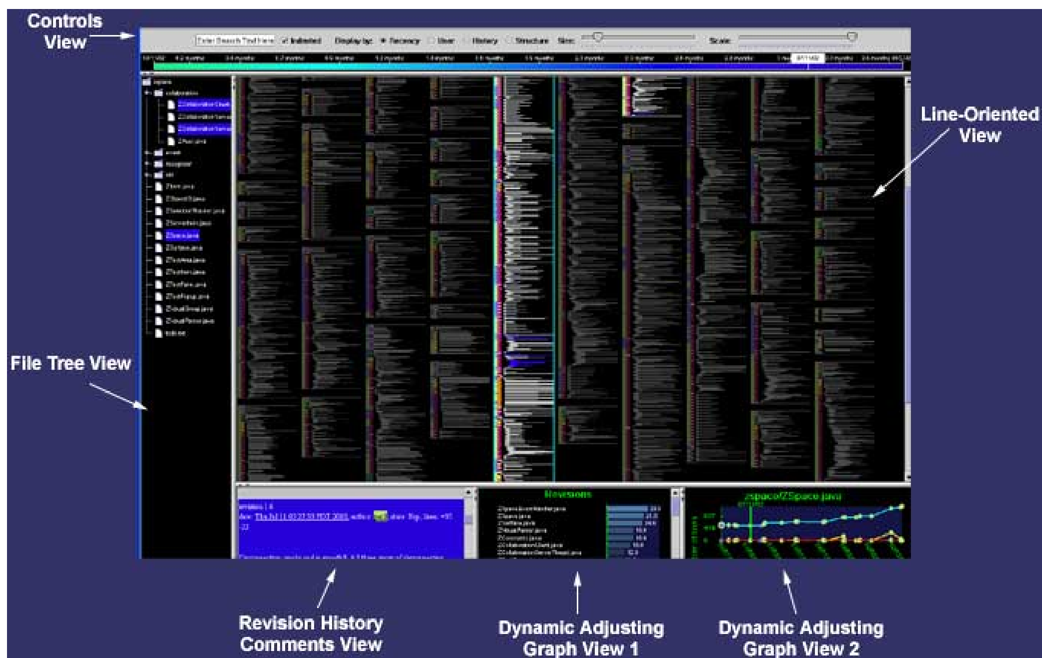


Figure 2.16: The Augur inspection tool [41].

For change impact reporting the Palantír architecture exists [98]. Figure 2.17 presents Palantír's visualisation component, which informs devel-

المنارة للاستشارات
www.manaraa.com

opers of potentially conflicting source file check-outs from code repositories. The goal of Palantír is to raise awareness of currently isolated programmers.

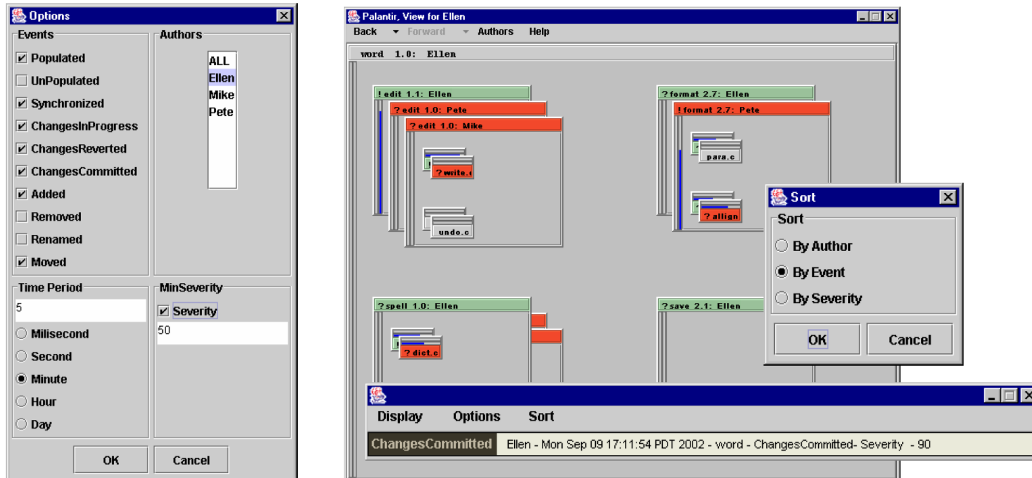


Figure 2.17: The Palantír collaborative visualisation tool [98].

2.5.5 *Comparison to* Caise*-Based Tools*

Before presenting the Caise framework and associated tools, it is worthwhile comparing the features and abilities of existing CSE tools. The CSE tools presented previously in this section are categorised in Table 2.1. The categories used to summarise these tools are explained further in Section 4.2.

With reference to the feature matrix presented in Table 2.1, it is apparent that CSE tools vary in the number of core features supported. This is to be expected, as CSE tools are tailored for specific purposes and do not generally need to support all SE tasks. Being collaborative and written without the use of supporting frameworks, however, guarantees a high implementation cost for these tools, yet they are not generally as powerful as all-purpose single-user SE tools such as IDEs.

In the design of the Caise framework, a key objective is to be able to provide as many core CSE features as possible for application developers to utilise. Instead of writing tools for all purposes of CSE, the research presented in this thesis aims to provide a framework to support rapid CSE tool

المنارة للاستشارات
www.manaraa.com

Tool	Extensible	Synchronous	Multiple Language Support	Multiple Views of Artifacts	Semantic Modelling	User Presence Feedback	Impact Reporting	Supported Phases
Tukan	✗	✓	✗	✗	✓	✓	partial	implementation
Poseiden	✗	✓	✓	✓	✓	✓	✓	design
Moomba	✗	✓	✗	✗	✓	partial	partial	implementation
Palantír	✗	partial	✗	✗	✗	✓	✓	implementation
Rosetta	✗	partial	✗	✗	✗	✗	partial	design
Eclipse	✓	partial	✓	partial	✓	not yet	partial	implementation
Augur	✗	✗	✓	✗	partial	✗	✗	implementation
CAISE-based tools	✓	✓	✓	✓	✓	✓	✓	design & implementation

Table 2.1: Feature matrix of existing CSE tools.

www.manaraa.com

construction. Given CAISE, CSE tools will have the potential to successfully support all of the categories listed in the feature matrix presented above by utilising the framework's services.

The key differences between the CAISE framework and other types of CSE tools are that CAISE-based tools are easily extensible through clearly defined APIs, have access to rich project information via a shared incremental source code semantic analyser, and tools are fully synchronous in which any number and types of tools can collaboratively edit artifacts in real time, even from different artifact views. In Chapter 6, a discussion on how to rapidly implement new CAISE-based CSE tools is presented.

2.6 Collaborative Software Engineering Barriers

Some of the original claims made in CSCW and Groupware literature included assertions that collaborative applications showed great potential to become commonplace. Today, however, it is still a daunting task to implement collaborative tools for SE purposes. Many areas of expertise are required, and industrial-strength tools have many facets that must be suitably supported. This section presents a discussion on the barriers to the support of CSE and associated tools.

2.6.1 Groupware Support

The literature surrounding CSCW originally spoke of solving many problems related to computerised collaborative work [49], which suggests that CSE tools would be trivial to implement once technology had naturally progressed. Unfortunately, twenty years on from those claims, we are still no further to having a 'silver bullet' technology that facilitates CSE or any other complicated domain. The failures of ambitious CSCW projects within other fields are now well documented [50].

Groupware toolkits such as GroupKit [95] and Maui [55] allow the sharing of files, whiteboards and other common forms of electronic media, and good results have been achieved when converting some generic applications to their multi-user equivalents [33]. Extending Groupware to specific CSE applications has been trialled elsewhere with varying degrees of success [100, 22].

المنارة للاستشارات
www.manaraa.com

Problems occur, however, when building industrial-strength CSE tools from Groupware toolkits. Professional tools are not limited to a single task, language or artifact view, and this is orthogonal to the characteristics of Groupware. CSCW technology is based on the support of unstructured and transient documents that have little or no semantic relationship to other artifacts. SE involves highly structured, evolving documents that have vast inter-dependencies and long lifetimes, but Groupware technologies have no understanding of the complex semantics or syntax of such artifacts, and the relationships between artifacts and users.

An attempt could be made to extend a single-user IDE collaboratively through the use of a CSCW toolkit, allowing it to support distributed collaborative development of code or UML diagrams. A significant difficulty, however, is that conventional SE tools are designed for single-developer use, and appending collaborative features to single-user tools does not necessarily scale or provide the level of improvement envisaged.

2.6.2 Building Industrial-Strength Tools

While the proposal of tools to support CSE often draws an enthusiastic response from practitioners, the design and implementation of industrial-strength tools is a challenging task; very few research prototypes have evolved into features within professional tools. Even once such tools have been developed, there is no guarantee that they will gain widespread adoption.

Implementing collaborative features for industrial-strength tools is a very challenging problem. To date, SE tools typically work with the lowest common denominator of SE artifacts: source code. By employing source files as the finest-grained type of shared information, and by supporting information sharing only through code repository systems, it is difficult to extend IDEs to support real time within-files collaboration, to provide support for multiple views of software, and to provide collaborative access to the underlying semantic model of software for tool extensibility purposes.

Given that CSCW technology does not scale to meet the needs of CSE, and IDEs do not provide enough fine-grained information to support the development of highly-synchronous new types of collaborative tools, often

www.manaraa.com

the only means of producing new collaborative tool sets is by completely redesigning tools upon a foundation of CSE technology.

Summary

In this thesis, an approach to supporting CSE in a practical and extensible way is to be defined. This is an important contribution; current tools do not support key aspects of SE such as communication and collaboration as a core part of the software development lifecycle. Consequently, problems such as merge and transactional conflicts are often hidden for long periods of time using conventional SE approaches and tools, which is usually against the ideals of the developer and the employed SE methodology.

A solution where computer-mediated CSE is supported may allow developers to be aware of the actions and intentions of others in real time, avoiding coding errors and potentially speeding up the software development process considerably. A means to supporting CSE has been challenging to derive because conventional SE tools are inherently single user, and they can not easily be augmented by Groupware to solve all CSE problems. Purpose-built CSE tools can provide specific facilities to support SE more collaboratively, but a more general-purpose solution may be required for real-world software development teams.

In Chapter 3, patterns of collaboration evident within SE are identified. By exposing the recurring modes of work between developers within collaborative software projects, CSE tools can be designed according to the observed key requirements of collaborating developers.

المنارة للاستشارات
www.manaraa.com

Chapter III

Patterns of Collaboration

Patterns of collaboration have been identified by several research groups as a useful means to describe trends of interaction and cooperation. This chapter has a specific focus on patterns to support CSE, starting with an overview in Section 3.1. General patterns of interaction are introduced in Section 3.2. In Section 3.3, a discussion on the types of collaboration within SE is presented. This chapter concludes with the presentation of candidate patterns for CSE in Section 3.4.

3.1 Motivation

To support genuinely useful CSE tools, identification of the demands that programmers are likely to place on such tools is essential. To facilitate tool development, it is equally important to identify the core functions that researchers will require when constructing new types of tools. Research behind the Caise collaborative framework was guided by the identification of the patterns presented in this chapter. The research goal is to support developers working together in ways described by these important patterns.

Patterns are commonly used to document recurring situations and solutions. Alexander's concepts from the architectural domain [3, 2] have been remarkably successful, and design patterns for SE [43] have become an industry standard form of documentation for the construction of software systems. Patterns vary in degree of detail; some describe simple concepts such as elegant mechanisms for iterating over a list, others describe entire organisation structures such as *Conway's Law* [63].

The success of patterns within software design has led to the construction of pattern languages for other fields. Martin and Sommerville [68] have iden-

المنارة للاستشارات
www.manaraa.com

tified a number of patterns which reflect the ways in which groups of people interact to perform tasks. Patterns have also been identified for the organisational management of software development by Harrison and Copelien [63], and patterns for Groupware have been described by Schümmer [101].

In this chapter, the concept of patterns of CSE is introduced. These patterns are described for motivational purposes, providing examples of common scenarios within CSE for software engineer researchers. The patterns presented in this chapter are not claimed to be complete. The purpose of providing these example patterns is to illustrate basic situations that CSE tools should support; an inability to support these various modes of work suggests that further work is needed in CSE tool design.

3.1.1 The Patterns Language

Design patterns are typically described through a *patterns language*. A pattern language attempts to abstractly define recurring trends of software design. While no one patterns language has gained universal acceptance, most describe the following list of properties:

Name The common name given for the pattern

Context The general situation in which the pattern can be applied

Problem The problem that the pattern addresses

Forces The factors that govern the use of the pattern for the given context. Forces include ease of pattern application, scalability of design, and robustness

Symptoms Known undesirable characteristics of software that indicate the given pattern might provide suitable relief

Solution A description of how to apply the given pattern. This will typically include coding examples, UML diagrams, and a discussion of the intricacies of applying the solution to specific contexts

المنارة للاستشارات
www.manaraa.com

Rationale A justification of why the pattern is desirable, and how the solution provides a better final design than other approaches

Examples A walk-through of software as it is refactored to accommodate the given pattern. Examples normally discuss the problem, context and forces as well as the solution

Danger Spots The common pitfalls when using this pattern, and recommended work-arounds

Known Uses Identification of existing applications of the given pattern in software systems or processes

Related Patterns A listing of collaborating and associated patterns, as well as patterns that may provide an alternative solution

Known as the *rule of three*, the publishing of a pattern by its designer is generally discouraged within the patterns community. Rather, three independent examples of the pattern being used within typical SE scenarios should be identified and documented by a third party. This rule is aimed at preventing the proliferation of weak patterns.

3.1.2 An Example Pattern

To serve as an overview of this chapter, the *Mode of Development* pattern is presented. This is a candidate pattern identified within the field of CSE. As part of the work towards CSE, Mode of Development is discussed in detail in Section 3.4.2.

The Mode of Development pattern describes the predominant ways that programmers interact with each other when working collaboratively on a shared set of artifacts. As explained in Section 3.4.2, there are many modes of development, including one identified as *Action/Reaction*. The Action/Reaction mode of development encapsulates the recurring behaviour of the following common situation: one programmer makes a modification to the code base, a second programmer is alerted to a possible conflict, and then both programmers group to resolve the conflict.

المنارة للاستشارات
www.manaraa.com

Figure 3.1 presents the Action/Reaction pattern as it eventuates. In this example, CAISE-based CSE tools are used to illustrate the pattern within the context of fine-grained interaction. To set the scene, user Alice is working with a class diagramming tool, and user Bob is working with a text editor. This example continues from the scenario presented in Section 1.1. Both tools are running in real time collaborative mode, operating on a shared code base.

In Figure 3.1(a), Alice decides to rename the method `saveXML()` in class `Persistence` to `saveAsXML()` through the class diagrammer. Alice is not aware, however, that a new call to `Persistence.saveXML()` has recently been made by user Bob in the file `GUI.java`, and that renaming the method will break the code.

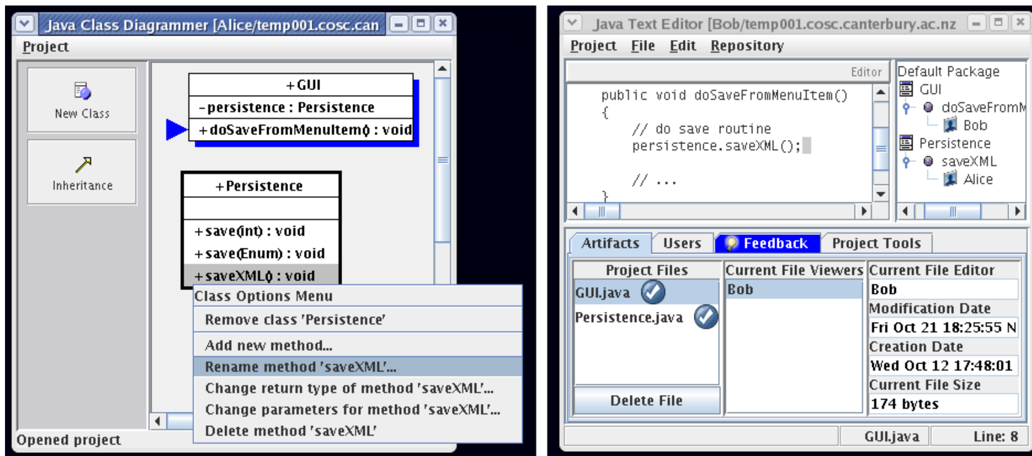
In Figure 3.1(b), Bob is notified through an awareness mechanism that the project has recently moved into an inconsistent state. In the lower half of the text editor presented in the right hand side of Figure 3.1(b), the Artifacts Pane highlights the file that currently contains a semantic error. By inspecting the Feedback Pane at this point, Bob will be informed that the method he is currently editing makes a call to `Persistence.saveXML()`, which is now unresolved. At this point, Alice will also be made aware of the same problem through feedback mechanisms.

In Figure 3.1(c), the problem is resolved. By using the feedback information presented to both users, or perhaps by simply talking with each other, both Bob and Alice can decide that either the method renaming operation needs to be reversed, or refactoring of all calls to the changed `Persistence.saveAsXML()` method is required. In this example, Bob simply updates the method call from within the file `GUI.java`, and the project again reaches a buildable state. This concludes the Action/Reaction cycle of events.

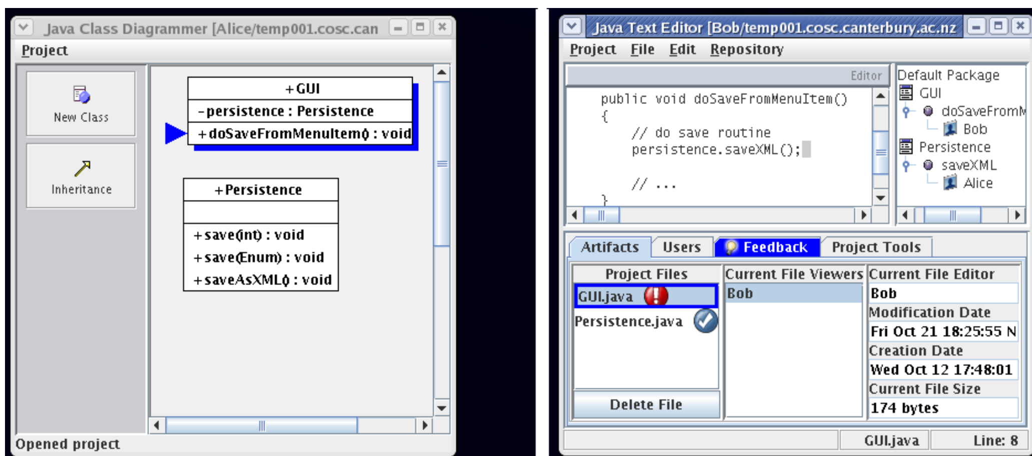
Given adequate and suitable tool support for the Action/Reaction pattern, both users will be alerted to the problem at hand and are also given the opportunity to immediately consult with each other and correct the problem.

To describe the Action/Reaction pattern in terms of a patterns language, some example excerpts are presented. The *context* for this pattern is any situation where overlapping modifications can be made by any number of users

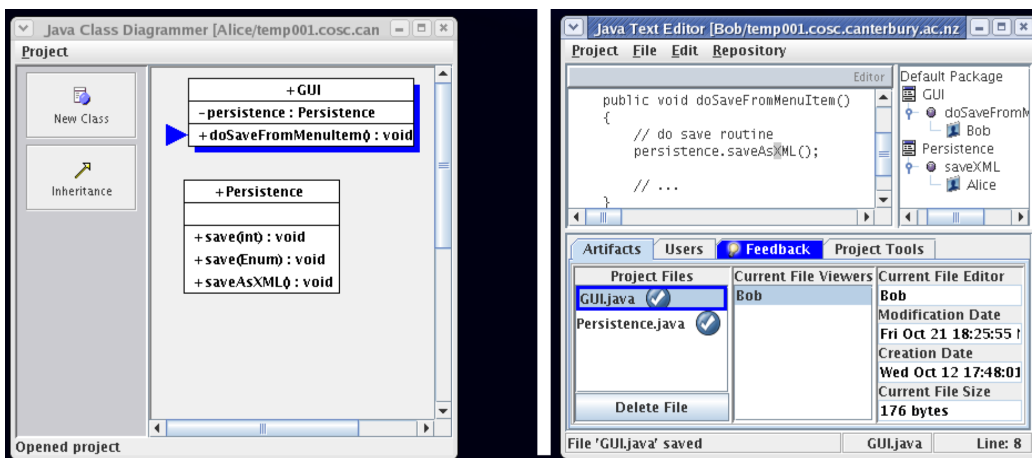
المنارة للاستشارات
www.manaraa.com



(a) The action: Renaming a method.



(b) Notification: The project has become unstable.



(c) The reaction: Refactoring all relevant method calls.

Figure 3.1: An example of the Action/Reaction candidate pattern of CSE.

المنارة للاستشارات
www.manaraa.com

in either synchronous or asynchronous settings. The *problem* can be defined as a lack of awareness that allows conflicting changes to be made without detection by any party. *Competing forces* include the degree of isolation each programmer desires, the feedback mechanisms available, and the ability to recognise conflicting changes. A potential *danger spot* is transient changes that can safely be ignored, although the identification of genuinely transient changes is most likely impossible to automate. *Related patterns* include Observer/Observable from Gamma, Helm, Johnson, and Vlissides [44], Private Work from James O. Coplien and Neil Harrison [63] and Public Artifact from Martin and Sommerville [68].

Without strong tool support for the Action/Reaction pattern, Alice and Bob could continue to cause conflicts. If both users are not working closely together, they are likely to only detect the inconsistent state of the program after synchronising their source files. In this case, both users will probably attempt to fix the problem by modifying their individual changes—which will again break the main project build once both sets of source files have been committed back into the code repository.

It should be noted that even with conventional tools, the Action/Reaction pattern will still apply. In such cases, however, the time-scale between each phase of the pattern will be longer. This is because the notification of conflicting actions is usually actuated by periodic build reports from the central code repository. The resolution of the conflict may also be confounded by other modifications made prior to conflict identification.

3.2 Patterns of Interaction

Previous work towards the identification of interaction patterns exists. The main perspectives that are closely related to CSE, as outlined in the previous section, include Groupware Patterns [101], Patterns of Cooperative Interaction [68], and Organisational Patterns for Agile Software Development [63]. These patterns to describe interaction between participants within collaborative settings are well aligned with CSE processes.

Example patterns from Sommerville and Martin's Cooperative Interaction collection include *Artifact as an Audit Trail* and *Collaboration in Small*

www.manaraa.com

Groups. The Artifact as an Audit Trial pattern discusses how various types of artifacts are used within collaborative systems as a means for providing revision histories. This includes a discussion on why keeping the history of artifacts is useful in terms of providing audit trials, which is of direct relevance to many types of SE artifacts.

Many other patterns from this collection can also be used to describe activities and aspects of CSE. This collection has recently been expanded to describe XP as a collaborative process [67], using a subset of patterns with specialised examples and descriptions. Since XP can be viewed as a subset of generic CSE in many ways, patterns of cooperative interaction for XP are very closely related to CSE.

Copelien and Harrison's Organisational patterns [63] for agile software development serve to document and describe the recurring themes within a modern software development team. These patterns, based from extensive observations and community input, include *Private Work*, *Incremental Integration*, and *Developing in Pairs*. Many other patterns in this large collection are also highly related to the collaborative construction of software, which is not surprising considering the large amount of group work within SE teams today.

As an example pattern, Developing in Pairs discusses the inevitable blindness of working alone, and the psychology behind teamwork for problem solving. This pattern also comments on the high likelihood of producing more as two developers together than the sum of two people working alone. For CSE researchers, this pattern identifies important associated patterns, and presents some examples of tasks well-suited to development in pairs.

Schümmer's Groupware patterns, while not specifically related to SE, are closely related to CSE processes and the support of CSE tools. These patterns, which are a work in progress, outline the recurring themes within systems that utilise computer mediated interaction. Example patterns include *Mode of Collaboration*, *Private Workspace*, *Shared Workspace*, *Collaborative Virtual Environments*, *Floor Control* and *User Awareness*. This collection of patterns list and describe the considerations for any CSCW-based tool developer, with aspects ranging from multi-user widgets to social protocols of interaction.

المنارة للاستشارات
www.manaraa.com

Patterns for collaboration within SE and related fields are becoming increasingly investigated and documented. What appears to be missing, however, is a discussion of patterns of collaboration for real time, fine-grained CSE. It is likely that researchers have not addressed a CSE-specific family of patterns before because the technology and tools to support CSE are not yet readily available. Given tools to support fine-grained real time CSE, new research opportunities exist to explore CSE patterns as they emerge.

3.3 Collaboration within Software Engineering

Before discussing patterns related to CSE, the functions and roles common to SE are investigated. This section presents recurring SE topics such as the modes of collaboration evident within SE today, and the types of feedback necessary to support well-informed development of software. The discussion presented in this section provides a listing of key observations within SE practice, independent of specific methodologies that might be followed.

3.3.1 Modes of Collaboration

The quadrants in Table 3.1 represent the four main modes of computer-mediated collaboration as practiced today. Conventional, code repository based SE can be mapped to either quadrant in the asynchronous column, depending on whether development is entirely in-house, or distributed via a networked repository such as SourceForge. The pair-programming aspect of XP can be mapped to the same-time/same-place quadrant.

	Synchronous	Asynchronous
Co-located	Face-to-face meetings	Office document editing
Distributed	Text chat	Email

Table 3.1: Synchronous versus asynchronous development: typical tasks within each quadrant.

For developers of CSE tools, support for collaborative development can potentially be mapped to all quadrants within Table 3.1. For example, a tool that supports the real time editing of source files can be used in co-located

المنارة للاستشارات
www.manaraa.com

or distributed settings if it provides adequate awareness support about the actions and intentions of other collaborating users. Similarly, if users are located in different time zones, it is possible that usage might be asynchronous rather than in real time. Other CSE tools might be designed only for use asynchronously and/or in co-located settings.

3.3.2 Current Facilities for Collaboration

Within nearly all fields of work, people find it necessary to collaborate with each other. Typical means of collaboration include face-to-face meetings, tele-conferencing, email, text and audio chat, telephone correspondence, memos and written letters.

Within SE, other facilities for collaboration are available and used regularly. Using code repositories, developers are notified when their commits back into the repository fail due to merge conflicts. Additionally, automated build results will notify software engineers of compilation problems. If automated test facilities exist, engineers will also be notified when unit tests fail due to recent changes within the code base.

In terms of face-to-face meetings, the XP process formally prescribes daily meetings to discuss the current state of the project. Regular meetings to plan and discuss current development activities for most other SE processes are implicit.

At an hour by hour granularity, however, field studies suggest that engineers spend from quarter [120] to half [15, 86] their time communicating with others even when supposedly developing code individually. This type of impromptu interaction occurs whenever problems are detected during development and testing. Most problems are detected from repository notifications and build reports. These events prompt developers to discuss their recent code changes and proposed resolutions in more detail than the daily recap meeting, and only with the subset of developers who are most closely tied to the changes.

المنارة للاستشارات
www.manaraa.com

3.3.3 Examples of Existing Collaboration Support

Collaboration within SE extends to processes, tools and artifacts. Existing support for collaboration within SE is limited, which serves as motivation for the work presented in this thesis.

As described in the previous section, software engineers use meetings, email and impromptu face-to-face discussions to communicate, coordinate and resolve issues during the development process. As an aide to discovering potential issues and conflicts, feedback from code repository systems, daily builds and unit test facilities are common sources of activity information.

In terms of tool support for collaboration during conventional SE, examples have been introduced in Section 2.4.1. Prototype components and frameworks to assist collaboration beyond the capabilities of conventional tools have been presented in Section 2.5. In this section, a discussion is presented of how developers typically use conventional SE tools during group development.

To support pair-programming, a single instance of an editor is typically used. Two programmers have alternating ownership over the input devices, where one user will make changes to artifacts based on agreement with the observer. Code reviews are also carried out in a similar manner to this. For editing of source code within a conventional team environment, each developer will use a text editor of his or her choice. Files are typically shared asynchronously via a code repository, with regular integration of modified files.

When using more complex tools such as IDEs, collaboration exists even within single-developer projects. IDEs typically support multiple views of artifacts, which means that all components within the IDE must collaborate with each other to keep their views consistent—otherwise known as *round-trip engineering*. When multiple developers within a team use IDEs as their code editor, collaboration is again facilitated through a code repository.

Regardless of the types of tools used, developers typically also use email and mailing lists to help them coordinate and communicate at a higher level than what their tools and code repository permit [51].

المنارة للاستشارات
www.manaraa.com

3.3.4 Types of Awareness

At present, single-user SE tools do not have the ability to report the actions of others or the global impact of changes made as they happen. At best, conventional tools can analyse changes to the current code base only when code is integrated with the central repository, or when the tool updates its version of files from the repository. Most forms of static code analysis take no notice of which user made modifications, and can only be performed once the changes have been made.

An interesting challenge to researchers and designers of CSE tools is defining the types of feedback that should be presented to users, given the ability to fully analyse a software project as it evolves in real time. Given tools that can integrate the efforts of any number of programmers in real time and from any type of tool, it is difficult but possible to generate rich information related to the impact of modifications, and the identification of relationships between distinct units of code being developed concurrently.

A listing of typical types of awareness information for collaborative software projects is to be presented in Section 4.4. Given these types of feedback within SE tools, the level of awareness afforded to individual developers may be greatly improved. Awareness extends not only to changes made by a single user within his or her private workspace, but the actions, physical locations and relationships to others within the entire software project.

3.3.5 Atomic Elements of Collaboration

Ultimately, supporting collaboration can be reduced to the identification and control of *atomic elements* of SE tasks. Atomic elements are the smallest useful units of activity that a developer can produce and a tool can recognise. Using the Action/Reaction example presented in Section 3.1.2, tools need to identify what is being done in terms of changed program syntax and semantics, synchronise all interleaved actions between developers, identify any potential conflicts between changes, notify others of what is happening, and facilitate coordinated discussions on how conflicts and design decisions should be resolved.

The level of granularity in identifying actions is dictated by the degree of

المنارة للاستشارات
www.manaraa.com

synchronisation supported; a shared code editor is likely to expose changes on a per-character basis, whereas UML diagrammers may only recognise changes once they have become syntactically-complete.

The floor control policy will dictate the order that atomic operations are propagated to users within the project. In a token-passing scheme, only one person might be allowed to make a change at a time. Within a fully-synchronous application, operations might be propagated purely in chronological order, with the possibility of blocked operations if a conflict is detected.

3.4 Candidate Patterns of Collaborative Software Engineering

In this section, some patterns evident within the process of CSE and development are identified. Currently, these patterns are not well supported by SE tools. For example, following the leader as he or she demonstrates a new coding idiom is a commonplace activity within development, but typically the only way to view such a demonstration is by all developers gathering around one workstation.

It is unlikely that an expansive new family of patterns exclusive to CSE exists. It is observed, however, that real time support for CSE is limited, and with adequate tool support these patterns would be more readily recognised.

Some patterns of collaboration for SE are difficult to support with conventional tools, which means that they can only be accommodated with a very coarse granularity. For example, seemingly independent changes to source files often involve unforeseen side affects such as broken code dependencies. Currently, the only means to detect subtle coding conflicts between collaborating programmers is to integrate all checked-out source code and investigate the errors from the resultant project build. This lengthens the development time between releases of stable versions and can also confound any other concurrent source code modifications.

It can be envisaged that with real time tool support for patterns such as independent code modification, software development might become considerably easier.

المنارة للاستشارات
www.manaraa.com

3.4.1 Formal Identification of Patterns

The research in this thesis does not specifically involve the formal identification of CSE patterns. In order to design successful CSE tools, there is a natural interest in observing how software engineers coordinate their tasks and collaborate during development. It is hoped that these findings are useful to CSE researchers and the general patterns community alike. At this stage, however, it is not a key research objective to exhaustively critique and formally publish the patterns presented here.

As discussed in Section 3.1.1, the *rule of three* states that the designers of patterns should not be the same party that publishes the pattern. For patterns enthusiasts, there are many recurring examples of CSE patterns within existing software development practices and tools, giving researchers the opportunity to document and publish such patterns formally if desired.

3.4.2 A Patterns Map for Collaborative Software Engineering

Figure 3.2 presents a CSE-focused patterns map. In this map, related patterns families described in Section 3.2 are grouped, such as organisational patterns, Groupware patterns and patterns of cooperative interaction. This map represents the recurring trends of interaction between collaborating software engineers that I have identified; it is provided as a means of understanding the implications, competing forces, and different contexts related to CSE. As indicated by the key in Figure 3.2, several of the patterns within this map are my own contribution.

CSE is essentially a union of these related families of patterns, with some additional specific characteristics. In a manner analogous to the ten patterns of cooperative interaction, work is being carried out to identify these special characteristics. From preliminary observations based on prototype trials of tools, I introduce two new candidate patterns for CSE in this section: *Atomic Integration* and *Modes of Development*.

Atomic Integration

James O. Coplien and Neil Harrison discuss the pattern of Incremental Integration [63], where modified units of source code are regularly and frequently

المنارة للاستشارات
www.manaraa.com

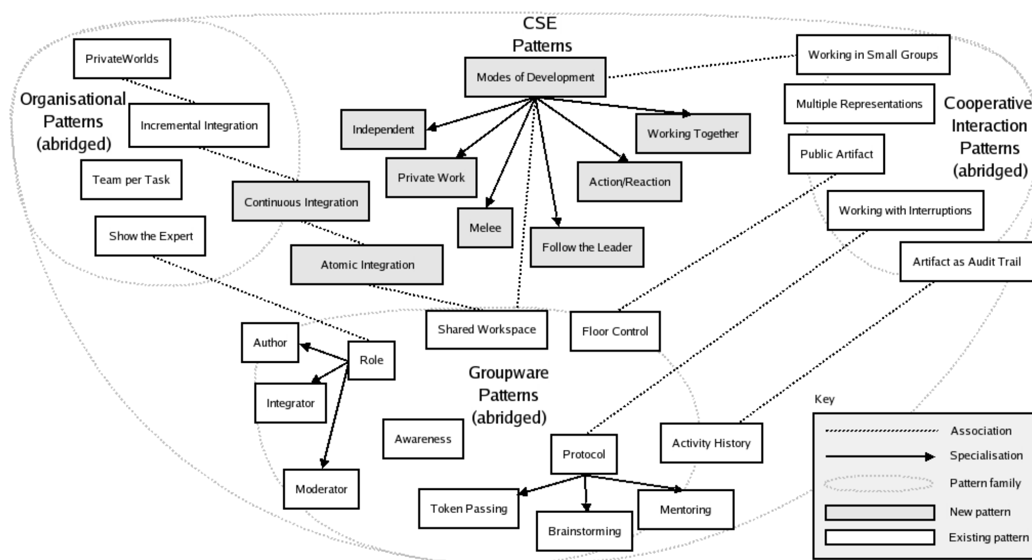


Figure 3.2: The CSE patterns map.

checked back into the main code repository to prevent any major development skew. Most proponents of incremental integration argue that a daily code integration process is sufficient [69].

Going one step further than this, the concept of *Continuous Integration* has been proposed [40]. Continuous integration encourages developers to integrate their modified source files immediately after any modification, as well as re-synchronising their own cache of unchanged files with the latest version from the code repository. Some tool support for continuous integration also exists [39].

Fully-synchronous development tools such as Poseidon [11] and Moomba [92] allow the concurrent modification of shared SE artifacts. As the artifacts are shared in real time, no integration effort is required. I define this mode of configuration management as *Atomic Integration*, where integration is constant. Each modification is instantly incorporated into the global project state, with all observing views updated accordingly. Atomic integration is a key characteristic of the Caise framework, as presented in Chapter 5.

المنارة للاستشارات
www.manaraa.com

Modes of Development

The *Protocol* pattern for CSCW has been provisionally identified within the Groupware patterns collection, but no documentation has been published as of yet. The modes of work defined by the protocol pattern, such as token passing and brain storming, are well suited to generic CSCW. For CSE, however, more specific modes of work are encountered. Therefore, I identify *Modes of Development* as a new pattern specifically for computer-mediated CSE.

Several interaction modes [29] have been identified, each characterised by the degree of coordination required and the nature of the activity:

Private: A user effectively withdraws from the group temporarily, typically to convince his or herself of the viability of a change before revealing it to the others. Such a user may require the rest of the project to appear frozen in time. Ideally, it should be possible to integrate the change rather than having to repeat it publicly.

The atomic unit of interaction for private work is a set of source files; once the private work has been performed, the modifications are merged into the main project version. Notifications of modifications in the case of private work are likely to be delayed until the code integration period. It is still possible, however, to alert all parties to conflicting changes even when a copy of the project's source files is being developed in private. This is how the Tukan CSE tool operates [100].

An example of private work is when a user develops a complicated algorithm that has low coupling to the rest of the project. In this case, the developer might prefer to work in complete isolation, knowing that feedback events from other related users are likely to be of low importance. Upon implementation of the algorithm and integration with the main project, the developer is likely to revert back to a more collaborative mode of development. Private work is the key pattern used within conventional SE artifact integration, by way of the copy/modify/merge idiom.

المنارة للاستشارات
www.manaraa.com

Independent: Users are located in regions of code whose semantic relationships are sufficiently weak that they can safely assume independence. Frequent communication is unnecessary and project integrity is not threatened by independent updates.

The atomic unit of interaction for independent work might be most useful at the level of *semantic change*. In this case, when an area of code has been meaningfully changed, such as a method being renamed or an additional class being declared, other related users are notified. It is unlikely in the case of independent development to have overlapping modifications that cause conflicts.

An example of independent work could be user A editing a GUI (view) class to alter a menu, user B editing a customer record (model) class and user C adding a new package which does not yet interact with other classes. In this case, only marginal feedback between users is likely, and communication is expected to be at a low level. The independent mode of work candidate pattern is commonly exhibited in large, well coordinated software projects; a field study of the NetBSD project shows situations where developers follow this pattern [51].

Follow the Leader: One user takes others on a guided tour, possibly making coding modifications along the way. Strict What You See Is What I See (WYSIWIS) might be used to coordinate views, particularly if all users are using the same tool. However, in a more relaxed scenario, users would navigate individually, guided by audio commentary and gestures.

The atomic unit of interaction for a follow the leader situation needs to be fine-grained. Ideally, a modification of any kind needs to be immediately propagated from the leader to all followers. Fortunately, as the leader is the only person likely to be making changes, control of event ordering is trivial. In strict-WYSIWIS environments, CSE tools will propagate changes in the leader's view to all other tools.

A follow the leader scenario might consist of one key developer showing the details of a recently completed change. Another example would be

المنارة للاستشارات
www.manaraa.com

showing a group of developers how to modify one of several classes that all require the same type of refactoring. The follow the leader candidate pattern is exhibited in the JBuilder 2006 IDE [12], by way of a token-based file sharing mechanism.

Working Together Workers examine and edit the project as a pack. In some situations this might mean a group of two or three developers working on the same physical region of code. In other situations this might mean a close group of developers making careful and informed modifications to areas of highly related regions of code. This mode of development is very similar to *Follow the Leader*, except that all members of the group are likely to be involved in the modification of the project rather than just the leader.

As developers in this mode work very closely together, a fine unit of interaction is required, allowing all changes to be propagated immediately. In the case of source code editors, this might mean the propagation of changes on a per-character basis. Social protocols are likely to dictate the order of events when working together; for example, it is unlikely that one user will select and delete an entire method if he or she is aware that another user is currently modifying it.

An example of working together could be three users attempting to split a large class into two smaller ones. One user can define the second class, placing it in the appropriate package. The second user can start moving the relevant methods from the first class to the second class. The third user can start searching for newly broken references and begin correcting them. Until the completion of this task, it is very likely that the users will be in frequent communication, coordinating their efforts and discussing design implications. The working together candidate pattern of CSE can be observed in a field study of the SubVersion project [51].

Action/Reaction: Stronger constraints exist as users become closer in physical, logical or semantic terms. Changes made by a user (the actions) to aspects such as the number and type of class properties, the parameters

المنارة للاستشارات
www.manaraa.com

and return types of methods or the inheritance and interface structure will require responses (reactions) from other users whose work is potentially affected. Awareness mechanisms can alert users to possible threats (e.g. another user is editing a superclass). Collaboration support mechanisms, such as text or audio channels and gestures, can then be employed to discuss and resolve the issues.

The atomic unit of interaction for Action/Reaction modes of work ideally should relatively small. At the most coarse level, every event that updates the project's semantics should be propagated to the view of all other participating users. Even though most propagated changes are likely to go unnoticed as they do not affect the work of other users directly, the instant a modification does cause a conflict for another user, notification and discussion should take place.

A detailed example of the Action/Reaction mode of work was presented in Section 3.1.2. A similar example could be one user changing the type of a parameter in a method definition in class C1. Another user editing class C2 may need to update newly created calls to that method. The sooner the action is exposed to all related users the better in terms of avoiding confusion and development delays.

Mêlée: Several users are making (potentially-) conflicting changes to a set of artifacts and these will be in a state of flux for a period. Such changes would typically be negotiated in advance, and mediated throughout, by infrastructure features such as an audio channel.

There is no obvious atomic unit of interaction for the mêlée mode of development. For groups where communication is restricted, such as in distributed development, fine-grained changes such as per-character modifications are likely to require detection and propagation to all collaborating users. For other situations, the atomic unit of interaction might only need to be relatively coarse, in order to reduce continual interruptions. The level of granularity of change ultimately depends on the existing social protocols for mêlée-based modes of development.

An example of the mêlée mode of development could be a large refac-

المنارة للاستشارات
www.manaraa.com

toring effort where all developers are aware that refactoring is being carried out. In this case, specific refactoring duties are likely to have been allocated in advance. During this period of development, feedback about relationships between participating users and currently broken code dependencies might be replaced by a richer mechanism such as audio-conferencing. The mêlée mode of development can be observed in most conventional software development projects, whenever a period of concerted refactoring takes place using conventional source code control tools.

At a basic level, these modes of development are evident within SE practices today, even without the support of CSE tools. Given a progression towards more synchronous tool support for generally collaborative tasks, it is also possible that these modes of development provide a suitable summary of the main interaction patterns for all fields of computer mediated interaction and CSCW.

3.4.3 Applying Patterns of Collaborative Software Engineering

To assist researchers in the design of new CSE tools, they should be aware of the CSE-related patterns. These patterns represent the recurring themes of software development that have intricate design and implementation considerations. The CSE patterns map presented in Figure 3.2 is a suitable starting point when considering the design of any CSE tool.

With reference to this CSE patterns map, developers of CSE tools need to take into account the modes of development that the tools will support. While a powerful CSE tool might be able to accommodate all modes of development, other tools might specifically support only one mode. In this case, the fundamental design of the CSE tools is likely to differ. For example, if *Follow the Leader* is the only supported mode, then strict WYSIWIS may be the only view that requires implementation.

The type of tool being developed also brings in special considerations. For a collaborative class diagramming tool, perhaps the *Independent* mode of development is assumed. In this case, locking of each currently modified section of the project's semantic model could be implemented to control

المنارة للاستشارات
www.manaraa.com

concurrency; this is Poseidon's [11] primary mode of development. In this case, the *Floor Control* policy can employ relatively simple *Token Passing* to control concurrency issues.

For a CSE tool that supports both source code and diagrammatic views, it will be worthwhile investigating the *Multiple Representations* pattern. For the mode of integration, a decision needs to be made as to the supported levels of collaboration granularity. The tool designer must determine whether modifications are to be committed and integrated incrementally, continuously or atomically. In some tools, multiple levels of collaboration granularity may be possible to support.

Another important consideration is that of *Private Worlds*. If private worlds are to be supported, allowing developers to work in isolation, there are a key number of aspects to consider. How long is a developer allowed to work in a private workspace for? Is the limit based on time, or perceived integration effort? How will integration back into the main collaborative project be supported? Will awareness mechanisms still be afforded to the user when he or she is working on a separate code base?

In terms of developer *Roles*, will there be certain roles for different users, or is this handled at a higher level? Most CSE tools in existence today are built around specific SE processes. Poseidon [11], for example, bases its support primarily around software design. Moomba [92] supports pair-programming based development. Single-user tools also often support specific roles. Therefore, another consideration is which roles will be supported by the CSE tool, or will social protocols alone provide adequate governance for user interactions?

User presence is another very important aspect for any CSE tool. The Groupware patterns introduced by Schümmer provide an excellent starting point for ensuring an adequate system design in terms of user awareness and feedback [101]. Considerations include support for telecursors, multi-user scrollbars, relaxed WYSIWIS, metaphors for user proximities, and audio gestures as described elsewhere [55, 95].

The cooperative interaction pattern *Artifact as an Audit Trail* has interesting implications for CSE. Most CSCW-based collaborative editing systems work on transient artifacts such as shared whiteboards, and a revision

المنارة للاستشارات
www.manaraa.com

history is not required. For CSE, however, in most cases a history of artifact modifications is a useful tool function. If a history is recorded, is it limited to artifact modifications? Or should further information be logged such as user interaction, attempted project builds, and semantic events such as a new class being added or a reference being resolved?

As illustrated in this section, there are several serious issues to consider when designing tools to support CSE. The patterns map for CSE is a useful reference for determining system requirements. Once the core requirements of the proposed CSE tool have been defined, reference to the relative patterns is likely to be of assistance during the tool design and development phases.

3.4.4 Collaboration Antipatterns

Software *Antipatterns* [17], are recurring themes of development or design that negatively affect the SE process. One example of a common antipattern is the *God Object*, where an object has too much knowledge about all other objects in the system. This violates commonly accepted programming principles such as data encapsulation and low coupling.

In the context of CSE, some commonly accepted patterns to assist the process of SE may lead to issues under certain circumstances. *Private Worlds*, for example, may be necessary during periods of experimental coding, but will increase the integration effort if used exclusively during a large project.

SE practices are based upon existing tool support. It is envisaged that once CSE tools become commonplace for group development, some currently accepted idioms of SE may become obsolete or superseded. These include *Private Worlds*, restriction of *Development in Pairs* to just two developers, and possibly the organisational pattern *Face-to-Face Before Working Remotely*.

It is conceivable that patterns previously considered as good practice might eventually be reclassified as SE antipatterns in some programming scenarios. There will be times, however, when patterns seemingly orthogonal to CSE will still be required. For example, several teams may occasionally choose to work on separate code bases and integrate their changes back into the main project, regardless of the ability of CSE tools to provide a fully

المنارة للاستشارات
www.manaraa.com

synchronous integration facility.

A discussion is presented in Section 8.1.2 on how varying levels of collaboration, such as in the example described above, can be supported within CSE systems.

The Private Worlds Pattern

The *Private Worlds* pattern is investigated in detail to provide an example of how currently limited SE technology can restrict programming practices.

From Harrison and Copelien, the *Private Worlds* pattern is described as *"... balancing the need for developers to use current revisions, based on periodic baselines, with the desire to prevent developers from experiencing undue grief by having development dependencies change from underneath them"*.

There is undeniably a time and a place for 'Private World' development, using code repositories to integrate off-line development efforts back into the main project branch. The concept of private work has also been identified in Section 3.4.2 as a mode of development that should be supported by CSE tools. The problem is, however, that private work is predominantly the only mode of development for software engineers at present.

The following research findings provide arguments *against* continual protection from "changing development dependencies":

1. Too much time is spent correcting mistakes based from limited communication [15]. Even programmers who work 'privately' spend up to half their time each day collaborating rather than coding [86]

2. The earlier conflicts are detected, the earlier they are resolved [99]. In addition, the longer problems take to fix, the more expensive the software project becomes [104]

3. Programming using private work-spaces is difficult to manage, new users struggle to gain acceptance, and the time to market is slow [51]

4. Merging tools still struggle with concurrently edited source files [70]

www.manaraa.com

5. The concept of private development areas for individual programmers does not scale well [121] [16]. Brooks Law states that "the complexity and communication costs of a project rise with the square of the number of developers" (quoted in [91]).

6. New generation code repositories such as Bit-keeper [10] and CruiseC-ontrol [39] are starting to change the conventions of repository-based SE. They support more synchronous distribution of code by frequently updating all developer code bases with fine-grained units of change

The *Private Worlds* pattern is definitely warranted in times where a low code integration effort is likely. It is unfortunate, however, that private work areas provide the main facility for conventional software development, even in teams where regular and frequent collaboration is encouraged.

I therefore classify the *Private Worlds* pattern as one of several likely antipatterns of CSE if used unwisely. This is due primarily to the limitations that private work areas place on communication and user awareness, and the high effort often required to integrate code back into the main project.

Summary

The work presented in this chapter towards patterns of CSE is consistent with the way in which patterns have been defined in related fields of research. The classification that is presented here is certainly not the only way CSE-related patterns can be grouped, but it does immediately assist in the discussion of requirements for future CSE tools.

One of the most important aspects of CSE patterns is that they provide tool developers with valuable design information that would be otherwise hard to obtain. Being aware of CSE-related patterns allows researchers to focus on getting the fundamental design of CSE tools and supporting facilities correctly. This is significantly different to the costly and often unsuccessful alternative of building tools and then trying to redesign them after user trials and evaluations.

The patterns that have been defined, such as *Mode of Development*, were difficult to identify because professional CSE tools are not yet available and

المنارة للاستشارات
www.manaraa.com

in widespread use. It is likely, however, that these modes of development will be supported by CSE tools as new tools emerge. The rate of uptake of these patterns will be the most accurate determinant of successful pattern identification.

There are undoubtedly other as-of-yet unexplored patterns related to CSE. Many of these patterns, however, might not be exposed until researchers have performed longitudinal studies of a diverse range of groups using highly-functional CSE tools.

Some patterns of conventional software development have been identified as potentially harmful if used in fully collaborative systems. This point is raised to make CSE tool developers aware of potential pitfalls when converting conventional SE tools to CSE-capable ones.

The research of this thesis is aimed to support the patterns of collaboration evident within small groups of software engineers. These patterns are difficult to support with conventional SE tools, therefore attention is focused on the design of flexible and powerful CSE-based tools. The design requirements for such tools are discussed in Chapter 4, and an implementation of a collaborative framework to support such tools is presented in Chapter 5.

المنارة للاستشارات
www.manaraa.com

Chapter IV

Supporting Collaborative Software Engineering

> "As an aside I would like to insert a warning to those who identify the difficulty of the programming task with the struggle against the inadequacies of our current tools, because they might conclude that, once our tools will be much more adequate, programming will no longer be a problem. Programming will remain very difficult, because once we have freed ourselves from the circumstantial cumbersomeness, we will find ourselves free to tackle the problems that are now well beyond our programming capacity."

Edsger W. Dijkstra,
1972

Determining the requirements for CSE tools is an important task. Without clearly identifying the core requirements for CSE tools, it is unlikely that any tool will be met with great success.

In this chapter, the essential considerations for CSE tool developers are presented. These considerations reflect the core features that any CSE tool must support. This provides a context for distinguishing appropriately designed CSE tools from poor ones. In Chapter 5, a framework is presented that demonstrates one way to support these pertinent aspects of CSE tools.

A discussion of tool support for CSE patterns is presented in Section 4.1. In Section 4.2, requirements for tool design and implementation are presented. Threats to successful tool adoption are also discussed. In Section 4.3, semantic model-based SE is described as a mechanism for supporting some of the tool requirements identified. This chapter concludes in Section 4.4 with a discussion of awareness support within CSE tools.

المنارة للاستشارات
www.manaraa.com

4.1 Tool Support for Collaborative Software Engineering

Tool support for CSE and its associated patterns, as identified in Section 3.4, is imperative. Conventional SE tools, however, often provide substandard support for these patterns of collaboration. At present, developers have their coding activity stifled by tools that do not seamlessly integrate the related work of others. Additionally, conventional tools provide only minimal support for various modes of coordination and communication, yet these are aspects of SE identified as significant barriers to development [34].

Examples of the inadequate tool support for CSE patterns are briefly listed here. The *Follow the Leader* pattern can only be applied if the leader's display is relayed to each developer in the team by video-conferencing or specialised software. The *Working in Pairs* pattern can normally only be implemented by sharing one workstation between two co-located users. The *Action/Reaction* pattern can be supported by current tools, but the delay between the action and the reaction is often measured in days, not seconds.

4.1.1 The Need for Better Communication Support

As discussed in Section 2.2, collaboration is a significant factor within SE. From Perry [86], communication consumes approximately half of each developer's time. From Vessy [117], software development activities involve cooperation 70% of the time. Subsequently, breakdowns in coordination and communication are a major development problem, as asserted by Curtis [34].

Estublier claims that frequent updates are necessary for the successful coordination of changes within and between source files [37]. This claim has been strengthened by data produced during a study of large-scale software development by Perry, Siy, and Votta [87]. Clearly, more automated support for coordination of development efforts will reduce the time spent correcting conflicting code modifications.

With the presence of collaboration-aware SE tools, it is possible to identify and analyse the concurrent activity of other users in real time. This provides the potential for errors and conflicting actions to be *proactively* detected and avoided, rather than the *reactive* approach of waiting on eventual failed project builds to initiate costly error correction.

المنارة للاستشارات
www.manaraa.com

Real time, shared development of software between groups of participating engineers also means that the use of code repository systems can be avoided for fine-grained modifications of a project. Given technologies to support real time development, either distributed or co-located, the possibility of merge conflicts is removed altogether. This in turn is highly likely to reduce the overall cost of software development; some evidence of this is provided in Section 7.3.2.

4.1.2 Common Tool Design Approaches

There have been many types of CSE tools constructed previously, as presented in Section 2.5. The following common types of design approaches are identified, accompanied by their inherent limitations:

Conventional Tool Augmentation A common approach to supporting CSE is the augmentation of conventional SE tools with collaborative services. Palantìr [98], for example, does this by analysing code repository activity and reporting potential configuration management conflicts back to each user in real time.

This approach of observing code repository information, however, restricts feedback to the detection of potential conflicts between source files. It is not possible to detect semantic errors until the source code has been committed back into the repository. Additionally, the granularity of feedback information is governed by the frequency of repository updates, which can be very irregular. There are also many other issues with converting single-user tools to being collaborative, as discussed previously in Section 2.6.1.

Custom Tools Several collaborative tools have been constructed specifically for certain SE tasks. Tukan [100], for example, provides user presence information as developers work on files from the same source code repository. Rosetta [48] allows collaborative web-based construction of UML diagrams.

While these tools are near ideal for the given task, their resultant restricted and inflexible nature prevents them from gaining wide-spread

المنارة للاستشارات
www.manaraa.com

acceptance. The language grammars are typically hard-coded into the tools, which means that support for new or multiple languages is difficult to implement. Additionally, as these tools are often limited in design, they do not scale well in terms of number of concurrent users and size of the working project.

Workflow Systems To control an overall group process, workflow systems such as The Coordinator [19] have experienced varying degrees of success. Within a CSE setting, workflow systems such as Visual Studio Team System [73] have been used to coordinate the efforts of application developers according to the project plan and prescribed development process. Quality assurance systems such as Bugzilla [76] also employ workflow mechanisms to coordinate testing and bug fixing.

As workflow systems enforce specific processes by their very nature, however, it is difficult to envisage a workflow-centric system that supports software development down to the coding level—far too much development activity is unplanned and volatile in nature. The Coordinator, for example, failed to accommodate daily variances in project plans, which resulted in strong user resentment and resistance, even when applied to a general workflow context.

IDE Integration Professional IDEs such as Eclipse [83] are fully featured, and if they are extensible by way of a plug-ins interface or are open-source, it is theoretically possible to convert them into rich CSE tools. Such IDEs have semantic models that can be employed to analyse relationships between users making concurrent modifications, and already have large user bases.

To make the transition from conventional to CSE tools, candidate IDEs are likely to require conversion from code repository-based collaboration to fully-synchronous artifact sharing systems. This conversion, however, is a major task that requires a large development effort. Subsequently, all collaboration support for IDEs to date simply involves the augmentation of inbuilt code repository facilities, as in the Jazz

المنارة للاستشارات
www.manaraa.com

project [21], or collaboration is limited to high-level functions such as chat and shared white-board applications, as in the ECF project [66].

It is apparent that regardless of the development approach taken, designers of CSE tools face difficult problems to solve. The resolution for most of these problems can be provided by fully modelling source code and its inherent relationships explicitly, and providing shared real time access to this model. The concept of shared semantic modelling is discussed further in Section 4.3.

4.2 Considerations for Tool Developers

Despite recent technological advances in distributed systems technology and desktop processing power, no single system exists that solves all the current challenges in supporting CSE. Instead, there has been a proliferation of prototype tools that support specific SE tasks [92, 98, 41, 11, 100, 48], and subtle collaborative enhancements have been made to existing commercial single-user tools [66, 12, 115, 21].

In this section, the key aspects that CSE tools must address in order to satisfy the requirements of SE in the large are discussed. The key questions are: given CSE tools that operate potentially in real time on a shared set of evolving SE artifacts, what are the changes from the perspective of the developer, and are these changes acceptable?

4.2.1 Tool Design

In the previous chapters, the need for greater collaborative support for SE has been demonstrated. CSE tool design, however, is a difficult and challenging task—many aspects must be considered in order to provide successful facilities for CSE. Additionally, when designing a CSE tool, initial investigations may not necessarily reveal the full set of requirements essential for adequate tool construction.

The main aspects for consideration when designing a CSE tool of any type are identified in this section. While not necessarily complete or exhaustive, the list has been derived through lessons learned during extensive CSE tool

المنارة للاستشارات
www.manaraa.com

design, testing and evaluation. The criteria given in the following list have been grouped into three categories: *Tool, Task* and *People*. The aspect of collaboration is dispersed throughout each of these categories.

This list is provided as a useful discussion document, providing CSE tool designers with means to define their own set of requirements as they develop specific tools.

Considerations for Supporting CSE Tools

The following considerations are applicable to the design of any CSE tool.

Management of Artifacts Which types of artifacts are to be shared? How will they be stored? Will a modification history be kept?

Mode of Change Integration Is a pessimistic or optimistic locking scheme employed to control artifact modification, or is some form of real time artifact sharing possible? If so, what floor control policies are in place to manage collaboration? Are private work facilities available?

Multiple Language Support Should the CSE tool support more than one language? Are languages restricted to a particular paradigm such as OO languages?

Multiple Views of Artifacts Are multiple views of artifacts supported, such as viewing of a source file as both code and as part of a class diagram? If multiple views are supported, is the mapping mechanism capable of fully catering for each view, and translating between views?

Extensibility Should the CSE tool support extensibility and customisation? If multiple views of artifacts and multiple languages are supported, how can new views and languages be added?

Semantic Model Construction Will a full semantic model of the shared software's entities and relationships be constructed? Or, will the primary source of tool information be pattern matching and other heuristic approaches that scan source files for named tokens and declarations?

www.manaraa.com

If a semantic model is constructed, will it replace the source files as the authoritative repository of information for the project, or will source files be annotated with semantic model markup tags? Additionally, can parts of the semantic model be locked for concurrency control as well as source files?

User Presence Feedback Should the CSE tool be capable of detecting overlapping areas of modified code based on semantic relationships? If so, how is this to be supported?

Impact Reporting Should the CSE tool be capable of immediately detecting a change in the program state, such as a reference being resolved or broken, immediately after the modification is made? If so, how is this to be supported?

Usability How should the CSE tool deliver feedback to the user? Will such feedback be embraced by the user or seen as a hindrance? Can feedback be customised or suspended by individual users?

Communication Media Richness Which types of collaboration media are available within the CSE tool? How well-suited is the richness of the communication facilities in comparison to the complexity of the development task supported by the tool?

Workflow Is the CSE tool expected to interface with third-party services such as bug-tracking databases, documentation libraries, component libraries and workflow/project management systems?

Considerations for Supporting CSE Tasks

The following considerations address key aspects relating to the types of tasks, independent of the actual tool type.

Task Type Which types of SE tasks are to be supported? Tasks likely to be supported in CSE tools include requirements gathering and analysis, system design, implementation, unit testing, program maintenance, validation and verification.

المنارة للاستشارات
www.manaraa.com

Task Complexity Can a range of tasks be supported in terms of complexity, or are only simple tasks such as code reviews to be supported?

Task Size What is the size of a typical task supported by the CSE tool, in terms of terms of number of files, classes, packages and lines of code?

Task Duration What is the length of a typical project that uses the CSE tool? For fine-grained tasks, how long will users keep the same set of files open for? The duration of fine-grained tasks will impact on the choice of concurrency control within the CSE tool.

Task Process Which development methodologies should be supported? Is the development process open source, where there are often no time pressures and heavy moderation of changes, or is a closed-source and highly coordinated approach more likely? How many modes of development are likely to occur? Is the CSE tool focused on a specific methodology such as XP or RUP? If so, are additional methodology-specific considerations required?

Considerations for Supporting Developers

The following considerations are independent of tasks and tools—they address aspects of typical tool users.

Group Size What number of people are likely to be supported? What is the maximum number of people likely to be working closely together on the same subset of artifacts within the project?

Culture What are the ability levels of each developer? Should a mix of abilities be supported? Does a culture exist within the team where certain informal social processes are likely to be followed, such as posting code update notifications to a mailing list?

Roles Are there predetermined roles within the development group, such as moderators, project managers and analysts? If so, should the CSE tool explicitly support such roles?

المنارة للاستشارات
www.manaraa.com

Location Are the developers in a face-to-face and constantly co-located setting, or are they distributed throughout several departments or organisations?

Time Will developers typically work at the same time, different times, or a combination of both possibilities?

For various CSE tools, some questions raised in the above list of considerations will matter more than others. For example, a collaborative code editing tool might need to accommodate multiple languages, but may not be concerned with the representation of multiple views. A sequence diagramming tool might only be interested in the semantic model, and has no concern for other artifacts or specific languages.

In producing the above listing of tool considerations, it is clear that there are many aspects to CSE tool design, and that the features a CSE tool is likely to support must be planned from the earliest stages of tool design. For CSE tools that are intended to be very general and scalable, it is important to ensure that each consideration in the above list can be supported. If a CSE tool can not support different types of tasks, artifacts and group sizes, then perhaps the tool is not as applicable to CSE as originally intended.

Many of the aspects of CSE tool design presented in the above list are discussed in further detail during subsequent chapters in this thesis.

4.2.2 Requirements for Large-Scale Development

For the CSE researcher, it is not a trivial task to produce robust and well-designed tools. From concept to evaluation there will be countless bugs, design faults and unexpected limitations to encounter. While it is tempting for the CSE researcher to only produce prototype tools, the aspect of how to build robust and scalable CSE tools must be addressed for SE to progress. This is the motivating factor for the development of the Caise framework.

In order to produce realistic CSE tools of any nature, the developers of the tools will need familiarity with concurrency control, distributed systems, source code control systems, parsing and semantic analysis of program code,

المنارة للاستشارات
www.manaraa.com

human-computer interaction and design factors, and performance optimization just to name a few areas. CSE tools must also support most if not all of the tool functions listed in Section 4.2.1.

CSCW challenges surrounding CSE tools are immense. When writing collaborative systems, nearly every core feature requires additional, complex functionality. Using collaborative text editing as an example, what should happen if two users type a keystroke at the same time into a shared document? If one keystroke effectively cancels out another, how should this be actioned? Similarly, what should happen if one developer is half way through declaring a new method in a text editor, and a second developer moves the containing class to another package through a class diagramming tool? Is one user's set of actions lost, or can both users' coding efforts be preserved?

Given the complexities of CSE tool design, perhaps it is not surprising that token-passing floor control policies, where only one user can edit a region of code at a time, are common within the few commercial tools that support collaborative development.

Industrial-strength tools are difficult to construct from other aspects as well. Semantic modelling is valuable within a CSE tool architecture, as explained in Section 4.3, but the construction of a semantic model requires significant effort. Expertise is required in parsing and code analysis, and working through a language's grammar to construct an accurate semantic analyser is a tedious, difficult and time-consuming exercise.

Standard software development kits, such as the Java SDK [113] and IBM's JIT compiler [107], may also struggle to compile the complex source code that CSE tools consist of. Additionally, code libraries, virtual machines and operating systems may also struggle to load and execute CSE tools due to the huge volumes of data associated with large software projects. Converse to this is the orthogonal requirement of low-latency responses to user events such as file modifications.

The difficulties in implementing robust CSE tools have received little emphasis in the literature related to CSE. Researchers, however, need to be drawn to the fact that implementing real tools is a non-trivial task that requires careful attention to design, considerable expertise, and a pool of capable and willing programmers.

المنارة للاستشارات
www.manaraa.com

4.2.3 Threats to Tool Acceptance

There is a risk that when any large system is developed, its uptake may be less than what the designers had envisaged. For example, a code editor that will be used on a daily basis is unlikely to gain acceptance if the quality, functionality and look-and-feel do not meet the majority of any given user's preferences.

For SE tools, users are particularly critical of user interfaces. Individual programmers have strong opinions on which code editor and UML diagramming tools they prefer, and migrating to any new set of tools requires motivation and training. Even if new types of tools are ultimately more productive, the incentive of long-term gain is unlikely to mitigate the cost of tool usability dissatisfaction by the group of core users.

CSCW research particular to SE has produced findings that tools must be suited to the needs of the users. Gutwin, Penner, and Schneider, for example, interviewed experienced open-source developers within the NetBSD project [51]. One interesting finding from this study is that the team of experienced developers were not particularly enthusiastic towards new kinds of activity awareness tools—rather the developers were already indoctrinated into reading newsgroups and using social protocols for supporting user awareness, and perceived no strong need for the tools. The participants in this study suggested, however, that the tools might be of a higher value for less experienced or new developers to the group.

An effective way in reducing threats to CSE tool acceptance is that of heuristic evaluations. Heuristic evaluations for CSE tools are discussed in Section 7.1. The principle of this type of evaluation is to constantly check tools as they evolve against a set of accepted evaluation criteria. This reduces the cost of large, formal evaluations at the end of the CSE tool development cycle—where it is very expensive to make any types of changes—and it also ensures that the CSE tools will reach suitable standards of design and performance early in the construction process.

www.manaraa.com

4.2.4 Future Tool Design

A simplistic approach to further CSE research would be the development of more prototype tools that fulfill specific niche requirements. It is of concern, however, that prototype tools are considerably different from those of which real software engineers will use in practice.

The research in this thesis towards CSE is not based around building yet another set of tools and finding areas where they might produce favourable subjective results over conventional tools. Instead, the failings of previous CSE tools have been carefully studied in order to determine the set of requirements that any successful future CSE tool is likely to conform to.

Beyond the requirements given in this chapter, such as support for CSE patterns, it is apparent that any successful CSE tool must be of a high quality. Any tool that has response latencies, user interface design faults, or is not robust and reliable will inevitably face rejection from its users. Similarly, the behaviour of CSE tools must be similar to that of conventional tools; any change in fundamental behaviour could make the tool learning curve too high. Ideally, CSE tools should give the appearance and behaviour of single user tools when only one person is actively working on the project.

In order to provide such tools, a complete semantic model of the software being developed is essential for most purposes. As to be discussed in the next section, a semantic model, once constructed, is an efficient and effective way to provide rich feedback information, allow extensibility of tools and languages, provide fast response times to modification requests, allow multiple views of artifacts, and support accurate refactoring.

4.3 Semantic Model-Based Software Engineering

To adequately support the patterns of CSE observed within team development, tools require more than just source files to provide complete program construction information. While source files are an important means of input for a software product, the technology and computational power available today allows for far richer analysis of software. Similarly, to support complex interactions between multiple tools, source code is not an ideal means of information interchange.

المنارة للاستشارات
www.manaraa.com

Source code is full of implicit relationships that require thorough semantic analysis in order to transform a sequence of characters into useful SE information. Powerful IDEs almost certainly require internal construction of a project's semantic model to analyse code changes and to provide features such as code completion and class hierarchy browsing. For example, Eclipse [83], Netbeans [114], and Together Architect [46] all have inbuilt comprehensive semantic analysers to produce a full model of the project's software.

Symbol tables [1] are the simplest type of semantic model; these are used predominantly by compilers to convert Abstract Syntax Trees (ASTs) into machine or byte code. While symbol tables are an accurate source of information for compilers, richer types of information are required for tools that support automated refactoring, metrics analysis, and querying of the code base.

A full semantic model of the software project where rich types of information are available explicitly, such as the relationships between all program declarations, is an extremely useful asset for most SE tools. A typical semantic model of software contains representations of all of the declared entities such as classes and methods, along with a map of all relationships, such as all subclasses for a given superclass and all method invocations of a given method declaration. To assist in illustrating what a semantic model encompasses, a simplistic example of a semantic model is presented in Figure 4.1.

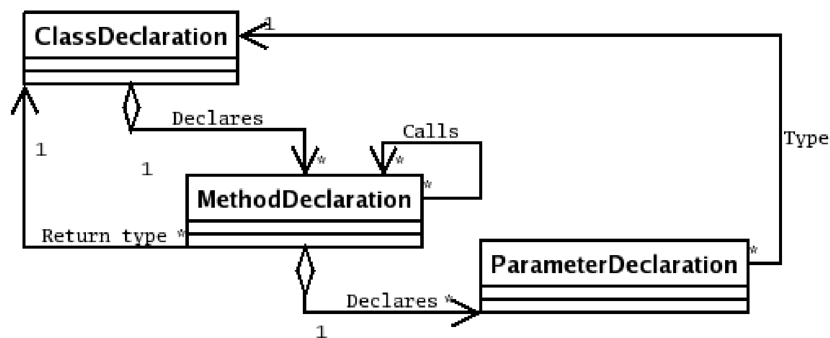


Figure 4.1: A UML class diagram for a simplistic semantic model of software.

المنارة للاستشارات
www.manaraa.com

To fully model of OO software, the corresponding semantic model design will contain a large number of classes and relationships. A semantic model for Java 1.4, for instance, is presented in Section 5.3.1. This model, capable of describing any compilable set of Java source files, contains approximately thirty core classes and over a hundred relationships between them.

For CSE to be fully supported, tools that operate on semantic models of projects are likely to be required. It is possible to derive program information through heuristic approaches such as pattern matching of tokens, but for OO software, this type of syntactic analysis alone may produce incorrect results. Only with full semantic modelling can tools determine with confidence the relationships between concurrent modifications, and the impact of pending changes.

The technique of using a shared semantic model between developers is substantially different from all other approaches that I am aware of. For example, even though IDEs use a semantic model to provide rich functionality to each user, they still revert to file-based code repository systems to accommodate change propagation between developers. No attempts are made to inform pairs of developers about overlapping areas of related code, or the effect that local modifications will have on the very latest version of the project.

In Section 4.3.1, the implications of building a semantic model for a software project are discussed. In Section 4.3.2, a discussion on how a project's semantic model can be shared concurrently is provided. In Section 4.3.3, a discussion is given on how relationships between pairs of users are identified.

4.3.1 Constructing a Semantic Model of Software

The modelling of a project's semantic relationships is a difficult task. All source files must be parsed, and a semantic model must be constructed that records every component within the software, from packages and source files down to parameters within methods and local variable declarations. Additionally, all relationships must be determined such as method invocations, inheritance, method overloading and polymorphism.

Even once a means for semantic modelling is in place, SE tools such as

المنارة للاستشارات
www.manaraa.com

IDEs must also be able to accommodate incremental updates to source files and other SE artifacts. Therefore, a semantic analyser is required and must be able to accept continual changes to the underlying model via new versions of source files and parse trees or direct modification commands. This also implies that the semantic analyser must be able to reconstruct the semantic model in real time.

Once facilities are in place to perform semantic modelling, CSE tools can offer the following functionality:

- Feedback on relationships between code components, such as the callers of any given method
- User presence calculations between each pair of programmers, based on their current location within the semantic model
- Immediate modification impact reports
- Accurate metrics of any kind
- Mappings between different views of the semantic model
- Pretty printing and formatting of source code based on the current state of the semantic model
- Fast and efficient refactoring of semantic model components

By using a semantic model to offer this functionality, tools can be assured of correct results regardless of the operation. The same can not be claimed by artifact-based pattern matching techniques. An example of the failings of pattern matching could be a refactoring operation that renames a method—unrelated calls to a method of the same name outside the selected method's lexical scope may be incorrectly be renamed as well.

A complete discussion of semantic modelling for OO software is presented by Irwin [58].

المنارة للاستشارات
www.manaraa.com

4.3.2 Sharing the Project Model

Regardless of the type of CSE tool constructed, the research project presented in this thesis demonstrates how the semantic model of software can be used as the authoritative source of all project information. This is the underlying fundamental element of the Caise approach. CSE tools that use this approach can be classified as *semantic model-based* rather than artifact-based.

Given semantic model-based tools, it is the semantic model that should be shared by all tools and users within the project, effectively making the artifacts themselves simply views of the underlying semantic model. If real time sharing of the semantic model is supported, any number of different views can be supported, with changes in one type of artifact propagated to all other views immediately.

As described in Section 4.1.1, by sharing and updating the semantic model in real time, merge conflicts are avoided. Unnoticed transactional conflicts, such as when two developers mistakenly break a code dependency because of conflicting tasks, are also less likely due to the possibility of feedback messages highlighting current dependencies within the given scope, and immediate notification when such dependencies are broken. By sharing the project's semantic model, CSE tools also receive all the other benefits described in Section 4.3, including deep metrics information and accurate refactoring mechanisms.

The essential concept of semantic model-based tools is that instead of sending entire batches of source files at a time back to a central repository, tools simply report what they are doing at a fine-grained level to the semantic model. Parsers and analysers, normally housed within the semantic model, can convert tool actions into semantic model modification actions. Changes to the semantic model are then propagated out to all other participating tools in the project. As long as the tools update the semantic model at a fine granularity, there is little or no chance of conflicting actions, at least at a syntactic level.

Despite the relatively simple idea behind sharing a project's semantic model of software, it is considerably challenging to design and implement

المنارة للاستشارات
www.manaraa.com

semantic model sharing if fully synchronous tools such as shared text editors are involved. If the update granularity is too coarse, there is a risk of transactional conflicts, such as a method being renamed by a diagrammer just before a text editor submits the method body—this would result in a loss of work if not specifically guarded against.

An illustration of how the CAISE framework shares its semantic model and maps between different views is given in Section 5.3.1. The CAISE framework is the first known CSE system to use the approach of a shared semantic model of software between tools.

4.3.3 The Code Neighbourhood

Given the ability to atomically integrate code changes as they occur, as discussed in Section 3.4.2, and the ability to translate artifact modifications into semantic model translations, as discussed previously in this section, all users effectively work on the single instance of the project in real time. With this comes the ability to immediately detect areas of interest that are common to a set of users as they navigate through the software under development.

An example of a shared area of interest within software is presented in Figure 4.2. In this example, user Carl is editing the method named `update()` within class `AnimatedSprite`. At the same time, user Wal is editing properties within the class `DynamicSprite`, which is the superclass of `AnimatedSprite`. The superclass of Wal's class is named `Sprite`, which currently has no superclass declared.

With or without the presence of tools that operate on a shared semantic model of the project's software, it is clear that there is an overlap between the proximities of Carl and Wal at this point. Both users should be closely coordinating their actions, as any modification that Wal makes could have a significant impact on the class that Carl is editing. For example, if Wal changes any of the properties in `DynamicSprite`, these changes are immediately inherited by `AnimatedSprite` due to the semantics of OO languages. Similarly, if Wal declares a method named `update()` in the `DynamicSprite` superclass, this may change the number of invocations made to the method named `update()` that Carl is currently working on.

المنارة للاستشارات
www.manaraa.com

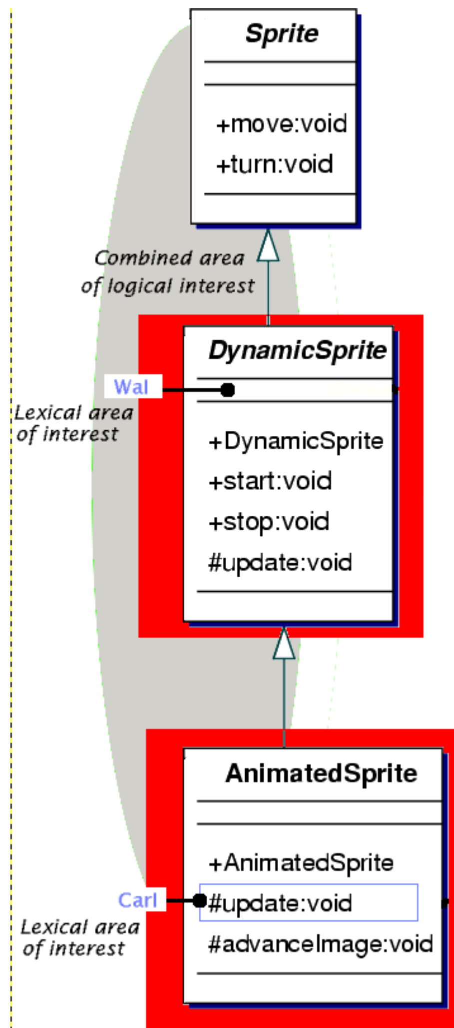


Figure 4.2: The combined code neighbourhood for two developers, using UML notation.

An appropriate term for this area of related code is a *Code Neighbourhood*. A code neighbourhood is the entire region of code that is semantically related to a user's current area of focus. This can also be viewed as the scope of effect for any given point in the software project, taking into account lexical scope, inheritance, method invocations, composition and all other identifiable semantic relationships within the project's software structure.

If real time support exists for a shared semantic model of a project's software, the identification of the code neighbourhood for any given user

المنارة للاستشارات
www.manaraa.com

can be calculated immediately as the user navigates from one section of code to the next. Without atomic integration of changes, only the code neighbourhood for the last committed version of the project's source files can be calculated.

Given that the code neighbourhood for any user can be calculated in real time for CSE tools that use a shared semantic model, only a simple calculation is required to determine if two users are within a connected region of code, or in other words, are semantically related to each other. This is a very important advantage of real time CSE tools over their conventional counterparts: developers can be alerted to overlapping areas of interest immediately, rather than on reflection during conflict resolution in response to a failed repository check-in or build.

Use of the Code Neighbourhood

The ramifications of being able to automatically calculate code neighbourhoods and inspect for overlapping areas of interest are great. As the example presented in Figure 4.2 suggests, for a complete understanding of the critical areas related to any one line of code, tools need to be aware of the logical composition of the project, not just the declarations contained within the lexical scope. In fact, tools may need to look much further than the immediate logical structure such as the inheritance hierarchy; tools often also need to identify which other parts of the system depend on the main classes in focus, and in turn, which areas on the project these classes depend on themselves.

In OO software there are many implicit and subtle but important relationships to identify and understand. Even the most proficient groups of developers will occasionally make incorrect modifications to a project because a subtle dependency between units of code was overlooked. This is why tool support for shared code neighbourhoods is important—developers do not necessarily have to maintain a mental picture of the entire semantic model, rules of the language and current locations of all other users; CSE tools have the potential to proactively provide context-specific information on related areas of code.

المنارة للاستشارات
www.manaraa.com

It would be very challenging to predict conflicting modifications between related areas of code before they happen, but CSE tools can alert users to the semantic proximities of others in real time. This is a notable difference between conventional, repository-based tools and real time semantic model-based CSE tools.

User Interface Support

In Section 6.2.3, a text panel is demonstrated that provides extensive user presence information between pairs of users based on the semantic model of any CAISE-based project. Multi-user widgets to augment SE tools are also presented in Section 6.2.3. Again, these components operate on information related to the code neighbourhood rather than physical proximities of users.

Other tools such as Palantír [98] and Tukan [100] also provide a degree of code neighbourhood information, but within the CAISE framework, information is based on a full semantic model, giving the ability to represent even the most subtle types of relationships in real time.

4.4 Awareness Support

There are many different types of information that developers may be interested in during the course of any software development phase. From the previous section, it is clear that semantic model-based tools can provide feedback on current modifications, the actions of other users within the project, and metrics information. In this section, the main types of information available for CSE tools are presented, and a discussion is given on the implications of presenting these types of information to the user.

4.4.1 Types of Awareness

Regardless of the underlying architecture, many types of information can be generated as developers work on a set of given tasks collaboratively. CSE tool developers need to consider which types of information are important to their users for subsequent tool integration. The following list outlines the range of different types of information that developers might be interested in. This range has been categorised into *view, semantic model*, and *workflow.*

www.manaraa.com

View Awareness

View awareness represents feedback related to the modification of a SE artifact that does not immediately or directly map into a modification of the project's underlying semantic model.

Physical Proximity The physical locations of other developers within the software project is an important aspect of awareness. Physical proximity refers to the distance from one developer to all units of code being edited by others at any given point in time. For a text editor, the physical locations of a user are simply the cursor positions and areas of current focus within each opened artifact. For a class diagram, the physical locations of a user are likely to be the currently selected methods or classes within the diagram.

View Modification The modification of a view within a CSE tool, without necessarily affecting the underlying semantic model, is a common operation. An example of this would be changing the layout of classes within a UML diagramming tool. Another example could be the moving of method declarations within a source file.

To support the propagation of changes in views, a decision in advance needs to be made on whether the view is shared between all users, or each tool is responsible for its own view. For shared views, change events need to be propagated to all other CSE tools within the project, allowing them to integrate this change with their own views. For tools that allow individual views, a separate mapping must be maintained for each tool, and changes in local views will not normally require propagation to any other parties.

Textual Modification An elementary type of feedback within CSE tools is that of textual modification, independent of any underlying semantic change to the software. For example, if two users are editing the same source file, how are these edit events propagated between views? Are views updated using keystrokes as the atomic unit of action, or are displays updated only after a fixed time period or burst of activity? If

المنارة للاستشارات
www.manaraa.com

textual modification is not propagated in a fully synchronous manner, then a conflict resolution facility might be required in the event of interleaved modifications.

If a strict floor control policy is used such as token passing, the immediate propagation of textual modification events is relatively easy to support. Alternatively, if fully synchronous editing of code is provided with immediate propagation of changes, a reliable model-view-controller design approach is likely to be necessary.

Code Neighbourhoods Within a software project, many types of semantic relationships exist such as inheritance, association and aggregation. When two or more developers are modifying or inspecting units of code that are semantically related, a transient relationship now exists between the developers, termed as the code neighbourhood. Given an overlap between code neighbourhoods, the developers must apply their changes with caution and greater communication, otherwise a transactional conflict is possible during periods of concurrent modification.

For direct semantic relationships, such as a superclass/subclass pairing, it is wise to avoid making concurrent modifications within the two classes due to the high coupling and inter-relationships. Other types of relationships, however, are more subtle than this and not immediately noticeable. For example, renaming a property in one class of a given package might cause a problem for a method in another class in a different package that has access to this property through an inherited superclass. While semantic analysis can detect this type of relationship, a DOI mechanism might be required to restrict the number of peripheral relationships identified between two or more concurrent developers.

Semantic Model Awareness

Semantic model awareness represents feedback related to a change in the semantics of the software project.

المنارة للاستشارات
www.manaraa.com

Semantic Changes Independent of the tool type, eventually a sequence of events will lead to a modification of the underlying semantics of the project's software structure. A new statement from within a text editor might add a method call. A drag and drop event within a class diagramming tool might establish a new inheritance relationship.

In some situations, changes to the underlying semantics of the software may not require reporting. For example, if a new method is added to a source file through a text editor tool, it is questionable whether any other types of tools require feedback above that of the new method coming into view. In other situations, developers and project managers might be very interested in significant changes in a project's semantics, particularly for projects that are mature and are not expected to undergo any further substantial development.

Impact Reports Beyond the actual identification of a project modification, it is also desirable to be able to immediately discover the effect of the modification. It is useful to know if a modification, such as renaming a method without refactoring the existing calls to that method, has broken the project. Project modifications that result in a fix to outstanding compilation errors are equally as important.

Aside from compilation errors either being introduced or removed, an additional type of impact report-based awareness is that of syntactically and semantically correct modifications. When semantically related units of code change, it may also be of interest to developers. For example, a developer might be working on a given method. If a second developer makes a legal change to a class elsewhere that the method depends on, both parties might want to talk about the modification regardless of whether or not the compilation state was affected.

Software Metrics Analysis of metrics can be useful to highlight both good and bad areas of software design. Traditionally, software metrics are calculated as a batch process at the end of a development phase or upon code integration. Within CSE tools, however, it is possible to calculate

المنارة للاستشارات
www.manaraa.com

metrics in an event-based manner whenever the state of the software changes, with the information propagated to relevant developers.

For some developers, notification of changes in software metrics values will be immediately useful. For other developers, more subtle types of feedback might be required such as background visualisations and cues.

Test Case Results Automated testing of software is becoming increasingly popular. Regression testing, for example, applies a common set of unit tests to software on a regular basis, and if test results differ from the expected values, a warning is issued.

Given CSE tools, testing could be automated upon changes to a shared semantic model, or the conventional approach of batch-testing could still be used. Regardless, information about failed tests is certainly worth considering for integration within CSE tools. If users are working within an area of code where a test case has recently failed, it might be in the developers' interests to be made aware of this.

Similarly, if the data used within the test cases has been changed, again, users working within areas of code related to the affected tests may require notification of the change.

Workflow Awareness

Workflow awareness represents feedback related to the modification of SE artifacts not directly involved with the project's semantic model.

Bug Catalogs By the very nature of software bugs, often no direct, detailed relationship exists between entries in bug tracking databases and their related areas of code. But, for documented bugs that can be attributed to specific packages, classes or methods, it is possible for tools to generate feedback relevant to users who enter those units of code. This type of feedback may become particularly useful when the bug documentation and its related units of code are being modified concurrently by independent developers.

المنارة للاستشارات
www.manaraa.com

Documentation Changes Documentation libraries such as Javadoc have direct relationships with the code, diagrams and underlying semantic model of the project's software. Again, changes to the documentation might have important ramifications for users of the related code components, and vice versa. If a developer is editing or accessing a method, class or field that is published within a document library, and the documentation is also currently under modification, immediate and directed feedback on changes to that documentation might be valued.

As can be seen, the range of different types of feedback information is great. It is unlikely that any one tool will need to support all of these different types. Identification of the main types of awareness information to be supported for a given CSE tool, however, is essential during the design phase.

4.4.2 Media Richness

Once the types of feedback relevant to the CSE tool have been determined, presenting the information to the user in a correct and acceptable manner is a challenging task. Work towards facilities to support appropriate means of feedback within collaborative systems continues today, particularly within projects such as GroupKit [95]. Considerations include not unduly interrupting the user, making the information as relevant as possible, minimising the amount of screen space required to present feedback, and allowing for customisation of the level of feedback.

A topic closely related to the support of appropriate CSE tools is that of *media richness*. This concept, as introduced by Reichwald, Moeslein, Sachenbacher, Englberger, and Oldenburg [93], is presented in Figure 4.3. From this figure, it is apparent how the richness of the presentation media must be matched to the complexity of the task, otherwise an over-complication or under-simplification can take place. While the concept presented in this figure is for general collaboration, the principle of matching the media to the task is also true for computer-supported CSE.

As an example of how media richness should be considered within the context of CSE, a simple task to edit a few lines of code that is completely

المنارة للاستشارات
www.manaraa.com

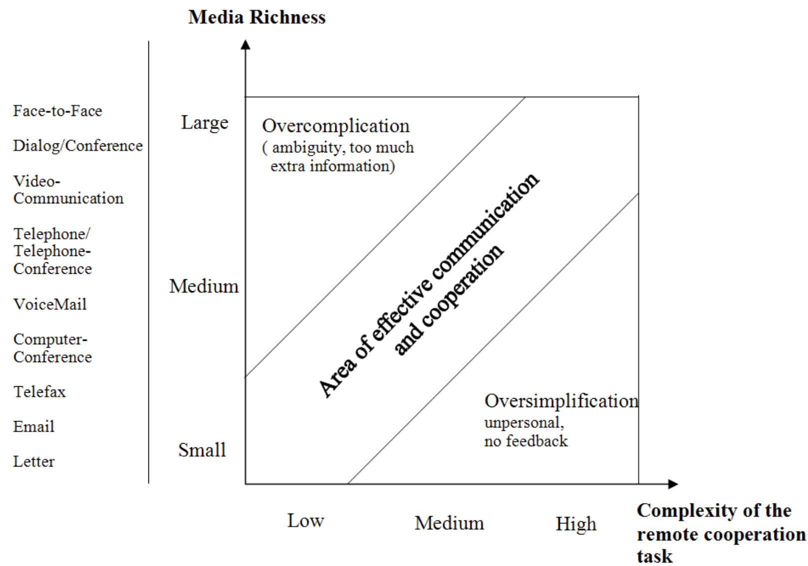


Figure 4.3: Media Richness Theory: reducing ambiguity by media selection.

independent from all other concurrent changes does not require vast amounts of unrelated feedback about the actions of other users. Conversely, it is argued that within conventional SE tools, the media is not rich enough to convey the full implications of complex software modifications within a group project [99]. For the CSE researcher, tools must be designed where the levels of feedback match the task at hand.

4.4.3 The Collaborative Spectrum

Real time collaboration within SE tools is gaining popularity: IDEs to add synchronous support for collaboration during 2005 alone included Eclipse [66], Borland's JBuilder [12], SubEthaEdit [85] and Sun's JSE [115].

In a manner similar to the determining the correct types of feedback and levels of media richness, CSE tool developers also need to determine just how collaborative their tools should be. Again, there are a range of choices.

A spectrum of possible levels of collaboration is given in Figure 4.4. At one end of the spectrum, tools are completely free, such as shared white-board applications. These types of unrestrained tools typically work on unstructured and transient documents.

المنارة للاستشارات
www.manaraa.com

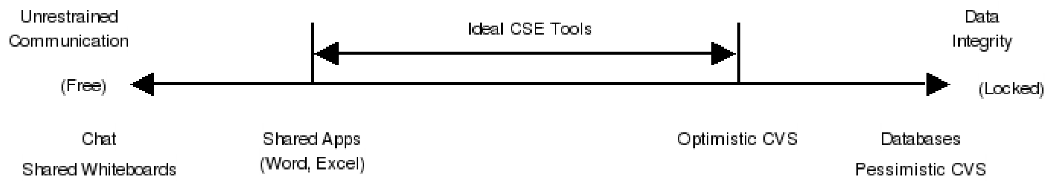


Figure 4.4: The collaborative spectrum of software engineering.

At the other end of the spectrum, tools may maintain artifacts that are completely locked. These types of tools work on persistent artifacts that typically have a rigid structure such as database records or source files. Such tools can usually guarantee the integrity of the documents that they operate on, but these tools struggle to provide synchronous file sharing.

On this spectrum, it is also shown where CSE tools are likely to be placed in terms of collaboration. For typical CSE tools, some control over the granularity and order of collaborative modifications may be required to avoid situations of undesired mêlée. However, it is not desirable to force tools into completely locking artifacts, otherwise difficulties related to reduced awareness are encountered, including conflicting modifications and merge conflicts.

Summary

The purpose of this chapter is to promote the careful consideration of all aspects pertinent to CSE tool design. The construction of adequate solutions for CSE tool support is preferred over the mechanical development of endless numbers of CSE tool prototypes that have limited chance of success.

A key conclusion drawn from this chapter is that there are many design aspects to be considered when constructing genuinely useful CSE tools, based on patterns evident within CSE. The design approach taken to accommodate large numbers of developers and code bases requires careful balancing, with equal consideration of conflicting factors such as awareness of others versus uninterrupted modes of work.

In the remainder of this thesis, it is demonstrated that powerful real time CSE tools can be constructed rapidly given a supporting framework, and

المنارة للاستشارات
www.manaraa.com

that such tools have measurable benefits over conventional SE tools. The framework-based design approach to supporting CSE tools is described in Chapter 5. In Chapter 6, several CSE tools are presented and discussed. CSE tool evaluation is presented in Chapter 7.

للاستشارات المنارة
www.manaraa.com

Chapter V

The CAISE Framework

> "The vision is that, well before the seventies have run to completion, we shall be able to design and implement the kind of systems that are now straining our programming ability, at the expense of only a few percent in man-years of what they cost us now, and that besides that, these systems will be virtually free of bugs."
>
> Edsger W. Dijkstra,
> *1972*

In this chapter, the CAISE framework is introduced. In Section 5.1, the need for a framework to construct realistic CSE tools is discussed. An overview of such a framework, CAISE, is presented in Section 5.2. The architectural design of CAISE is detailed in Section 5.3.

5.1 The Need for a Better Tool Support

The authors of the Concurrent Versioning System (CVS) say *"CVS is no substitute for communication"* [9]. This statement reflects the fact that code repository systems are not designed to support communication, cooperation and coordination of tasks.

The goal for CAISE-based CSE tools is to allow programmers to work collaboratively without sacrificing communication. Communication is important to avoid coding conflicts, share ideas and resolve problems. CAISE achieves this by keeping all programmers synchronised in real time, and at the same time providing user awareness and project state information to individual tools. CAISE-based tools support what code repositories do not provide: communication between developers and tools during fine-grained real time collaboration.

المنارة للاستشارات
www.manaraa.com

CAISE-based CSE tools operate by exposing all developer actions, such as source code modifications, immediately. For developers working on well-partitioned SE tasks, this allows merge conflicts to be avoided, and transactional conflicts can be detected immediately. For developers working on the same artifact concurrently, this forces each tool's view of the artifact to be immediately updated upon any modification.

While it may seem distracting for some users to work in a fully-synchronous mode, the premise of my research is that immediate awareness of the actions of others promotes good SE. This premise has been asserted elsewhere [99], and in Section 7.3 I show that immediate propagation of changes between developers raises no significant usability or coding issues between pairs of co-located programmers working on several common coding tasks, even when editing the same lines of source code concurrently.

5.1.1 Motivation

Many prototype CSE tools, such as those discussed previously in this thesis, are well suited to a single task or development methodology. Unfortunately, they are for the most part fixed and non-extensible, despite the considerable development efforts during tool construction. A key objective of the research in this thesis is to reduce the barrier of high construction costs for CSE tools by providing a framework that enables many different types of CSE tools to be developed rapidly. This objective is met by separating concerns such as tool functionality, user awareness mechanisms, parsing and semantic analysis, and concurrent artifact modification.

To produce more comprehensive SE tools, there has been much recent development towards collaborative add-ons and toolkits for IDEs. Examples include Jazz [21] and Palantír [98]. Unfortunately, these collaborative extensions are still based on conventional file sharing technology such as source code repositories. While they may provide useful information to collaborating users, the underlying IDEs remain predominantly focused on single-users.

Providing adequate tool support for CSE is a hard problem. After years of research, as outlined in Chapter 2, it appears that there is no quick fix in improving the levels of collaboration support in conventional SE tools.

www.manaraa.com

In order to truly progress, it is likely that the complex and implicit relationships in the software being developed need to be modelled, as discussed in the previous chapter. It may also be necessary to analyse the locations of users within the software project in order to detect areas of potential conflict and overlapping duties. Single user IDEs adequately expose relationships between different regions of code, but comprehensive CSE tools need to incorporate the dimension of multiple users as well.

Extensibility is a key property of any SE tool, collaborative or conventional. In terms of CSE tools, support for extensibility can not be overlooked; a tool that can not be customised or evolved is unlikely to gain widespread acceptance. Similarly, if CSE tools are developed for one specific process, their usage may be unnecessarily limited. Ideally, CSE tools should be adaptable for new SE processes as they come into mainstream development practice.

Any functional and usable CSE tool is likely to be complex in design, as discussed in Section 4.2.4. Similarly, as discussed in Section 4.2.2, it is considerably high-risk to construct one large monolithic system where every SE task is supported. From Section 4.2.1, it is also apparent that many aspects of tool design are considered during CSE tool construction. In Section 4.1.2, several design approaches were discussed that often fail to produce extensible, scalable and general-purpose CSE tools. In this chapter a framework-based approach for CSE tool design and support is presented that allows such tools to be developed.

The decision made within this thesis, based on the aspects identified in Chapter 4, is to provide a framework with the potential to support nearly all types of CSE tools, tasks and people. The framework, based on a shared semantic model of software and the propagation of atomic-level events, is an approach entirely different to the construction of task-specific tools or the augmentation of conventional tools with CSCW toolkits. A framework based approach, if implemented correctly, has the ability to support the rapid construction of a virtually endless number of quality CSE tools that can operate together in real time.

المنارة للاستشارات
www.manaraa.com

5.1.2 Framework-Based Tool Support

The development of the Caise framework was a major undertaking in terms of design and development, but the premise was that such a framework would be valuable to the progress of CSE if implemented correctly and thoroughly.

A key objective of the research in this thesis is to find means to provide proactive information to users as they develop software collaboratively. This is opposed to the conventional approach of reaction-based problem resolution stemming from failed repository commits and project build errors.

The initial approach for supporting proactive CSE was to produce collaborative tools through conventional means, such as augmenting standard SE tools with Groupware capabilities. I discovered that this approach was not feasible due to the limited support that CSCW has for highly-structured documents such as source code, the lack of support for multiple views of artifacts, and the inability for CSCW technologies to identify relationships between different units of code.

Other methods for CSE tool support, as outlined in Section 4.1.2, have also met limited success. CSCW approaches to CSE increase the communication bandwidth, but are not scalable or necessarily appropriate in all development scenarios. Conversely, using CVS and single-user tools appears initially scalable, but communication is crippled and code integration can be highly problematic. Software engineers require a design that provides the best of both worlds—a high communication bandwidth and structured control over software artifacts.

For the work in this thesis, a totally different means to facilitate CSE tool construction has been taken, in the form of a fully-synchronous approach. The general schematic view of such a framework, Caise, is presented in Figure 5.1. The key concepts of the framework is a shared set of artifacts that individual tools can edit in real time, and a server that coordinates the actions of each user and tool. By way of a framework, different types of CSE tools can operate together on a common project in real time. The architectural details of the Caise framework are presented in Section 5.2.

To address the difficulties in constructing genuinely useful and usable CSE tools, the Caise framework was designed to support a range of different

www.manaraa.com

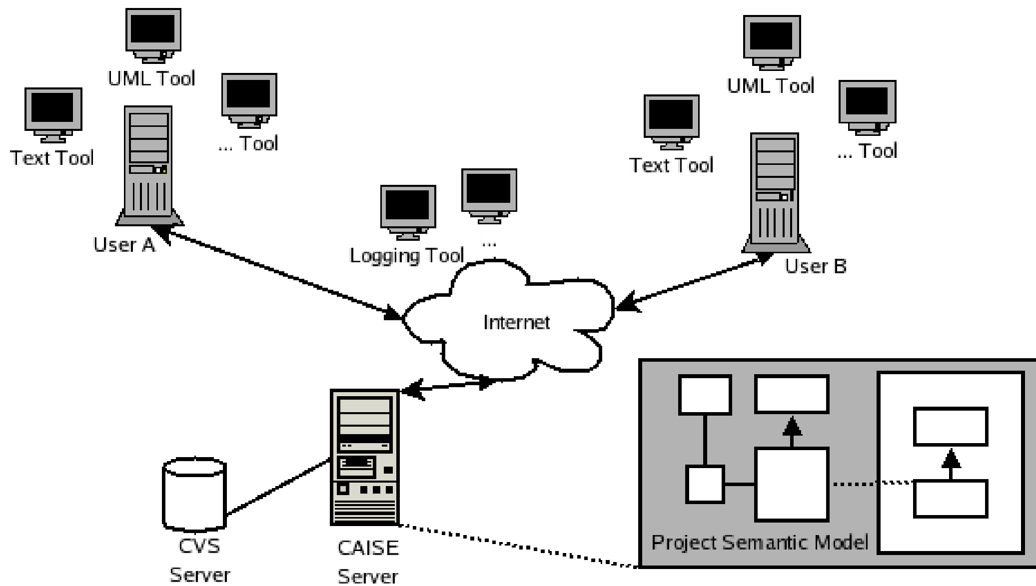


Figure 5.1: A general schematic representation of the CAISE framework.

types of CSE tools. Given a framework that provides common collaborative services through a central server, the premise is that it should be possible to rapidly construct tools of many different types. This premise is demonstrated in Chapter 6, where simple construction of several different types of CSE tools from within the CAISE framework is demonstrated.

A framework approach was favoured as it allows the core CAISE architecture to be relatively simple; an over-architected, heavyweight CSE framework may be unworkable and present too much of a learning curve for developers of practical CSE tools. By default, the CAISE framework does very little; it just supports generic sharing of artifacts, a basic event model, interprocess communication, and facilities for incorporating user-defined operations. Developers have the duty of providing specific tools, language support, and analysis routines through a plug-ins facility. Extensions that could be incorporated also include external components such as document libraries and bug tracking systems.

It is possible to support a range of new collaborative services through the CAISE framework. These include real time editing of artifacts, shared semantic modelling of the software project, fine-grained locking of the se-

المنارة للاستشارات
www.manaraa.com

mantic model rather than file-based version control, recording of the full development activity of the project with subsequent visualisations, real time user awareness information and feedback, multiple language support, multiple views of artifacts, no limitations on the types of applications that can work together on the same project, allowance for any number of participating users, and dynamic metrics gathering.

The implementation benefits for CSE tools using framework-based support became apparent as soon as the CAISE framework was operational. By having a central server controlling the activity of individual tools, artifact modification requests can be serialised in a stable order, which makes it possible and straight-forward to support real time editing, including facilities for the challenging problem of collaborative undo [119, 90, 108]. Additionally, with the majority of the functionality implemented within the server, client tools are relatively simple and light-weight.

It is possible to implement comprehensive CSE tools in ways other than using a collaborative framework such as CAISE. As described in the remainder of this chapter, however, the CAISE approach works well both theoretically and in practice. The CAISE approach of a shared semantic model, propagating atomic events, a central server, and a protocol for tool interaction can be used as a blueprint for other collaborative frameworks.

5.2 Overview of the CAISE Framework

The general concept of the CAISE framework is presented in Figure 5.2. Tools can join a CAISE-based project and begin editing artifacts using the CAISE tool protocol, as presented in Section 6.2.4. An API is available to access the underlying semantic model and project change history, as presented in Section 6.2.2. CSE tools can be developed rapidly, as long as the CAISE tool protocol is adhered to and the semantic model is used as the authoritative source of all project information.

The design philosophy of CAISE and its associated tools is the favouring of continual communication and conflict resolution over working in private with delayed identification of coding problems. Within the CAISE framework, development in private on a separate copy of source files followed by a

المنارة للاستشارات
www.manaraa.com

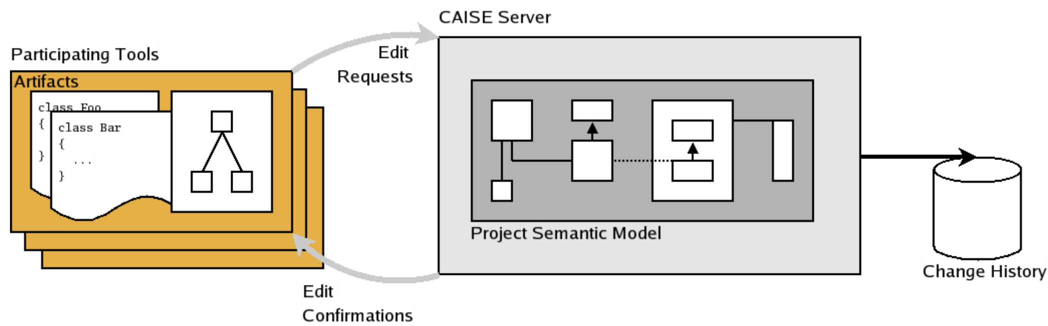


Figure 5.2: Artifact modification within the CAISE framework. Internal to the framework is a constantly-updated semantic model, which represents the authoritative structure of the software project, and is used to provide accurate, fine-grained feedback information to participating tools.

subsequent merging process is not supported explicitly. Rather, all changes to a project's artifacts are propagated to the entire software development team as they happen.

A key aspect of the CAISE framework is the ability to generate detailed and accurate information related to user activity, impact of changes, and relationships between users. As all actions of every user are observed by the central server, and all artifact modifications are semantically analysed, it is possible to explore the development history of a software project down to the finest level of detail.

The ability to share artifacts in real time, introduce new types of artifacts, map between alternate views of artifacts, and introduce new types of feedback allows the types of tools discussed in Section 4.2.1 to be supported. Additionally, given the rich semantic model of software housed within each CAISE-based project, any type of feedback information listed in Section 4.4 can be generated.

5.2.1 Architecture

CAISE is not a specific tool or IDE. CAISE provides CSE services and a semantic model of software, allowing tools to collaborate in real time. Tool developers can use CAISE in any way they see fit, and extend the framework if desired, such as adding new types of feedback information for CSE tools.

المنارة للاستشارات
www.manaraa.com

The Caise framework is simply one implementation of a system that incorporates the functional aspects listed in Section 4.2.1. Instead of building specific tools to match a given list of requirements, a framework has been constructed that provides essential services for CSE tools, allowing any type of CSE tool to be accommodated.

During initial research into CSE tools for this thesis, I realised that a large amount of processing is required in order to analyse source code as it evolves in real time. This was the main factor governing the decision to implement a collaborative framework with a central server. Therefore, the Caise server is essentially a shared IDE engine, where each Caise-based tool is a client. The only special requirements are a low latency network connection, such as a switched Ethernet LAN, and relatively powerful hardware to host the Caise server. The resource requirements for the Caise framework are discussed further in Section 7.4.

Caise is a large system, and was designed to meet a well defined set of requirements. It is extensible, customizable, and highly versatile. To demonstrate the completeness of the framework, a number of different types of CSE tools are presented in Section 6.3. Support for multiple languages is discussed in Section A.2. It should be noted that while the Caise framework can support any number of different languages, individual projects will typically be based upon a single language.

A Code-Centric Design

The Caise framework is designed as a code-centric system; the semantic model at the core of the framework, to be described in Section 5.6, represents the structure of a software project that is typically derived from the software project's source files. While corresponding alternative views of source code, such as UML class diagrams, can also be based directly from a semantic model of software, this does not imply that all types of SE artifacts can be supported natively. For example, a UML diagramming tool can obtain most of the information that it requires for a component diagram [38] from a semantic model, such as class and package names and associations between packages. However, higher level component diagram concepts such the key

المنارة للاستشارات
www.manaraa.com

components and connections of the system can not be automatically derived as they are not explicitly or implicitly represented within the semantic model.

The code-centric approach of Caise is well suited to a framework which supports CSE development tools; many popular tools in use today are code-centric, such as Eclipse and Visual Studio, and are often used in code-only modes. The majority of CSE tools are envisaged to be at the implementation, testing and maintenance stages of the SE lifecycle, and will subsequently be based upon the direct manipulation of source code and semantic model-based diagrams. A code-centric approach allows CSE tools to interact with other tools and the underlying framework without any need for code annotations or complex messaging interfaces. Additionally, the semantic model can be used as the canonical source of information, ensuring consistency between tools.

By having a code-centric framework, code-centric tools such as text editors, debuggers, and class diagramming tools are the easiest to support. Tools such as state and interaction/sequence diagrammers are also well supported, but will require some additional information, such as layout data, to be supplied by external means. The most complex types of SE tools to support within the Caise framework are those related to workflow, such as use case diagrammers, as concepts such as process flows, actors and customers are never modelled within the core software structure. In these cases, the Caise framework can be extended by introducing new types of tool artifacts, as discussed in Section A.3.3, by extending the semantic model beyond the source code level, to be discussed in Section 5.3.1, or even by using a different semantic model developed specifically for this class of tool.

Software Engineering Methodologies

The Caise infrastructure does not impose a specific methodology onto Caise-based tools; rather tool developers can implement particular methodologies on top of CSE tools if and when required. A key design decision was to avoid enforcing any particular programming paradigm—the Caise framework's key objective is to support generic collaborative software development. Processes such as RUP or XP can potentially be enforced by policies within

المنارة للاستشارات
www.manaraa.com

the CAISE server, and designers of CAISE-based tools are free to implement any process-specific mechanisms within their tool set.

CAISE is ideal for supporting distributed pair-programming. Using CAISE this practice can be referred to as *N-programming*, as there is no theoretical limit to the number of people and types of tools that can collaborate at any point in time. This is a significant advantage over conventional pair-programming; up until now collaborative technology limitations have restricted programmers considerably [92].

Degree of Collaboration

It is difficult to provide a fully synchronous service for the editing of source files and other SE artifacts. The few tools that do support collaborative editing, such as Borland's JBuilder [12], work on very restricted floor control policies such as token passing. The aim of the CAISE framework is to support any number of collaborating users in real time.

To illustrate the degree of collaboration offered by CAISE, CSE tools relative to the collaborative spectrum are presented in Figure 5.3. As indicated in this figure, CAISE-based tools are afforded some variation in the amount of collaboration they support. Most CSE tools, for example, are likely to support full collaborative editing of artifacts, but some tools might choose to propagate only significant events such as completed method bodies.

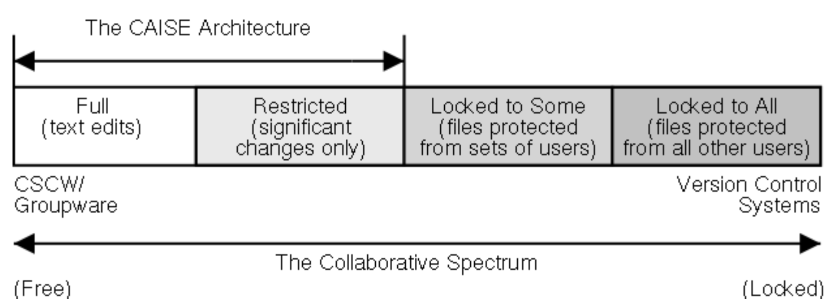


Figure 5.3: The CAISE framework in the context of the collaborative spectrum.

As also indicated in Figure 5.3, CAISE-based tools are not designed to support conventional modes of SE such as optimistic or pessimistic file lock-

www.manaraa.com

ing. Conventional SE is based upon the copy, modify and merge idiom of code repository systems, but with the availability of fully synchronous artifact editing, CAISE tools typically operate on central, shared artifacts, with social protocols to facilitate mediation between developers.

5.3 *Architectural Design*

An architectural overview of the CAISE framework, including participating CSE tools, is presented in Figure 5.4. This figure demonstrates CAISE-based tools that update artifacts (1), which are turn delivered to the CAISE server. The server analyses the artifacts (2) and updates the underlying semantic model (3). Updated artifacts are returned to each CSE tool (4a), and dynamic feedback such as user proximity information is also returned in an event-based manner (4b). Upon receipt of updated artifacts, tools adjust their local views of the project (5).

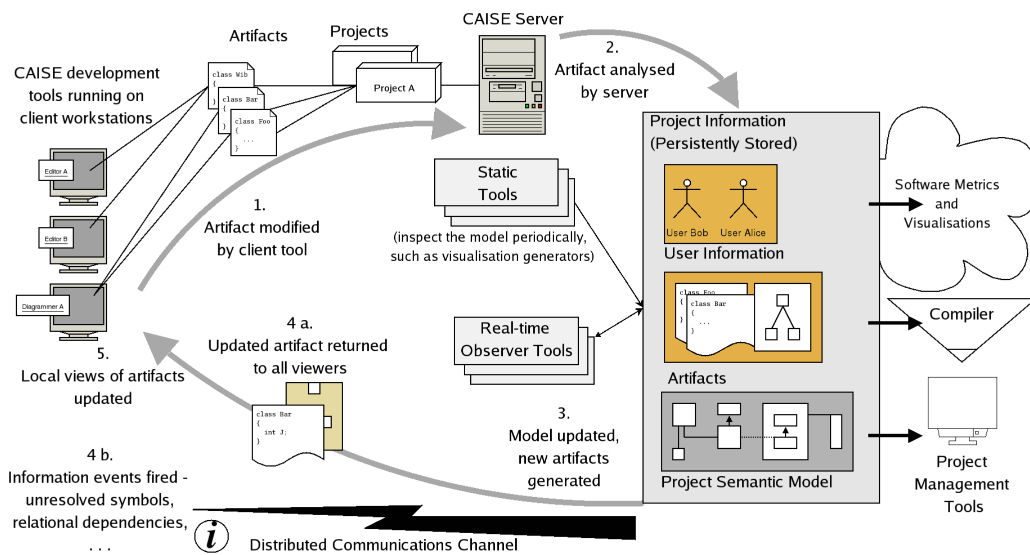


Figure 5.4: An illustration of the CAISE framework and participating tools.

To explain the relationships between artifacts, tools, the semantic model and feedback plug-ins, Figure 5.5 illustrates the key cardinalities between components within the CAISE framework. Users within a project can operate

www.manaraa.com

many different types of tools at the same time. A tool typically operates on one artifact type at a time, although multiple artifacts of the same type may be accommodated. An IDE might incorporate several tools in one application, but this is a function independent of the CAISE framework. Feedback plug-ins are specific to a semantic model, and may produce general feedback information or be tool-specific depending on the implementation. Each project has a single instance of a semantic model, but the CAISE server can support more than one type of semantic model.

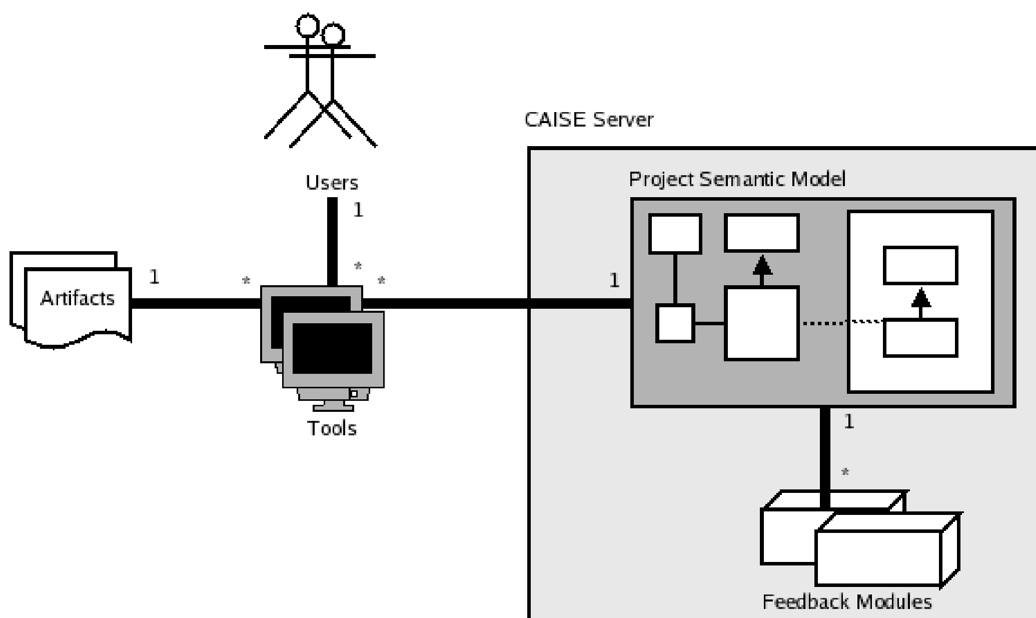


Figure 5.5: Relationships between key components of the CAISE framework.

The CAISE framework itself makes no assumptions about the types of tools, semantic models and feedback plug-ins. Its role is to coordinate the components of the framework based on well known interfaces, to support communication between tools, to facilitate the propagation of events between tools and the server, and to provide storage and shared access to CAISE-based artifacts. Analysers and feedback plug-ins support tool and language-specific operations.

المنارة للاستشارات
www.manaraa.com

5.3.1 The Project Semantic Model

The need for full semantic modelling of a software project's structure was discussed in Section 4.3. Advantages of semantic modelling include being able to rapidly refactor units of code, accurately query the semantic model in real time, and determine relationships between users and units of code. A semantic model for each CAISE-based project is stored within the CAISE server. The concept of using a shared semantic model to facilitated collaboration between SE tools is new to the field of CSE, and is a key research achievement within this thesis.

A semantic model represents the entire program within a CAISE-based project. As discussed in Section 4.3, a semantic model represents all software entities such as packages, classes and methods, and the relationships between them such as method invocations and code dependencies.

A key feature of the design of the CAISE server is the decoupling of languages and the semantic model of software. While source files and CAISE-based tools might be of a specific language, the semantic model is language-independent. This means that tools which inspect the semantic model, such as the feedback plug-ins presented in Section 6.2.5, can be written independent of specific languages, increasing their amount of use within the CAISE framework.

As the research for this thesis is based primarily for the support of Java and Java-like languages, the main semantic model used within CAISE at present is OO based. This semantic model is similar to that of Microsoft's .Net framework [20], where multiple languages can also be encompassed. For languages that are fundamentally different from the OO paradigm, another type of semantic model can be introduced into the CAISE framework, or the existing semantic model can be expanded.

The current semantic model of OO software offered by CAISE provides full support for Java 1.4. Work is near completion [79] for a .Net 2.0 version, including Generic Types. The architectural diagram for the Java 1.4 semantic model of OO software is presented in Figure 5.6, as reproduced from [60].

The semantic model presented in Figure 5.6 was taken from the research of Irwin and Churcher [60]. A fine-grained semantic model of software was

www.manaraa.com

Figure 5.6: A semantic model of object-oriented software [60], in UML notation.

المنارة للاستشارات
www.manaraa.com

required for the construction of the CAISE framework, and instead of writing one specifically, a new version of the JST [60] semantic model was formed. The JST semantic model was applicable because it accurately models Java programs based directly from the standard Java exposition grammar [47], and it has a logical API for semantic model navigation. Extensions required for use within the CAISE framework included supporting incremental updates to the semantic model, extending the explicitly mapped relationships within the semantic model, and calculating a list of semantic changes based upon semantic model updates.

The semantic model of OO software within the CAISE framework is designed for inspection, incremental updating, and user querying. This is in contrast to the semantic models within Borland's Together Architect and the Eclipse IDE; these semantic models provide limited programmatic access and have little documentation. Additionally, the semantic model within CAISE supports direct modification of the entire semantic model.

Example routines accessible by the API for the CAISE semantic model of software include: `lookupType()`, `get/addPackage()`, `get/addDeclaration()`, `get/addSourceFiles()`, `get/addType()` and `get/addMethod()`. These routines are as accurate as any compiler, and are callable from any participating CAISE-based tool.

The CAISE semantic model of software models Java source code down to the statement level. The only low-level constructs not explicitly modelled are control statements such as `if` statements, `for` loops, and `switch` statements. This design choice was made by the author of the original semantic model, and the inability to model these low-level concepts does not affect the CAISE framework in any significant way. Declarations and uses of all types are still modelled, even down to the local variable level; for example, an `if` statement is not directly modelled, but any uses of variables—including declaration and assignment within the statement—are represented.

A JavaDoc listing of the API for the semantic model of software is available from Appendix H. CAISE-based CSE tools (presented in Chapter 6.3) and server applications (presented in Appendix A.4) can access the semantic model directly, but other tools may choose to download a snapshot of the semantic model via the CAISE tool API, and inspect the semantic model offline.

المنارة للاستشارات
www.manaraa.com

The CAISE event log, as presented in Section A.5 may also be downloaded for user activity and change history information.

It is expected that the current semantic model within CAISE can be used by tool developers for most languages in common use today. The semantic model can be extended as required to encompass other concepts, however, such as low-level control statements or abstract components such as UML actors and state transitions. If a CAISE-based project requires a different type of semantic model, most likely due to an unconventional language or the use of CAISE for a completely different domain such as web site development, any existing tools and feedback plug-ins that are required for use must be updated according to the new semantic model's properties and structure.

Semantic Analysers

Semantic analysers within the CAISE framework are responsible for building and maintaining the semantic model for each software project. The primary task of CAISE-compliant analysers is to inspect parse trees and insert any identified declarations into the project semantic model.

At present, only one type of semantic model exists for the CAISE framework. This is the general semantic model of OO software, as presented in Section 5.3.1.

Two languages are currently supported in CAISE, namely Java and Decaf. Therefore, a corresponding semantic analyser for each language exists as a CAISE-based plug-in. As each language can be mapped to the general semantic model of OO software, both analysers populate an instance of this general semantic model.

Much work has to be done by the analyser in order to build a semantic model of software. Most of this work, however, can be performed in a language independent manner. Therefore, language-specific analysers are not solely responsible for constructing a semantic model of software. The CAISE general semantic model not only contains the declarations and relationships within software project, but also the routines to look up types and construct much of the semantic model itself.

Work performed by the general semantic model of software includes re-

www.manaraa.com

moving components declared in the semantic model from previous versions of parse trees, cross referencing the semantic model upon the introduction of new versions of parse trees, and calculating semantic changes in the semantic model.

The semantic model's methods for rebuilding itself are called every time a new parse tree appears from a tool, or a tool issues a semantic model change command. The general semantic model employs its associated analyser through a Strategy Method [44] when updating itself. Various interface methods of the analyser will be called, such as `addParseTree()` and `lookupType()`.

Designers of new semantic analysers can rely on the CAISE framework to call the prescribed analyser methods in the correct order as required, which reduces the complexity usually associated with constructing a semantic model of software from evolving source files. All duties associated with incrementally updating the semantic model, such as removing previous semantic model declarations and cross referencing new declarations, are performed by the semantic model itself. The main task of CAISE-compliant analysers is just to insert new declarations into the semantic model. A related duty for analysers is to override the methods for looking up types where the language-specific scope rules differ from the assumptions made in the general semantic model of software.

It is not difficult to map different types of programming languages into CAISE's general semantic model of software. For example, the C# language could be supported within CAISE by creating a new semantic analyser as a CAISE plug-in. Such an analyser would map C# parse trees into the existing semantic model of software, presumably with minor modifications, if any, to the general semantic model's current structure. The role of multiple language support within the CAISE framework is discussed further in Section A.2.

A secondary task of CAISE-based analysers is to convert segments of the semantic model back into parse trees for distribution to CAISE-based tools. This situation arises when the semantic model is directly edited by tools such as class diagrammers. The CAISE framework responds to this request by asking the appropriate CAISE analyser to modify a copy of the relevant source file's parse tree, reflecting the modification request. Once this parse

المنارة للاستشارات
www.manaraa.com

tree has been updated, it is incorporated back into the semantic model in the same manner as genuinely updated parse trees are incorporated from modified source files.

A full discussion of semantic analysis of OO languages is beyond the scope of this thesis, but is presented elsewhere [58]. For the purposes of the CAISE framework, a CAISE-compliant semantic analyser must have the ability to construct a semantic model given a set of parse trees, and conform to the CAISE AnalyserPlugin interface, as presented in Appendix D.

A Multiple-Layer Architecture

The CAISE framework can be accurately described in terms of three-layers, as presented in Figure 5.7. These layers are: collaboration within each type of tool, collaboration between each type of tool, and core SE functions. The three layers can also be described as CSCW, CSE and SE.

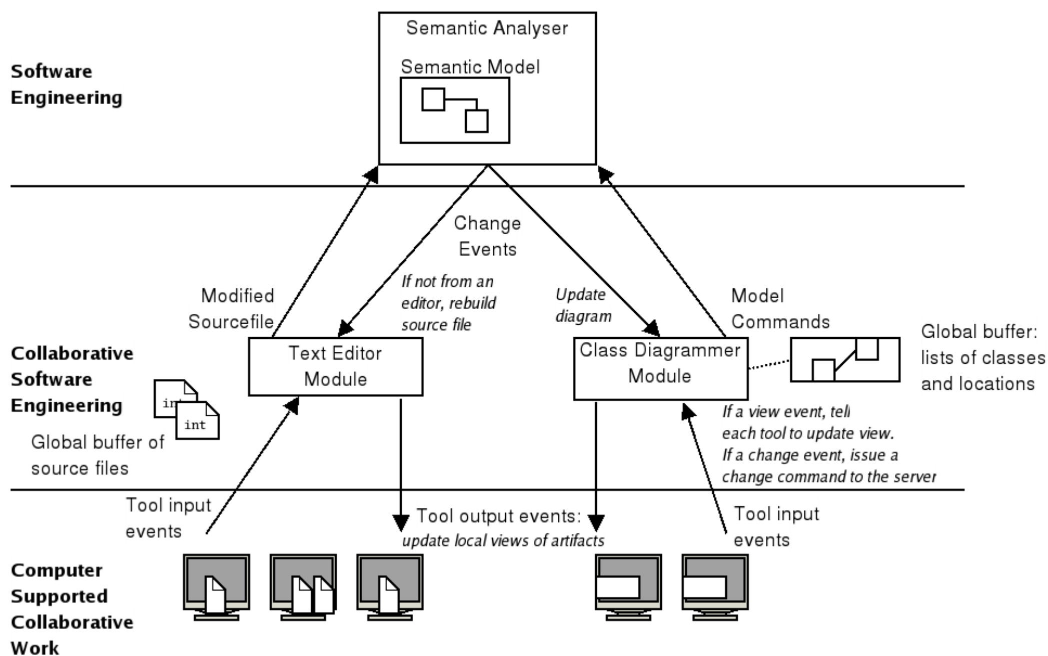


Figure 5.7: The three conceptual layers of the CAISE framework.

The CAISE framework was designed as an architecture of three layers to reduce the complexity of the CAISE server. If the CAISE server had com-

المنارة للاستشارات
www.manaraa.com

plete responsibility for interactions between all types and instances of tools within a project, the design and implementation of the server would be very complicated. Additionally, CAISE-based tools would also be very difficult to design as they would not be decoupled from the functions of other types of tools.

At the CSCW layer of the CAISE framework, as presented in Figure 5.7, syntactical events specific to each tool type are contained within the boundaries of that tool. In other words, this layer is responsible for keeping all instances of each type of tool fully synchronised. For example, the per-character events generated from modification of a source file are propagated to all other text editors within the framework, but not to other tools directly. Similarly, if a class within a diagramming tool is moved to a new location within a diagram, only tools sharing that specific view of the diagram are notified. The CAISE server is aware of all such changes internally, but these low-level, or CSCW-based events are only propagated to the relevant tools within the project.

At the CSE layer of the CAISE framework, the framework addresses events related to semantic changes within the software project; this is a unique feature of the CAISE approach. For example, if a code editor tool receives keystrokes that eventually form a new method or class declaration, this semantic event is propagated to all other types of tools. This allows class diagrammers and other tools to take note of the semantic change and update their own views of the project accordingly.

At the SE layer of the CAISE framework, the framework maintains the software product that the CAISE-based tools are working on. By decoupling the CSE and CSCW functions in other layers, the SE layer can focus on code analysis, generation of executables, and incremental integration of updated source files.

Representing Multiple Views

Round-trip engineering, where source code and diagrams are treated as equivalent representations of the same underlying semantic model, is a cornerstone feature of most modern IDEs. Supporting multiple views of software is rel-

المنارة للاستشارات
www.manaraa.com

atively straight-forward for non-extensible single-user tools. Multiple view support in Borland's Together IDE is discussed by Garrett [46].

For extensible collaborative tools, multiple views are difficult to support. The arbitrary introduction of additional types of artifacts and new languages adds great complexity to existing collaborative round-trip engineering facilities. A low-overhead mechanism must be available to allow tools to stay up-to-date with any new types of artifacts within the project. Network traffic volumes when transferring artifact information must also remain low to ensure tool response times are kept to an acceptable rate.

A key challenge in supporting many different artifact types within CSE tools is finding a way of keeping all currently supported views synchronised between multiple tools as artifacts are edited. For example, if a text editor tool is used to declare a new method, this new method should appear in the view of all class diagrammers, and vice-versa. There are two main ways of achieving synchronisation between different types of tools: using an explicit mapping mechanism from one view to all others, which arguably does not scale well, or having in place a semantic model that is rich enough to represent all views.

Within CAISE, the general semantic model of OO software is expressive enough to present views of artifacts as source files or diagrams; this is evident from the tool demonstrations presented in the accompanying resources disc. Typically, upon syntactically-correct modifications, updated parse trees and code buffers are distributed to tools, which allows them to update their own local views of artifacts, as illustrated in Figure 5.10. For example, if a class diagramming tool adds a new method to an existing class, the resultant updated source file that contains this class will be sent to all text editors for updating of their own views.

5.3.2 The CAISE *Event Model*

The CAISE framework is relies upon simple and frequent events from participating CSE tools. CAISE-based tools report to the server every action taken, such as a cursor location change or a keystroke, and the server updates the underlying model accordingly. This implies that the server has much fre-

المنارة للاستشارات
www.manaraa.com

quent processing to do, but each task is relatively small. By working with very fine-grained events, the CAISE framework can also provide synchronous collaboration, such as real time shared text editing and immediate analysis of code changes.

As well as input events from tools, the CAISE server generates output events which are reported back to tools and other interested applications. Output events include notification of recent artifact modifications, and feedback events such as metrics, impact reports and code neighbourhood information.

The event model for the CAISE framework is presented in Figure 5.8. Within the CAISE event model, the CAISE server broadcasts events of various types to all participating tools that are registered as event listeners. Tools can register as listeners for all events or just specific event categories. Events contain details of the general action, such as an artifact edited by a named user, and the specific details, such as the affected text and file offset. Each event generated within a CAISE-based project is also recorded in the CAISE event log, which is described further in Section A.5.

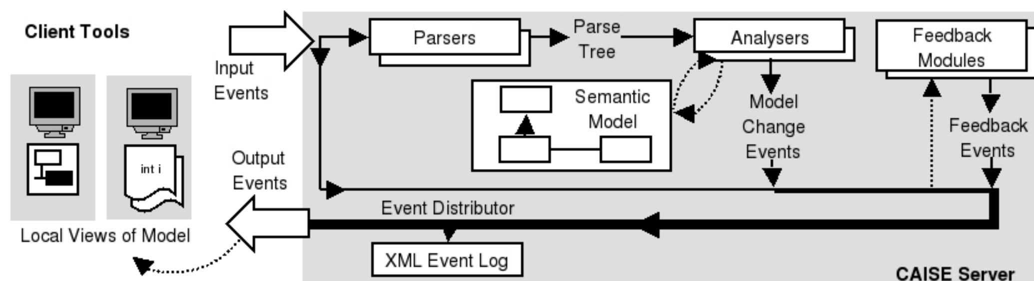


Figure 5.8: The CAISE event model.

Each CAISE event type is briefly summarised in Table 5.1. Fully featured tools are likely to register as listeners for all events, as can be seen in the code listing in Section 6.2.6. Other components, such as the Change Graph presented in Section 6.2.3, are only interested in specific event types.

The basic structure of CAISE events is as follows. Each event records the user responsible for generating the event, the users that received the event, the type of event, the time that it was generated, a reference to any semantic

المنارة للاستشارات
www.manaraa.com

Type	Typical actions
Project	A project is created or deleted.
Artifact	An artifact is added, removed or edited.
Chat	A user issues text or audio messages.
Feedback	Tool-specific custom units of information exchange.
Client	A client opens, closes or moves location within an artifact, or rebuilds a project.
Change	The project's semantic model is manipulated directly or via artifact modification.

Table 5.1: Event types within the CAISE framework.

model components directly related to the event, any other event-specific data. The full definition of the CAISE event structure is given in a JavaDoc API listing available from Appendix H.

The CAISE event log is discussed further in Section 7.2, where tools to analyse and visualise user activity are presented. The XML Data Type Definition (DTD) for the CAISE event log is given in Appendix C.

5.3.3 Artifact Modification

In terms of file sharing, the CAISE framework is based upon the pattern of Atomic Integration, as presented in Section 3.4.2. Each CAISE-based tool works on a shared set of artifacts that are stored on the central CAISE server. Artifacts are shared and modified in real time, implying that changes are replicated to all tools within the system as they happen.

In Figure 5.9, the key types of modifications made to artifacts are illustrated. These include changes in user locations (1), changes to the syntax of the artifact (2), and semantic changes that result from syntactical changes (3). This figure also illustrates the event-based nature of the CAISE framework—a tool generates an input event, the CAISE server responds by updating the appropriate artifacts, and finally the CAISE-based tools update their local views based on feedback from the CAISE server.

When a CAISE-based tool determines that a modification to an artifact has taken place, such as a source file undergoing an edit, the tool will notify the CAISE server of the modification. This is a stipulation of the CAISE tool

المنارة للاستشارات
www.manaraa.com

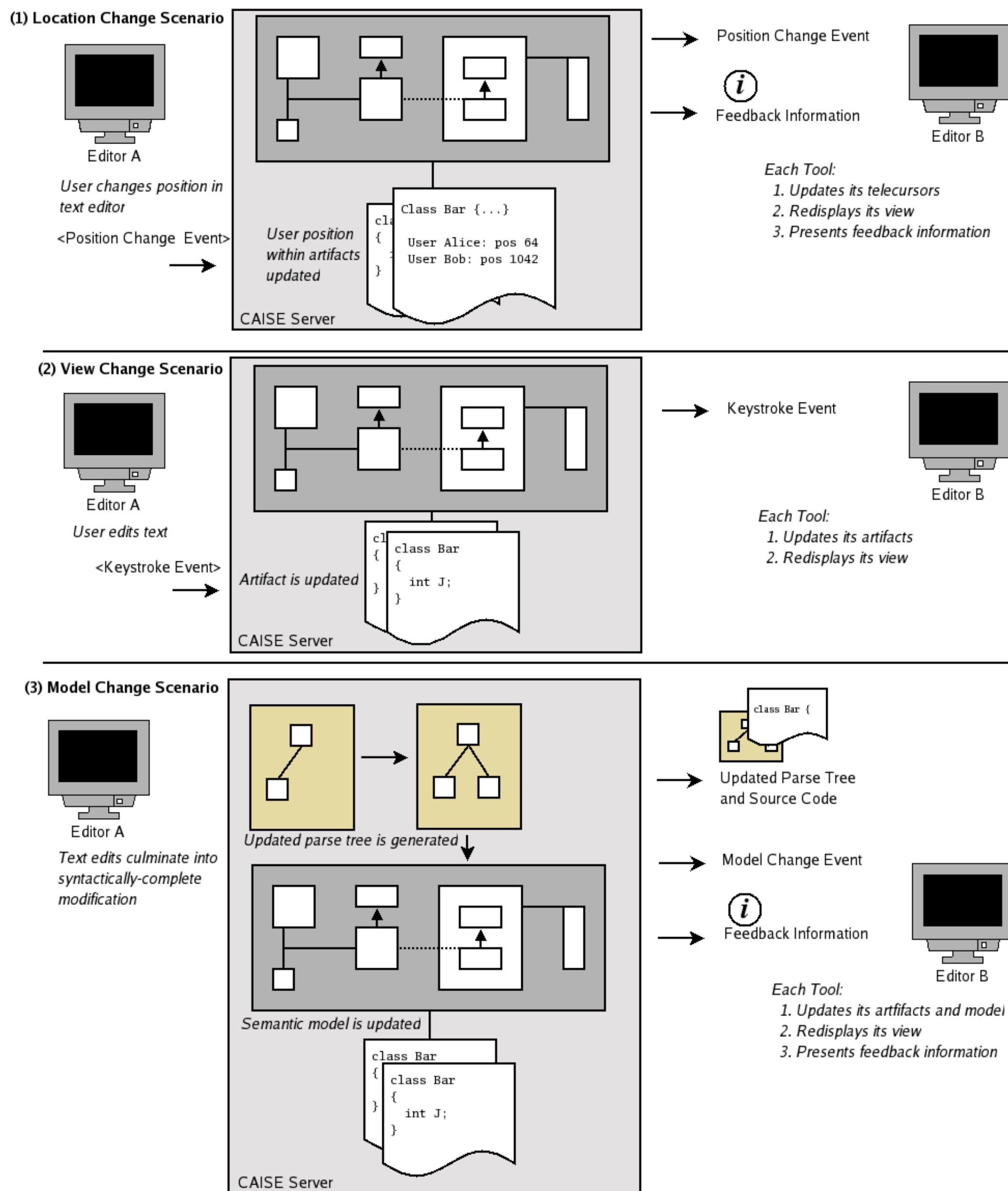


Figure 5.9: Key types of actions within the CAISE framework.

المنارة للاستشارات
www.manaraa.com

protocol, as discussed in Section 6.2.4. In the case of a fully-synchronous tool, such as a text editor, an edit will be an atomic event—typically a single keystroke.

When a tool has determined that the file might be in a syntactically correct state, the CAISE server is called upon to parse the file. Upon successful parsing, changes are semantically analysed, and parse trees are distributed to all participating tools that require them, allowing them to update their own local views of the project, as presented in Figure 5.10. If the project's semantic model was altered as a result of the modification, this information is also collated for distribution as feedback events for any interested tools. If a file fails to parse, this is noted in the CAISE project event log, and an information event is broadcast for any tool that may be gathering project activity metrics.

In most circumstances, only syntactically correct modifications are propagated to other types of tools. For example, while a new property is being typed in from a code editor, this partial and syntactically incorrect declaration is not transmitted to other tools such as class diagrammers. This approach does raise a transactional issue: if a line of code is currently being typed in by one user via a text editing tool, and a second user makes a change to the corresponding entity within a diagramming tool at exactly the same time, the CAISE server does not currently integrate the uncommitted changes of the text editor with the updated parse tree, resulting in a loss of any uncommitted text editor changes.

For rare situations where social protocols do not provide adequate protection against conflicting modifications between shared artifacts, the issue of text modification loss can be easily addressed. This is achieved by adding a mechanism within the CAISE server that simply merges any uncommitted changes within an existing source file into the newly formed parse tree and updated source code buffer. Alternatively, individual text editing tools can easily implement a merging mechanism which performs the same process on the client side.

Tool designers may also choose to ignore changes to artifacts made by other users in order to provide a degree of isolation for the programmer, but doing so runs the risk of having source files that are no longer synchronised

المنارة للاستشارات
www.manaraa.com

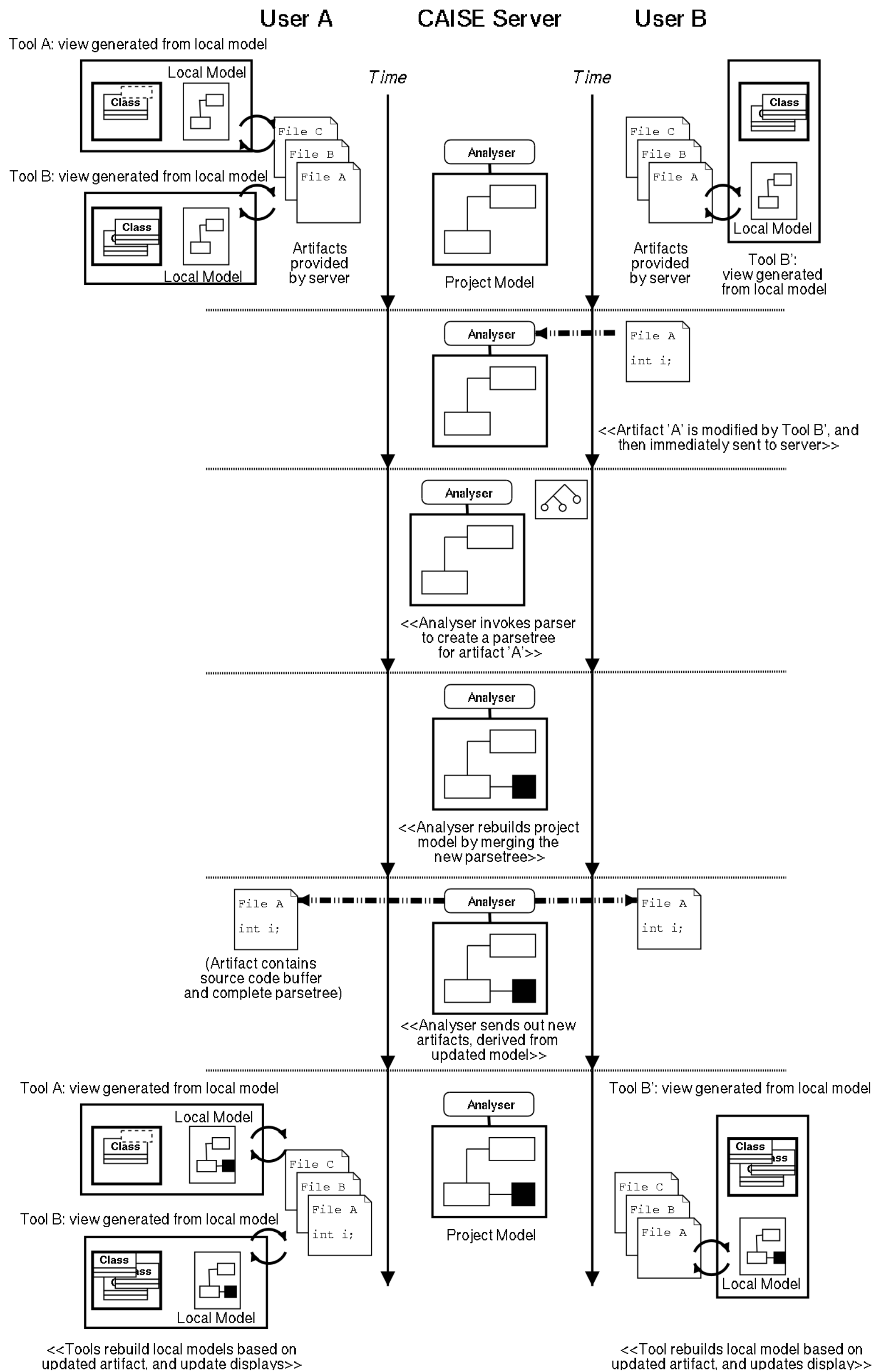


Figure 5.10: Schematic view of an artifact modification within CAISE.

المنارة للاستشارات
www.manaraa.com

with the authoritative CAISE server. It is, however, trivial to detect files that are not synchronised, and in such cases, a file can be brought back up to date easily via the CAISE tool API which is presented in Section 6.2.2.

5.3.4 The CAISE *Server*

To support the development of CSE tools, an underlying framework should provide core SE functions, as well as interprocess communication facilities and Groupware components. Within the CAISE framework, these facilities are all supported by the CAISE server.

The CAISE server is responsible for the storage and collaborative management of all artifacts within a CAISE-based project. The CAISE server also manages the semantic model of software, project event log and user information. Other functions of the CAISE server include relaying different types of events to appropriate listeners such as development tools, project management tools and visualisation generators. The CAISE server also defines mechanisms to support extensibility, such as introducing new languages, artifacts and types of feedback to the framework.

To support genuinely useful CAISE-based CSE tools, the underlying framework must be of a high quality. This implies that the CAISE server must be able to seamlessly handle multiple concurrent requests at any time, provide undo support within a collaborative setting, have acceptable response times even under heavy system load, and be practical to extend.

Implementation details for the CAISE server are described in further detail in Appendix A. Support for programming languages is presented in Section A.2, including mechanisms for parsing (Section A.2.1). Storage of CAISE-based artifacts is discussed in Section A.3, including support for collaborative undo (Section A.3.2). Adding additional server functions to the framework is presented in Section A.4. The CAISE event log is presented in Section A.5. Project administration is discussed in Section A.6. The CAISE plug-ins architecture is described in Section A.7. Interprocess communication between the CAISE server and participating tools is presented in Section A.8.

المنارة للاستشارات
www.manaraa.com

Communication Protocols

To provide a reliable means of distributed communications and synchronisation between participating CAISE-based applications, a central communications server is employed. This communications server is housed as part of the overall CAISE server. Many options are available to provide distributed communication, group management facilities and floor control policies, including toolkits such as the Java Shared Data Toolkit (JSDT) [18] and GroupKit [95]. I decided, however, to support communication through the `caise.messaging` framework, as presented in Appendix A.8. This custom framework provides a fast and low-overhead communications facility, essential in the construction of a practical CSE framework.

Floor Control Policies

Within the current version of CAISE, the server supports an unrestricted floor control policy, where social protocols and user awareness support are relied upon to prevent conflicting actions between participating CAISE-based tools. It is possible, therefore, with the current transaction control scheme to have situations where one user's change can be immediately negated by a subsequent or competing request of another user. Awareness mechanisms are in place, however, to highlight fine-grained concurrent work such as this, which normally provides adequate protection to avoid such situations. In Section 7.3, an evaluation is presented where CAISE-based CSE tools provide feedback related to conflicting changes during fine-grained concurrent development.

Tool Synchronisation

With conventional systems that provide concurrent access to shared resources, such as database management systems, a transaction control system is in place to detect conflicts between batches of modification requests. Such systems also provide mechanisms for rolling back conflicting modifications, allowing the system to reach a globally-stable state.

The CAISE approach is significantly different from conventional collaborative systems. A key purpose of the CAISE approach is to investigate

المنارة للاستشارات
www.manaraa.com

how far the real time, atomic sharing of artifacts can be applied to CSE using custom tool support, awareness mechanisms and social protocols. As described in Section 5.3.3, each artifact modification request consists of a fine-grained, atomic action which is propagated immediately to all participating tools within a Caise-based project. Given that all events are atomic and are serialised, and are processed and propagated by the Caise server, formal transaction control is not needed.

Within the Caise framework, the propagation of *atomic* events is necessary. Atomic events are defined as the smallest single units of actions that modify the state of the project semantic model or associated artifacts. Examples of atomic events include keystrokes within editors or user location changes from one position to another in any Caise-based tool.

The Caise server provides a single incoming event queue that all instances of Caise-based tools add to by way of the Caise tool API, presented in Section 6.2.2. By employing a single project-wide event queue, a serialisation of events is possible. This ensures that the order of processing is kept consistent as the event makes it way through the remainder of the Caise event lifecycle, as presented in Figure 5.8. As explained in Section 5.3.2, Caise-based tools make no assumptions as to when their events will be processed in relation to competing events from other tools. They can only assume that the order of their own events will remain consistent during processing by the Caise server.

Given that all Caise-based tools adhere to the Caise tool protocol, as described in Section 6.2.4, there is no possibility of a loss of synchronisation between running instances of Caise tools. The only caveat, as discussed in Section 5.3.3, is that text editors may be exposed to a loss of any uncommitted changes if parse tree based tools are modifying the same artifact at the same time, but this can be easily remedied in a future version of the Caise framework. Deadlocks are also not possible within the Caise framework as each operation on the project's semantic model and associated artifacts are independent from all other pending modification requests.

Sending modification requests to the Caise server infrequently and periodically as a batch is not recommended within the Caise framework, as this violates the underlying Caise tool protocol. The Caise server can be

المنارة للاستشارات
www.manaraa.com

extended to support this by employing conventional transaction control facilities, but this is not within the scope of the current version of CAISE, and is not well-aligned with the principle of real time CSE.

A Centralised Server

The CAISE framework is based upon a centralised server architecture. This design choice was made from necessity; at the commencement of this research project, desktop hardware was not powerful enough to perform processing of tool update events, parsing, semantic analysis, and updating of the project's semantic model. Additionally, a decentralised architecture would have also been prohibitively expensive to develop within the scope of the research project.

Today, high-end desktop hardware is capable of running the CAISE server process. Therefore, it is possible to support a distributed version of the CAISE architecture if required, which would be well suited to open-source projects and multi-national development teams as they become increasingly distributed in nature. The most significant change to the current architecture would be adopting a distributed concurrency control algorithm to maintain synchronisation between tools, to replace the central queuing mechanism currently used within the CAISE server.

Implementation Considerations

There were many technical issues to address when constructing the CAISE server such as supporting multiple views of software, providing a means for reliable distributed communication, facilitating real time synchronous editing of artifacts, and providing plug-in support for extensibility. Even compiling some of the parser-generator based source files was challenging due to constraints of standard Java compilers, and numerous workarounds were required to allow such volumes of data to be stored within the CAISE server.

Design and implementation difficulties such as those listed above are real to CSE tool and framework developers, yet these difficulties are not normally documented within CSE literature.

المنارة للاستشارات
www.manaraa.com

5.3.5 Collaborative Tool Support

CSE tools require considerable support, as discussed in Section 4.2.1. Here a summary is presented of the many ways in which the CAISE framework supports CSE tools.

CAISE-based CSE tools can implement all the modes of development identified in Section 3.4.2, such as Follow-the-Leader and Action/Reaction. Real time sharing and modification of source files, termed atomic integration, is also possible. The CAISE framework provides at zero cost to its tools semantic modelling, code neighbourhood calculation, change impact reports and user proximity feedback.

In terms of practicality, late join-ins for CAISE-based tools are supported by the design of the CAISE tool protocol. Additionally, to be discussed in Section 8.1.3, mechanisms to avoid development activity 'jitter' from other developers during the compilation phase are provided, and CAISE-based tools can also be used with existing code repository systems if required.

In terms of the considerations for tool design discussed in Section 4.2.1, the following points are noted for CAISE's support of CSE tools:

Tool A number of CSE tools can be constructed within CAISE. There are no known technical or theoretical limitations for tool support—CAISE supports synchronous editing, multiple languages and artifacts, and framework extensibility

Task Most SE tasks can be accommodated within the CAISE framework without any modification or system extension. Task-specific duties are typically facilitated by individual tools, but the CAISE server is powerful enough to accommodate high system loads, most types of artifacts, and calculation of semantic model-derived information

People The CAISE framework is suitably generic and unbounded to accommodate any number of collaborating developers and concurrently connected tools. People can be distributed throughout a global network, and be located across different time zones. The CAISE framework can also be extended to accommodate specific developer roles if not already supported by CAISE-based CSE tools

المنارة للاستشارات
www.manaraa.com

5.3.6 Framework Extensibility

Extensibility, identified as an important requirement in Section 4.2.1, is a core to the Caise framework. Extensibility is provided as follows:

- New tools can be introduced into the framework by adhering to the Caise tool protocol, allowing them to read and modify shared artifacts, as discussed in Section 5.2. Many facilities are available for new tools such as collaborative widgets and rich information sources such as semantic models and full event logs

- Existing SE tools can also be integrated into the framework, depending on the degree of extensibility such tools provided. An example of an IDE that has been integrated with the Caise framework is provided in Section 6.3.3

- New languages can be supported by adding a new parser and semantic analyser, as discussed in Section A.2. If the current semantic model is not suitable for accommodating constructs within the new language, the semantic model can be extended or replaced within each project

- New types of artifacts can also be incorporated into the Caise framework, as presented in Section A.3. The more detailed the grammar of the artifact, the more detailed the project information will be after semantic analysis

- If a new type of feedback event for Caise-tools is required within the software project, again, this can be easily supported through feedback plug-ins, as described later in this section

- If other kinds of functionality are required such as batch processing of the semantic model, a server application can be incorporated within the Caise framework, as discussed in Section A.4

- New types of collaborative widgets can be added to the framework for use within CSE tools. This concept is discussed in Section 6.3.4

المنارة للاستشارات
www.manaraa.com

5.4 Related Work

Several research projects have similarities with the CAISE framework's approach to supporting CSE. The FIELD environment defined a message passing interface between numerous types of common SE tool, and also supported a cross-referencing database, typically populated by scanning source files for symbols such as method and variable names. The PCTE project also defined several detailed interfaces for tool to tool communication, backed by fine-grained relational database schemas as the canonical source of project information. Similarly, the SPADE-1 [6] environment approaches CSE from a process-centric viewpoint, where a range of tools interact over a communication interface, controlled by a process engine, accessing data from an underlying artifact repository.

The CAISE framework differs from previous research approaches in two key areas. Firstly, the CAISE framework is collaboration centric—it supports fine-grained real time collaboration natively, as opposed to other frameworks that support collaboration secondarily as a by-product of their design. Secondly, the CAISE framework encompasses a very detailed semantic model of software as the authoritative source of project information. Other research approaches at the very most support only basic semantic analysis of software artifacts; the CAISE framework is the only integrated environment that has deep semantic modelling of software at the core of the architecture.

Summary

The CAISE framework is more than just another tool to provide some degree of collaboration within SE. CAISE provides a new approach to the support of CSE by way of semantic modelling, accommodating new languages and tools, supporting scalability, and allowing customisation and extensibility. The CAISE semantic model, event log and artifacts are rich sources of information for SE analysis, and the framework provides a real time, event-based environment for management of collaborative software projects.

In this chapter, an overview of the architectural design of CAISE has been presented, including the key framework characteristics and design principles.

المنارة للاستشارات
www.manaraa.com

A description of how the Caise server operates in terms of language support, artifact sharing, and server extensibility has also been provided.

In Chapter 6, the construction of Caise-based tools within the Caise framework is presented, and several example Caise-based CSE tools are demonstrated. The design and implementation considerations over a range of Caise-based tools are also discussed.

المنارة للاستشارات
www.manaraa.com

Chapter VI

Using the Caise Framework

In this chapter several CSE tools are presented. These tools have been constructed using the Caise framework presented in Chapter 5, have been built to support the patterns of collaboration identified in Chapter 3, and incorporate the design characteristics discussed in Chapter 4.

An overview of Caise-based CSE tools is given in Section 6.1. In Section 6.2, the construction of Caise-based tools is illustrated and described. In Section 6.3, a number of example CSE tools are presented.

6.1 Overview of Current Caise*-Based Tools*

Since their conception, the CSE tools presented in this thesis have been constantly refined in order to provide realistic SE environments. Common to these tools are the following features:

- Round-trip engineering between all tools and artifact views
- Multi-user artifact sharing and editing capabilities with *relaxed WYSIWIS* views, including collaborative undo
- Instant messaging and an audio chat channel
- Build and run facilities, including protection from remote development jitter when attempting to compile during times of high development activity
- Event-based collaborative feedback information, such as proximity reports relating to other user locations within the project, and semantic model change impact reports as the project evolves

www.manaraa.com

Attention has been given to ensuring that awareness information is presented effectively within each tool. Accordingly, all relevant aspects of each tool's user interface have been designed to accommodate and illuminate constantly changing states.

The majority of the features built in to the CAISE-based tools presented here are provided by components made available from the CAISE client widgets library, as presented in Section 6.2.3. The remaining collaborative features have been implemented manually, but rely on the services of the CAISE tool API presented in Section 6.2.2 to implement functions such as tool synchronisation and the shared modification of artifacts.

Users of such tools are likely to employ the majority of the common tool features listed above, such as shared concurrent editing of artifacts, build and run facilities, and chat services. In some development groups, it is conceivable that only one user at a time will make modifications to the project. In other situations, the tools might only be used for shared navigation and code reviews.

The CAISE-based tools presented in this chapter appear as single user tools when only one developer is active within the current project. When additional developers join the project, the awareness mechanisms such as telecursors and project explorer panes activate, providing context-sensitive feedback on the locations of others. Developers may choose to ignore or deactivate such awareness mechanisms, but by default the CAISE-based tools operate in a manner similar to other CSCW applications, where the presence of others is a key aspect of each tool's user interface.

Apart from the standard CSCW facilities, the key differences between conventional tools and the CAISE-based CSE tools presented in this chapter are that artifact modifications are propagated immediately to all participating tools within the project, and that information is presented immediately to specific users as the server detects relationships between users and units of code.

Users can rely on the CAISE server to ensure that all changes are recorded and updated against the authoritative set of artifacts and underlying project's semantic model. As the entire project is shared in real time, users will occasionally experience concurrent modifications to a common region of code,

المنارة للاستشارات
www.manaraa.com

but from anecdotal and empirical evaluations presented in Chapter 7, considerable productivity gains over conventional tools are possible within typical development scenarios, without significant user hindrance.

While not studied as part of the research within this thesis, is it also conceivable that gains in software quality may occur, due to the raised awareness of the actions and intentions of others within the project.

6.2 CAISE-Based Tool Construction

This section details the technical design and implementation of the CAISE-based tools, for the purposes of further custom development and reuse of existing components. The operation of CAISE-based tools within the CAISE framework is also explained. Working examples of such tools are presented in Section 6.3.

6.2.1 Tool Construction Overview

CAISE-based tools engage the services of the CAISE server by way of the CAISE API, presented in Section 6.2.2. Services include the downloading of artifacts, the parsing of updated artifacts, and the querying of the semantic model for information such as related users or units of code.

To edit shared artifacts, CAISE-based tools must adhere to the CAISE *tool protocol*, presented in Section 6.2.4. This ensures that each tool stays synchronised with the CAISE server and all other participating tools, and that conflicting batches of artifact modification requests are not encountered. The distributed MVC design of the CAISE framework provides a deadlock-free synchronous replicated view of all artifacts within the project.

In addition to using the CAISE server for the management of shared artifacts, CAISE-based tools may also use CAISE collaborative widgets, as presented in Section 6.2.3. Such widgets range from simple, such as text chat panes, to complex, such as multi-user text editor components.

Tools may choose to respond to feedback events from the CAISE server such as code dependencies being resolved. The current types of feedback events supported within the CAISE framework were presented in Section 5.3.2.

www.manaraa.com

Tools may also require specialised feedback events, which are provided by custom feedback plug-ins, to be discussed in Section 6.2.5.

Typical tools within the Caise framework include source code editors and UML diagrammers. Various examples of Caise-based tools are presented in Section 6.3. Key code examples taken from these tools are provided in Section 6.2.6, and these code segments can be expanded or customised for the purpose of additional CSE tool construction.

Caise-based tools by default do not have significant resource requirements. The amount of memory and network throughput consumed by typical Caise-based tools is presented in Section 7.4. The only requirements for Caise-based tools are that the Caise server is operational on a known machine within the network, the Caise tool protocol is adhered to, and the Caise tool API is available during compilation and tool operation.

No control over the interleaving of events is provided by the Caise framework. It is intended that fully synchronous views of artifacts, awareness mechanisms, feedback messages and social protocols will be adequate for coordinating the actions of developers within the project. If stronger floor control policies are required such as token passing, or locking is needed such as one person taking ownership of a specific region of code, this must be added to the Caise framework or incorporated within individual tools.

6.2.2 Tool Services

The Caise framework provides services which support the rapid development of CSE tools. By utilising the Caise framework, CSE tools rely on the Caise server to manage the storage and sharing of artifacts, and to control users as they join and leave projects and artifacts. Caise also provides low-level mechanisms to allow distributed messaging between tools and the Caise server, and supports a distributed event model.

A semantic model of the software for each project is maintained by the server, which is refined upon the actions of participating Caise tools. Caise-based tools are not required to perform any parsing or semantic analysis themselves; the server is responsible for translating modifications in artifacts to an updated semantic model. The semantic model is accessed by Caise-

المنارة للاستشارات
www.manaraa.com

based tools for reading and also direct modification through the CAISE tool API.

The functions provided by the CAISE server, both in terms of supporting collaborative work and performing core SE tasks, allow the CSE tool developer to focus on the specific requirements of the given tool rather than re-implementing functionality common to most CSE tools. If, however, the tool being developed requires additional features, the CAISE framework is easily extended to accommodate new artifact types and kinds of feedback. In Section A.3.3, the concept of framework extension is discussed further.

The CAISE *Tool API*

The CAISE *tool API* is provided as the means of accessing the functions of the CAISE server from within a CSE application. While the CAISE server typically resides on a separate machine, the CAISE tool API allows the calling application to view the server as if it was contained within the same process; the server functions appear no different to those of any other library. The server is accessed by a set of standard method calls, data is marshalled as method return values, and catchable events are thrown whenever interesting actions occur during the development of a CAISE project.

Table 6.1 presents the key CAISE tool API methods, providing a useful overview of the programming interface. A user manual for the CAISE framework, including a listing of the CAISE tool API, is available from Appendix H. The majority of the methods listed in Table 6.1 are demonstrated as coding examples in Section 6.2.6.

The CAISE tool API provides adequate functionality to implement a number of different CSE tools. Multi-user text editors, for example, can rely on the CAISE tool API to provide collaborative code editing widgets, semantic analysis of code modifications, and user presence feedback. To implement communication facilities, messaging can be provided via the Chat methods. Tool design and implementation will always be the responsibility of the CSE researcher, but the CAISE tool API prevents 'reinventing the wheel' for the essential yet complex CSE services.

المنارة للاستشارات
www.manaraa.com

Method	Description
Connect to Engine	Makes a new connection to the given CAISE server
Open Project	Opens an existing CAISE project
Add Artifact	Adds a new artifact to the given project
Open Artifact	Sets an existing artifact as open for a given user
Set User Location	Moves a user's cursor location within an artifact
Update Source Code	Appends a sequence of characters to an artifact
Update Parse Tree	Appends a parse tree of an artifact
Update Model	Directly manipulates the semantic model of a project
Get Model Snapshot	Returns a copy of a project's semantic model
Fire Tool Event	Allows a tool to invoke tool-specific server plug-ins
Get Event Log	Returns the complete event log for a given project
Send Chat Message	Allows users to send text messages between tools

Table 6.1: Key methods of the CAISE tool API.

6.2.3 CAISE *Tool Widgets*

Before discussing the construction of individual CAISE-based tools, the collaborative widgets available for use within any CSE tool are presented. This set of widgets has been produced as part of the first iteration of the CAISE framework's development; it is anticipated that the user community will contribute additional widgets.

CAISE tool widgets can be added to Swing/AWT-based Java applications without any additional coding requirements, which allow tools to be augmented with CSE capabilities for minimal effort. These widgets operate internally by communicating with the CAISE server and responding to real time events. A code listing of the GUI for a CAISE-based CSE tool is presented in Section 6.2.6.

Applications that use such components do not require any specific SE knowledge or capabilities. For example, a stand-alone text editor can be enhanced by incorporating the CAISE collaborative *User Tree* into its user interface. The User Tree will display the method, class and package that the editor is currently modifying, without the editor requiring any specific SE capabilities.

CAISE tool widgets operate by consuming CAISE-based events. As illustrated in Figure 5.4, the CAISE server generates events based on actions of

www.manaraa.com

participating tools. Various types of these events are captured by CAISE tool widgets, allowing the graphical state of widgets to be updated in real time.

There are three main types of widgets available to CAISE-based CSE tools: awareness, chat and editor widgets. Each of which will be discussed now.

Awareness Widgets

The User Tree is shown in Figure 6.1, which may be used within a CSE tool to support user awareness. This widget provides a user-centered view of CAISE-based SE projects in real time. Individual tools require no SE knowledge of the artifacts they are editing; in line with the CAISE tool protocol they simply have to keep the CAISE server informed of the name of the artifact currently being edited and the most recent cursor location of the user controlling the tool. The User Tree will keep itself updated with the latest view. The implementation of the User Tree is discussed further in Section 6.3.4.

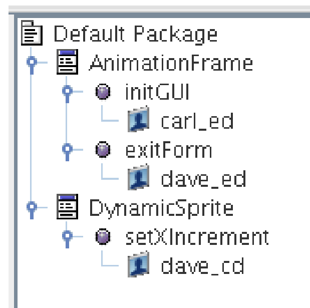


Figure 6.1: The CAISE *User Tree* widget, supporting a user-centric project view.

The *Change Graph* is another widget that can be readily added to any CSE tool. This widget is illustrated in Figure 6.2. The Change Graph widget keeps track of the cumulative additions and deletions to and from the semantic model on a per-user basis. This provides each user with an overview of the current development activity. Again, this component can be added to any CSE application, or housed in a dashboard display or separate frame.

The *Client Panel* is key component of the CAISE widgets package, and can be seen within the CAISE-based tools presented in Figures 6.20 and 3.1. The

المنارة للاستشارات
www.manaraa.com

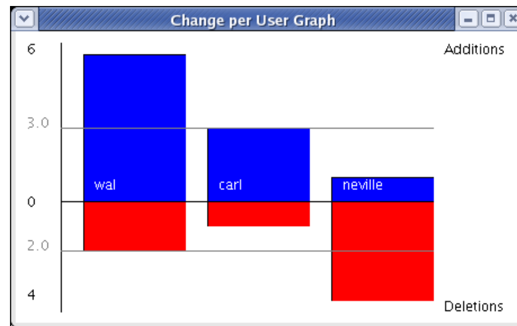


Figure 6.2: The CAISE *Change Graph* project management widget.

Client Panel typically houses four components known as the *Artifacts*, *Users*, *Feedback* and *Build Panes*, although the Client Panel can be configured to house any combination of specific panes. Individual panes can also be added to an application separately.

The Artifacts Pane, as presented at the bottom of Figure 6.20, provides file information on the artifacts within a CAISE project, including their current compilation state. The Users Pane is presented in Figure 6.3. This pane allows messages to be sent between users, including audio broadcasts.

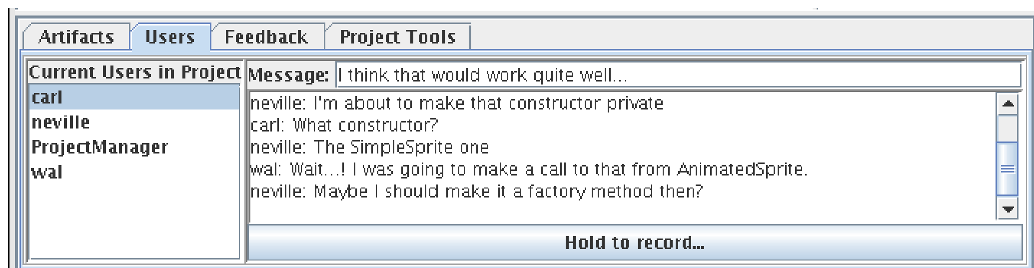


Figure 6.3: The CAISE *Users Pane*, providing voice and text communication.

The Build Pane is presented in Figure 6.4. It provides an adjustable level of collaborative awareness, allowing the user to temporarily ignore concurrent edits for the purpose of building the system without interruption. In addition to allowing the project to be built from the live, last parseable or last buildable version, the Build Pane allows the current project to be executed for testing purposes. The capabilities of the Build Pane are discussed further in Section 8.1.3.

المنارة للاستشارات
www.manaraa.com

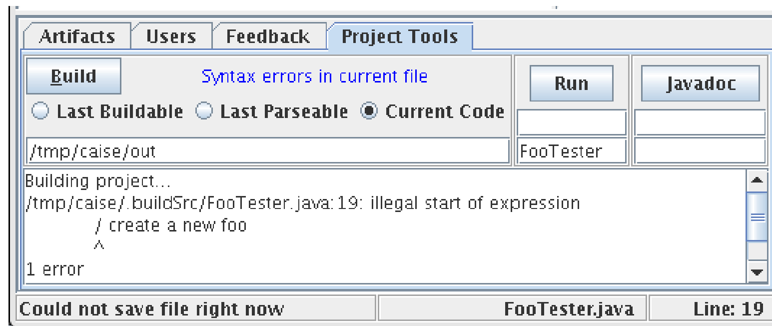


Figure 6.4: The CAISE *Build Pane* with adjustable levels of collaborative awareness.

Finally, the Feedback Pane, which was presented previously in Figure 1.2, displays plain-text information derived from semantic analysis, such as user proximity feedback between developers and impact reports that result from artifact modifications.

Chat Widgets

For developers who are co-located, communication will include face-to-face communication. For distributed developers, communication is likely to be based upon voice calling and email. The CAISE framework also assists person-to-person communication by way of text and audio chat.

Chat messages are broadcasted to all intended recipients in the same manner as all other CAISE events. Within the CAISE widgets package, the User Pane can be used as a widget to send and receive chat text messages, as demonstrated in Figure 6.3. If a custom GUI is required, then the CAISE API is used to send and receive messages in any manner desired.

For audio-based chat, the *Talk Button* widget is presented in Figure 6.5. This widget can be built into any Java application, including all CAISE-based tools. When the Talk Button is pressed, recording from the user's microphone commences. Upon release of the Talk Button, the message is serialised, sent and played to all other tools within the CAISE project that also contain a Talk Button.

The chat API is discussed further in the `caise.messaging` technical report, contained in the accompanying resources disc.

المنارة للاستشارات
www.manaraa.com

Figure 6.5: The *Talk Button* CAISE collaborative widget.

Editor Widgets

Currently only one widget to support general editing of artifacts is present within the CAISE collaborative widgets package. This widget, known as the *Collaborative Text Pane*, enables fully synchronous editing of text files through the support of the underlying CAISE framework.

The Collaborative Text Pane is presented in Figure 6.6, and is shown in use in Figure 6.20. This widget can be inserted into any Java application, providing fully synchronous text editing for any number of users, collaborative undo facilities, telecursors and remote modification highlighting.

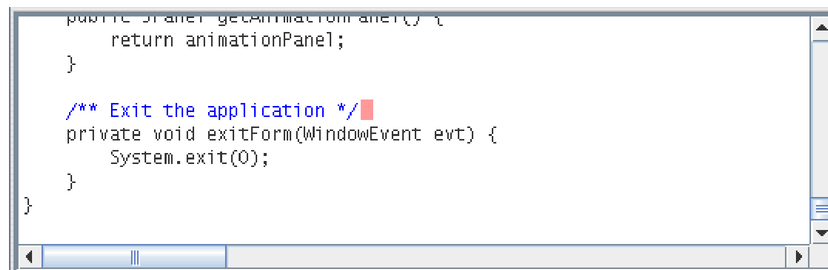


Figure 6.6: The CAISE *Collaborative Text Pane* with remote highlighting and telecursors.

As explained in Section A.3, the text contained in this widget is guaranteed to remain in a consistent state between all users, regardless of any sequence of modifications. The text pane sources its information for telecursor positions and remote text highlighting from the underlying CAISE document buffer.

www.manaraa.com

Other Widgets

Other types of collaborative widgets can be easily envisaged within the CAISE framework, including project management and awareness widgets, and perhaps sound-based awareness of remote user actions. As this is not the core focus of the research of this thesis, the search for new awareness mechanisms is currently left for other research projects such as Maui [55] to explore.

The implementation of new types of CAISE tool widgets is discussed in Section 6.3.4.

6.2.4 The CAISE *Tool Protocol*

By following the CAISE tool protocol, which specifies the contract between individual tools and the server, tools are assured of staying synchronised with each other, and the CAISE server is able to avoid concurrency issues such as deadlocks and forced roll-backs of tool requests. The CAISE tool protocol must be followed by all CAISE-based tools, otherwise indeterministic and incorrect behaviour may result.

Individual CSE tools have the ability to implement locks and other floor control policies that allow only one user at a time to edit a given region of code. By default, however, the CAISE framework allows fully synchronous editing of any artifact. To ensure that tools are always synchronised, a specialised Model-View-Controller [44] approach, by way of the CAISE tool protocol, is used which guarantees consistency over distributed parallel edits. Requests to edit the view are captured by tools, but the view is not immediately updated. Rather, the edit is sent to the server which in turn edits the global semantic model, and broadcasts the resultant change to all tools. Each tool then updates its local view, including the tool that made the edit request.

The CAISE tool protocol is formally specified in this section, but I do not intend publishing it as a standard of any kind. It is simply an implementation of a distributed version of the model-view-controller pattern, specialised for synchronous modification of shared artifacts. Within CAISE, this protocol must be followed, and developers of similar collaborative frameworks may also find the protocol suitable for adoption.

المنارة للاستشارات
www.manaraa.com

To implement a CSE tool that adheres to the Caise tool protocol, three application-level threads are typically used: a GUI thread, a worker thread and a Caise event listener thread. The threading model for Caise-based tools is presented in Figure 6.7. Most windowing toolkit libraries provide a GUI thread, and the Caise tool API provides a Caise event listener thread. As the worker thread is normally just the main application thread of the Caise-based tool, it is unlikely that any new threads need to be created explicitly within a Caise-based tool. With the existence of a worker thread, the GUI thread is free to take any volume of user input from the user interface, without causing jitter or lag as events and API commands are sent to and from the Caise server.

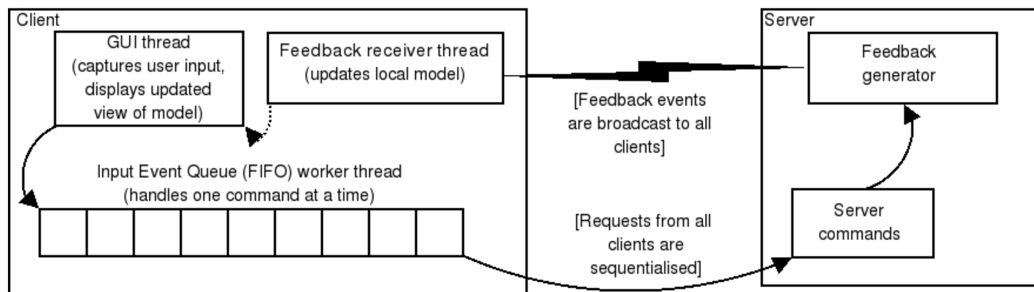


Figure 6.7: The recommended threading model within a Caise-based tool.

By using a Model-View-Controller approach and following the Caise tool protocol, CSE tools are guaranteed to stay up-to-date and synchronised with the Caise server, and there is no risk of deadlocks or loss of information.

The following list presents the six stipulations of the Caise tool protocol. This protocol describes the sequence of events presented in Figure 5.10, and a specific example of implementing the Caise tool protocol is given in Section 6.2.6.

1. The CSE tool captures all user input events such as keystrokes and caret move events, typically using action listeners. All actions are to be consumed, blocking the underlying view of the artifact from being modified

2. All captured events are placed into a FIFO event queue within the CSE

المنارة للاستشارات
www.manaraa.com

tool. The GUI thread returns immediately after placing the event in the queue, preventing any latency within the user interface

3. A separate CSE tool worker thread dequeues events in order and issues them to the server as corresponding CAISE tool API method invocations

4. The CSE tool worker thread waits for the return value of the CAISE tool API method invocation before processing the next tool input event. The CSE tool does nothing upon a successful method invocation, and escalates any errors if the method invocation fails

5. The CSE tool's CAISE event listener thread listens for broadcasted server events that result from CAISE tool API method invocations. Upon relevant events such as artifact modifications and user location changes, the tool's copy of the artifact is updated accordingly. This step is performed by all participating tools, not just the instance that invoked the event

6. Upon any semantic model update, the CSE tool's artifact view is redrawn by the GUI thread

During spikes of development by multiple CAISE tools, the server ensures fairness by queuing events evenly based on the number of contending tools, rather than absolute order of event arrival, as explained in Section A.8. In this manner, the situation where all other tools are unfairly delayed by an exceptionally active single user is avoided, at the cost of slightly unintuitive behaviour.

6.2.5 Building a New CAISE*-Based Tool*

When designing a new tool for use within the CAISE framework, four stipulations apply:

1. The artifacts that are displayed by the tool must be capable of being expressed in terms of the project's semantic model, if they are to be

www.manaraa.com

shared by all other types of tools. If not, the semantic model must be expanded for additional types of artifacts to be shared

2. If additional tool-specific information is required to be maintained, such as layout coordinates, an auxiliary artifact will be required. This is managed by a corresponding tool manager plug-in

3. The events generated by the CAISE framework must be sufficient to keep the tool synchronised with the CAISE server. If not, a new feedback plug-in will be required to provide such information

4. It is imperative that the tool follows the CAISE tool protocol, as described in Section 6.2.4

The remainder of this section discusses these four stipulations in further detail.

Bounds of the Semantic Model

All entities contained within a tool's artifacts must be able to be derived from the project's semantic model. For example, a class diagrammer can obtain all the information it requires from the semantic model, including packages, classes contained within packages, method names, scope information, and relationships such as inheritance and association. For a new type of tool, such as a use case diagrammer, the actor entity is beyond the scope of the current semantic model.

In the case where a semantic model does not incorporate specific entities for a new type of tool, two options are possible:

1. Extend the semantic model to accommodate the new entities

2. Do not model these entities as shared components throughout the framework

If option one is employed, other components within the CAISE framework such as feedback plug-ins and semantic analysers will require updating to

المنارة للاستشارات
www.manaraa.com

incorporate the new entities within the semantic model. This also allows existing tools to integrate shared views of the new entities.

If option two is employed, the entities can still be shared by using an auxiliary artifact, as described below. This simplistic approach, however, means that the server will have no knowledge about the entities. It also follows that feedback information related to these entities will not be generated.

Forming Auxiliary Artifacts

If a new type of tool is introduced into the current set of CSE tools, it is likely to introduce a new type of artifact as well. Code editors can simply display the contents of the standard Caise-based artifacts, but most tools will require additional caches of information beyond what is stored in the project semantic model. Class and sequence diagramming tools, for example, require layout information to display their respective diagrams.

When implementing new tools, additional file information is stored in auxiliary artifacts. A sequence diagrammer, for example, stores details of each sequence diagram within the project such as the sequences displayed and the methods involved. Typically, each type of tool will keep its tool-specific information private, where only instances of that tool respond to auxiliary artifact modification events of that tool type. This concept of isolating tool-specific events from other types of tools has been illustrated previously in Figure 5.7. An example of accessing and modifying auxiliary artifacts is presented in Section 6.2.6.

To be discussed further in Section A.3.3, a tool manager plug-in is required for each new type of tool that has specific artifact requirements. A tool manager plug-in for a sequence diagrammer, for example, creates or loads an artifact specific to sequence diagrams on startup, listens to layout modifications requests from instances of sequence diagrammer tools, updates the shared sequence diagramming artifact upon modification requests, and generates artifact change information for broadcast back to all participating sequence diagramming tools. Typically, layout information is held in a standard map collection, and change events are described by updated pairs of keys and values. An example tool manager plug-in is presented in Section 6.2.6.

المنارة للاستشارات
www.manaraa.com

If required, tool managers may allow tools of other types to access and even modify tool-specific auxiliary artifacts. It is difficult to envisage a situation where one tool type requires access to private information of another tool. As an example, however, a JavaDoc generation tool might produce a class listing based on the layout positions within a class diagram. To provide artifact access of another tool type, no special requirements are necessary—any tool can access and modify an auxiliary artifact as long as it knows the auxiliary artifact identifier and the tool manager identifier. Tool manager plug-ins can also be implemented with security features to control access to artifacts if required.

Providing Additional Feedback Information

Over and above the standard types of feedback delivered to CAISE tools, as described in Section 5.3.2, tool developers may require custom, tool-specific information to be broadcasted throughout the CAISE framework. An example might a project management tool that requires being informed whenever two users are located within units of code linked by a superclass/subclass relationship. Upon receipt of such feedback information, the project management tool could inspect the areas of code in detail and issue precautionary warnings if the units of code are deemed to have a high level of overlap.

To facilitate customised feedback, a CAISE-compliant feedback plug-in is created and loaded into the CAISE server. Feedback plug-ins are notified every time that a CAISE tool generates an input event such as a user changing location, and at this point, the feedback plug-in can inspect the project's semantic model and artifacts. If the feedback plug-in determines that a tool-specific feedback event should be broadcasted, it returns such an event to the CAISE server, which will distribute it accordingly.

All CAISE-compliant feedback plug-ins must conform to the CAISE FeedbackPlugin interface, as presented in Appendix D. Output from the CAISE DOI user presence plug-in is presented in Figure 1.2. This feedback plug-in is discussed in detail in Section 6.2.6, which includes a source code listing.

CAISE-based feedback plug-ins decouple the task of gathering SE information from CAISE-based tools. For a code editor that is to remain simple

المنارة للاستشارات
www.manaraa.com

and independent of specific SE methodologies, it is preferable to implement feedback by way of CAISE-based feedback plug-ins. In other cases, smarter kinds of tools may derive information locally, such as metrics for currently opened source files.

Responding to Local Modification Requests

A demonstration of how tools respond to modification requests by local users, as per the CAISE tool protocol, is given in Section 6.2.6. For the construction of new tools, a discussion is given here.

For a sequence diagrammer, typical modification requests from the user include the creation of a new sequence diagram, adding a new class or method to a diagram, or adding or removing a sequence from a diagram. For a use case diagrammer, typical modification requests include adding a new actor, or adding a new class or method. Deletion or renaming of entities is also possible.

The majority of these requests can be facilitated through the CAISE tool API. For tool-specific requests, such as changing the layout of a diagram, or modifying a component contained within a tool-specific artifact rather than the semantic model, then a tool manager request is issued to update the tool-specific artifact.

Tool Initialisation

The design of the start-up routine for new tools is typically as follows:

1. When the tool starts up, it establishes a connection to the CAISE server, as demonstrated later in this section

2. Upon server connection, it will download a copy of the project's semantic model (if required), core artifacts, and any tool-specific auxiliary artifacts

3. It will then display its tool-specific views, based on the downloaded project information

المنارة للاستشارات
www.manaraa.com

4. Listeners for CAISE-based events are activated, allowing the local copy of the semantic model and artifacts to be updated as required

5. Listeners are also activated for changes made from the local tool user, which are captured and directed to the CAISE server

In the next section, code examples for all the above operations are given.

6.2.6 Coding Examples

The code segments presented in this section are taken from the Java code editor and UML diagrammer, which are discussed in Section 6.3. These code examples represent the full set of actions necessary to complete the cycle of events presented in Figure 5.10, and can be used as a starting point for additional CSE tool development.

Connecting to the CAISE *Server*

Each instance of a CAISE-based CSE tool needs to establish a connection to the CAISE server. The most appropriate time to do this is at program startup. Within CAISE, each user has a unique name, and this is given during the call to establish a server connection. The code segment presented in Figure 6.8 demonstrates connecting to a named CAISE server, registering an application for CAISE events, and opening an existing project.

```
// create a new instance of a CAISE handler
CAISEHandler handler = new CAISEHandler(clientName,
                                        serverName, TextEditor.ID);

// tell the server to notify this class of any events
handler.attachCAISECallback(this);

// open the initial CAISE project with no event filtering
handler.openProject(projectName, CAISEEvent.ALL_EVENTS);
```

Figure 6.8: Initialising a CAISE-based CSE tool.

المنارة للاستشارات
www.manaraa.com

Adding CAISE *Widgets to a Tool*

CAISE widgets may be added as components within a user interface in the same manner as any other standard graphics widget. The only requirement is that events from the CAISE server are passed from the containing application to each widget. Event handling code is given later in this section.

In the code segment presented in Figure 6.9, it is apparent that adding CAISE-based collaborative widgets to a Swing/AWT Java application is no different than constructing a conventional user interface.

```
// build the gui for the editor
private void initGUI() {
    // create a new user tree
    userTree = new UserTree();
    userTree.setPreferredSize(new Dimension(150, 500));
    // create a new client panel to display shared artifacts
    clientPanel = new ClientPanel(handler, statusBar);
    // create a new shared text editor pane
    sharedEditor = new JavaTextPane(this, handler.getClient());
    // add the shared editor pane to a scrollable pane
    textPanel = new JScrollPane(sharedEditor);
    // create a split pane
    topPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT);
    // put the shared text editor on the left
    topPane.setLeftComponent(textPanel);
    // put the user tree on the right
    topPane.setRightComponent(userTree);
    // create the main application pane
    mainPane = new JSplitPane(JSplitPane.VERTICAL_SPLIT);
    // put the split pane as the top component
    mainPane.setTopComponent(topPane);
    // put the client panel as the bottom component
    mainPane.setBottomComponent(clientPanel);
}
```

Figure 6.9: Adding widgets to a CAISE-based tool.

www.manaraa.com

Downloading Server Resources

The code segment presented in Figure 6.10 demonstrates how the Java code editor tool downloads an artifact from the CAISE server for collaborative editing. By downloading the artifact, the code editor tool is now in a position to allow local modifications to the file by way of the CAISE tool protocol. Code editors do not typically need a copy of the project's semantic model, as they operate purely at the artifact level.

```
// open a shared file from the caise server,
// assuming that the handler exists and a project has been opened
private void openFile(String fileName) {
    // set the contents of the editor pane to the current source code buffer
    content.setText(handler.openFileAsSourceCode(fileName));
}
```

Figure 6.10: Downloading a CAISE artifact.

In Figure 6.11, the startup routine for the Java UML diagrammer is presented. After connecting to the CAISE server and a named project, the UML class diagrammer downloads a copy of the complete project semantic model and the diagrammer-specific auxiliary artifact which contains all the class layout information. A copy of the semantic model is required for the diagramming tool to extract fine-grained project information such as packages, classes, methods and visibility details.

Catching Local Tool Actions

The CAISE tool protocol stipulates that tools pass artifact modification events to the CAISE server, instead of allowing the tool's view of an artifact to be modified directly. To do so, tools must catch all artifact modification actions and queue them for subsequent proxying to the server. Only once the event has been processed by the server and a response has been broadcasted to all tools will local views be updated, as illustrated later in this section.

In the code segment presented in Figure 6.12, the key presses destined for the text pane within the Java editor are captured and queued. The current

المنارة للاستشارات
www.manaraa.com

```
// do start-up downloads for class diagramming tool
private void downloadModel() throws Exception {
    // get the latest version of the model from the server
    model = handler.getModel(projectName);

    // download the auxilary artifact (map of class diagram positions)
    viewMap = (Map) handler.openAuxilaryArtifact
        (JavaClassDiagrammerToolManager.ARTIFACT_NAME,
         JavaClassDiagrammerToolManager.PLUGIN_ID);
}
```

Figure 6.11: Downloading the semantic model and an auxiliary artifact.

cursor location is not recorded within the keystroke event—the server maintains the authoritative record of current user positions to ensure consistency between all the tools, and already knows the user location at the time of the pending key press. To maintain the record of user locations, cursor location changes are another type of Caise event governed by the Caise tool protocol.

Sending Tool Actions to the Server

As described in Section 6.2.4, the GUI thread is only responsible for capturing and enqueuing user input, and updating the local view of artifacts. The role of each Caise-based tool's worker thread is to take events from the local event queue and deliver them to the server as API method calls. As illustrated in the code segment of Figure 6.13, the worker thread blocks until a corresponding event has been broadcast by the server before processing any remaining queued events.

The code segment presented in Figure 6.14 illustrates the routine for the Java UML diagrammer where the layout of the diagram is locally modified. The tool responds to local modification requests by updating the class diagram stored as an auxiliary artifact on the Caise server.

www.manaraa.com

```
public void keyTyped(KeyEvent e) {

    // kill it before it gets to the editor
    e.consume();

    // ignore any keystrokes that involve alt or ctrl
    if ( (e.getModifiers() & (e.ALT_MASK|e.CTRL_MASK)) )
        return;

    // ignore escape key
    if (e.getKeyChar() == (char)27)
        return;

    // add regular key event to client queue
    enqueEvent(new EventWrapper(e, fileName));

    // update the state of the undo menu item
    EditorFrame.this.undoItem.setEnabled(true);
}
```

Figure 6.12: Implementing a key listener within a collaborative text editor.

Listening for Server Responses

The Caise server broadcasts events to all registered listeners upon any significant event such as an artifact modification or a change in the project's underlying semantic model. If a tool has issued a request to modify an artifact, the server will perform the modification on its master copy and then broadcast a corresponding event to all tools. The tool that issued the request will be expecting a subsequent modification event, and all other tools are also required to adjust their local artifact views upon event notification.

The code segment presented in Figure 6.15 illustrates the main event loop within the Java text editor, which is representative of typical Caise-based tools. As the editor also employs the User Tree widget, events are relayed to the widget, allowing it to update its own view of the project. The text editor also needs to keep track of user location changes in the same manner as it monitors artifact modification events, but for the sake of simplicity, this has been omitted from this example.

www.manaraa.com

```
// routine to empty GUI input event queue
final class EventHandler implements Runnable {

    public void run() {
        while (isThreadRunning()) {

            // remove event from queue and pass to server
            EventWrapper ew = clientInputEvents.take();

            if (ew.event instanceof KeyEvent)
                // send key events as buffer append requests
                handler.appendSourceCodeBuffer(ew.fileName,
                                               ew.event);

            // wait until the server has replied
            serverFeedbackEvents.take();
        }
    }
}
```

Figure 6.13: Sending a local tool action event to the Caise server.

Updating the Local Artifact View

To complete the Model-View-Controller pattern within the Caise event model, the final task for CSE tools upon receiving an event is to update their local view. Within the text editor, this involves appending and re-displaying the text pane upon artifact modification events, as presented in the code listing of Figure 6.16. For user location change events, this involves updating the local mapping of users and file positions and re-displaying all cursors. As the Java code editor uses a multi-user text component, artifact modification events only need to be relayed to the Collaborative Text Pane—this multi-user component will perform the text insertion and remote modification highlighting internally.

It is important to note that each tool's view runs no possibility of losing synchronisation with other tools or the Caise server, barring catastrophic network failure. As long as events are captured and delivered in order to the server, and the underlying artifact is only updated within each tool in

المنارة للاستشارات
www.manaraa.com

```
// inform server that a component within the diagram has been moved
private void issueMoveComponentEvent(Decl component, int xPos, int yPos) {

    // create new event to pass to server
    CAISEEvent event = new CAISEEvent(PLUGIN_EVENT, TOOL_MANAGER);

    // the plugin responsible for handling this event
    event.setID(JavaClassDiagrammerToolManager.PLUGIN_ID);
    // set the source entity: the component that has moved
    event.setSourceEntity(new Integer(component.getID()));
    // set the event data: the new positions
    event.setData(MOVE_COMPONENT, xPos, yPos);
    // set the sender
    event.setSourceUser(handler.getClient());
    // hand to server - this will make its way to the correct plugin

    handler.throwToolEvent(event);
    // now we just wait for the response from the event queue...
}
```

Figure 6.14: Modifying an auxiliary artifact.

response to server events, then synchronisation is guaranteed.

In Figure 6.17, the update routine for the Java UML diagrammer is presented. In this code segment, the UML diagramming tool responds to two events: core artifact modifications and changes to the class diagram auxiliary artifact. When a source file has been modified, the Java UML diagrammer immediately updates its own copy of the project's semantic model. When the class diagram auxiliary artifact has been modified, the local copy of the artifact is updated. Upon any artifact modification, the local view of the Java UML diagrammer is then redisplayed.

Providing Customised Feedback Events

In Figure 6.18, the structure for a custom feedback plug-in is presented. Feedback plug-ins, as described in Section 6.2.5, are invoked by the Caise server whenever the state of the project changes. As can be seen in the

www.manaraa.com

```
public void update(Collection events) {

    // for each event
    for (Iterator i = events.iterator(); i.hasNext(); ) {
        CAISEEvent event = (CAISEEvent)i.next();

        // inspect the event type
        switch (event.getType()) {

            // if an artifact event
        case CAISEEvent.ARTIFACT_EVENT:

            // if the artifact has been edited by anyone
            if (event.getSubType() == ARTIFACT_APPENDED)

                // if this is the current artifact
                if (event.getSourceEntity().equals(fileName))

                    // append the buffer of the underlying file
                    appendBufferFromRemoteChange(
                      event.getSourceUser(),
                      ((KeyEvent)(event.getData())[0]),
                      ((Integer)(event.getData())[1]).intValue(),
                      ((Integer)(event.getData())[2]).intValue());
        }

        // update user tree
        userTree.updateTree(event);
    }
}
```

Figure 6.15: Processing events thrown by the CAISE server.

www.manaraa.com

```
private void appendBufferFromRemoteChange(Client editor,
                                          KeyEvent change,
                                          int positionHint,
                                          int previousFileSize) {

    // check that our user location is in sync with the server
    assertUserLocation(editor, positionHint);

    // check that the file size is in sync with the server
    assertFileSize(previousFileSize);

    // update buffer
    buffer.appendDocument(change.getKeyChar(), positionHint,
                          editor, handler.getClient());

    // restore any previously selected text
    redrawSelection(editor.equals(handler.getClient()));

    // tell auto-save timer to restart
    setBufferDirty(true);

    // yeild lock if this edit originated from this app
    if (editor.equals(handler.getClient()))
        serverFeedbackEvents.put(new Object());
}
```

Figure 6.16: Updating the local view of the Java code editor based on framework events.

given example, the feedback plug-in inspects the state of the current project, and returns a collection of user-specific feedback events. These events are propagated to the relevant participating CAISE tools according to the CAISE framework's event model.

While not given in this code example, the algorithm to derive relational feedback is not complicated; for each user location a depth-first traversal of the semantic model is performed using classes, methods and packages as nodes. Associations, method invocations and inheritance structures are used as edges. A DOI function is also applied to ensure that weak relationships are excluded from the search results. As an example of the DOI function, all

www.manaraa.com

```
// handle events thrown to us by the caise server
private void handleRemoteEvent(CAISEEvent event) {

    // if the class diagrammer's auxilary artifact has been modified
    if (event.getType() == PLUGIN_EVENT) {

        // update our local copy of the map
        viewMap.put(event.getSourceEntity(), event.getData());

        // if event was initiated by this tool instance, yeild lock, allowing the
        // next event in our input event queue to be passed to the server
        if (event.getSourceUser().equals(handler.getClient()))
            serverFeedbackEvents.put(event);
    }

    // if an artifact within the project has been updated
    if (event.getType() == ARTIFACT_SAVED) {

        // create temporary artifact
        Artifact artifact = new Artifact((String)event.getSourceEntity(),
                                         event.getSourceUser(), null);

        // set parse tree buffer to the one received
        artifact.commitParseTreeBuffer(event.getData()[1]);

        // merge new parse tree with local model
        model.addArtifact(artifact, event.getSourceUser());

        // determine all declarations in updated model
        modelDrawer.reloadDecls();
    }

    // redraw the class diagram panel
    repaint();
}
```

Figure 6.17: Updating the view for the UML class diagrammer.

www.manaraa.com

```
final public class RelationalFeedback extends CAISEFeedback {

    // return all feedback events for this instance of time
    public Map getFeedback(Project project) {

        // for each artifact
        for (Iterator artifacts = project.getArtifacts().iterator();
              artifacts.hasNext();) {

            Artifact artifact = (Artifact)artifacts.next();

            // for each viewer in that artifact
            for (Iterator viewers = artifact.getViewers().iterator();
                  viewers.hasNext();) {

                Client localViewer = (Client)viewers.next();

                // generate warnings for all outwards references
                feedback.add(getFeedbackEvents(localViewer, artifact, project));
            }
        }
        return feedback;
    }

    // generate user-specific feedback events given the current model state
    private Set getFeedbackEvents(Client client, Artifact artifact, Project project) {
        // Walk through the model determining relationships between users and code.
        // Each time a relationship is discovered, add a new feedback event
        // to the given collection

        /* ... */
    }
}
```

Figure 6.18: Implementing a custom feedback plug-in.

المنارة للاستشارات
www.manaraa.com

classes in Java are related by the Object superclass; the DOI function deems this relationship as unimportant, otherwise all users would be considered related to each other at all times, producing spurious feedback messages.

The user presence feedback plug-in can be extended to provide additional feedback information, which will be in turn displayed in the Feedback Pane. New types of feedback widgets can be developed to present different types of feedback information, where the source information is derived from custom feedback plug-ins.

Implementing a Tool Manager Plug-In

The basic skeleton for the Java UML diagrammer tool manager plug-in is provided in Figure 6.19. The tool manager is responsible for providing access to all tool-related auxiliary artifacts stored on the CAISE server. The routine for updating the auxiliary artifact is also given, where first the artifact is modified, and then a modification event is generated for propagation to all participating tools. Each tool that maintains a local copy of the auxiliary artifact will then update its version upon receipt of the modification event.

6.3 Example CSE Tools

In this section, example CSE tools are presented, including code editors, UML diagramming tools and visualisation tools. In Section 6.3.3, an IDE is also demonstrated collaborating within the CAISE framework.

Many of the tools presented in this section have undergone heuristic evaluations to ensure their quality. Heuristic evaluations for CSE tools are discussed in Section 7.1. Additionally, a detailed user evaluation of the Java text editor and UML class diagrammer is presented in Section 7.3.1.

The tools presented in this section have been provided to give designers insight into the capabilities and potential of the CAISE framework to support new types of CSE tools. Demonstrations of various tools presented within this section as they execute a range of tasks are available from `www.cosc.canterbury.ac.nz/clc/cse`.

المنارة للاستشارات
www.manaraa.com

```
public class JavaClassDiagrammerToolManager extends CAISEToolManager {

    // user routine - called by server upon new project creation
    public void init() {
        parentProject.getAuxillaryArtifacts().put(ARTIFACT_ID,  new HashMap());
    }

    // return the correct artifact from the server, and remember who has opened it
    public Object openAuxillaryArtifact(String artifactID, Client requestor) {
        // retrieve the map.
        Map view = parentProject.getAuxillaryArtifacts().get(ARTIFACT_NAME);
        // add this client to the list of viewers
        viewers.add(requestor);
        // return map
        return viewMap;
    }

    // inform the server that the location of a component has changed
    public CAISEEvent processToolEvent(CAISEEvent evt) {
        // get auxillary artifact (class layout info) for this tool manager
        Map view = parentProject.getAuxillaryArtifacts().get(ARTIFACT_NAME);
        // handle the request (first integer in int array)
        switch (evt.getData()[0]) {
        case MOVE_COMPONENT:
            // update the underlying auxillary artifact
            LayoutPosition pos = (LayoutPosition)viewMap.get(evt.getSourceEntity());
            pos.setPosAndDeclID(evt.getData()[1], evt.getData()[2], evt.srcEnity());
            // generate response
            CAISEEvent outEvent = new CAISEEvent(evt);
            // set data to the component wrapper
            outEvent.setData(wrapper);
            // return response
            return outEvent;
        }
    }

    // remove this viewer from the set of viewers for this artifact
    public void closeAuxillaryArtifact(String artifactID, Client requestor) {
        // retrieve the map.
        Map view = parentProject.getAuxillaryArtifacts().get(ARTIFACT_NAME);
        // remove viewer
        viewers.remove(requestor);
    }
}
```

Figure 6.19: Implementing a tool manager plug-in.

www.manaraa.com

6.3.1 Code Editors

Code editors are a fundamental tool for software engineers. To demonstrate how the CAISE framework can support different types of code editors, we present three CAISE-based text editing tools.

A Java Code Editor

Collaborative editors are difficult to implement. Every operation that a shared editor supports, such as modification of text, cut and paste, undo facilities, and text selection, must be performed with the assumption that the document may change at any time. Additionally, the regions changed by several users have the potential to overlap.

Fortunately, code editors are well supported within the CAISE framework, which helps reduce the implementation workload. Services provided by CAISE for code editors include: artifact management, support for shared editing, build and run facilities, impact reports, user presence information, chat communication, a full semantic model to query for name completion, and various collaborative widgets such as the User Tree and Change Graph.

A CAISE-based collaborative editor for Java is presented in Figure 6.20. Features of this editor beyond those listed in Section 6.1 include:

- A multi-user text pane which provides remote modification highlighting and telecursors (A)
- A collaborative User Tree that provides a semantic model-based view of developers' locations (B)
- An Artifacts Pane that displays the current compilation state of each artifact as well as editor details and file information (C)
- Code repository support, discussed further in Section 8.1.1
- Collaborative undo support, discussed further in Section A.3.2

المنارة للاستشارات
www.manaraa.com

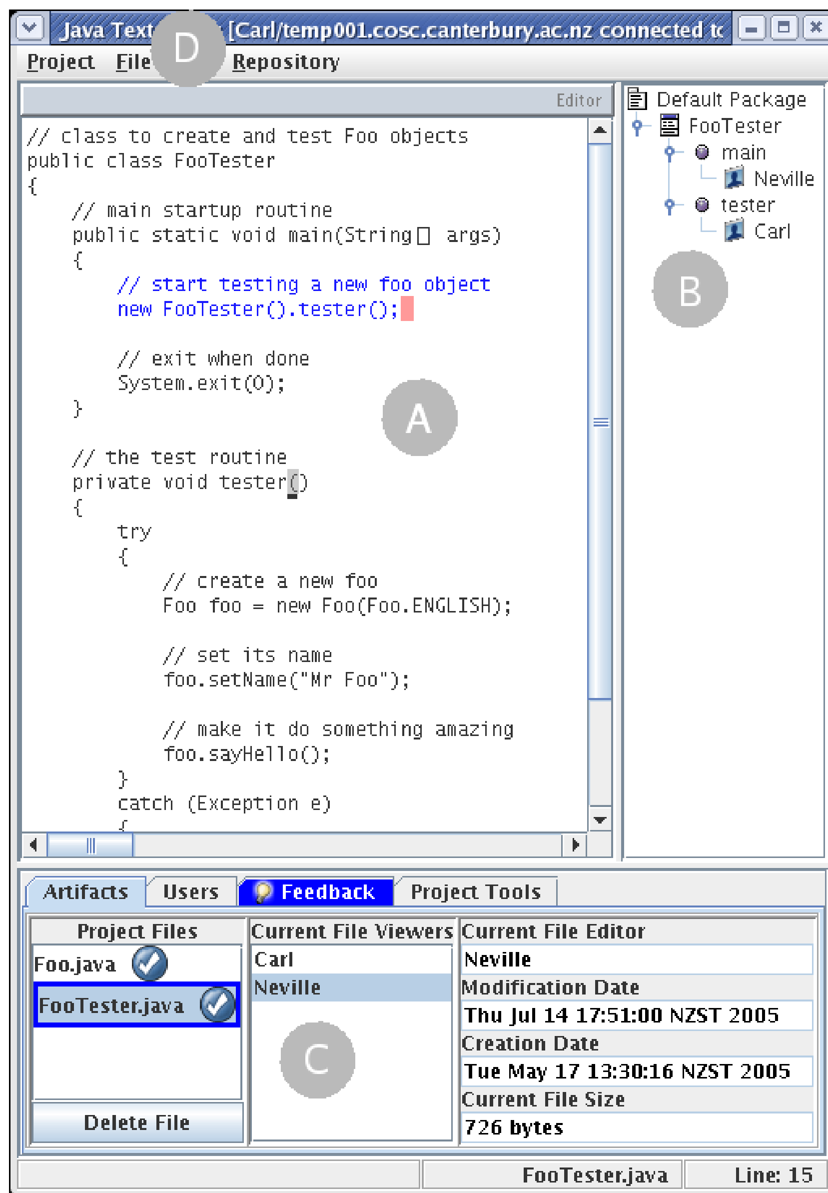


Figure 6.20: A CAISE-based collaborative code editor for Java.

المنارة للاستشارات
www.manaraa.com

In Figure 6.20, label D presents information on the title bar. This information contains the name of the current user, the project under development, and the location of the CAISE server.

The Java text editor provides a representative example of the capabilities of CAISE-based tool construction. This tool simply adheres to the CAISE tool protocol, and follows the design stipulations presented in Section 6.2.5 to provide multi-user artifact editing.

Software developers can use the Java text editor to work collaboratively within a project, modifying source files, compiling the project, performing testing and completing code reviews. Two or more users can edit the same file at the same time if desired, or work can be performed between files with the knowledge that all modifications are atomically integrated. Additionally, all other CAISE-based tools can operate on the same project, sharing the artifacts in real time.

A Code Editor for a Custom Language

The Java editor presented in Figure 6.20 could easily be extended to support other languages. Prior to the development of the fully-featured Java editor, however, a more simple code editor, presented in Figure 6.21, was developed to support the Decaf language. The Decaf language is described in Appendix B.

The Decaf code editor has the same basic functionality as the Java editor such as collaborative text editing and user presence feedback, albeit with a less comprehensive user interface. It is a proof-of-concept prototype to illustrate that tools for multiple languages can be supported within the CAISE framework.

A Code Age Editor

Code age displays were originally proposed as part of the SeeSoft visualisation package [36]. A code age display of a source file provides a line-by-line shading of code, where the level of shading is governed by properties such as the age of the code or the number of times that line of code has been modified. Code age displays are useful for quickly conveying to developers areas of interest

المنارة للاستشارات
www.manaraa.com

Figure 6.21: A Decaf collaborative code editor.

or concern within a set of source files.

Normally, it would be a complicated task to collate the information required to provide a code age display of a source file. The revision history for each file would be mined from a code repository, and then line-by-line properties would be calculated. Within the Caise framework, however, a code age editor is almost trivial to support.

A Caise-based code age editor is presented in Figure 6.22. This editor shades each line within the editor at a different level based on the number of modifications each line has received. The number of entries in the change log for each declaration increases the shading level for the corresponding line in the source file. If a finer level of detail is required, shading could be performed for each declaration in the source file instead of each line.

To derive the number of modifications per line, the tool simply inspects

المنارة للاستشارات
www.manaraa.com

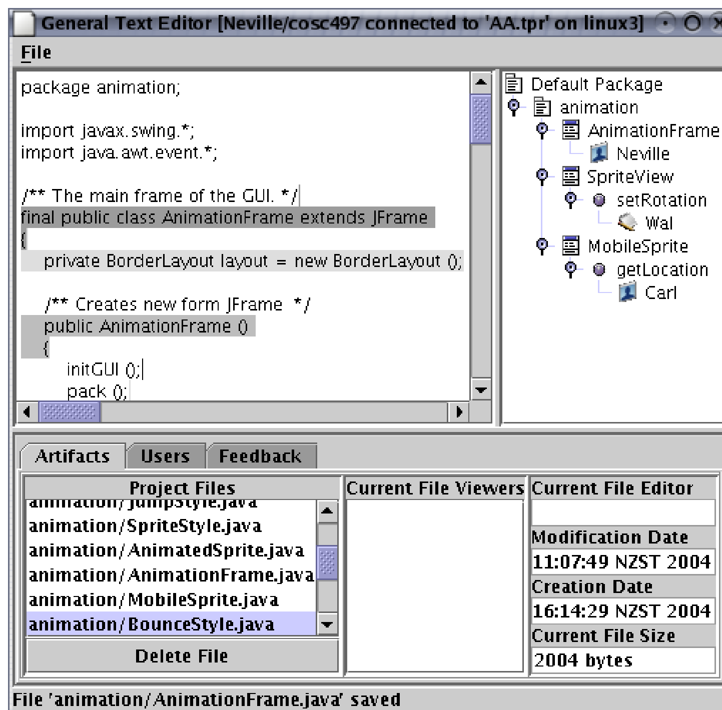


Figure 6.22: A code age collaborative text editor.

the underlying parse tree for the artifact supplied by the CAISE server, which incorporates a change log for each declaration. Alternatively, the CAISE event log could be inspected, but this would be computationally inefficient by comparison. Redrawing of the code age display occurs every time the artifact being displayed is changed; notification of this is facilitated by CAISE artifact modification events. The sequence of events for updating the code age display is illustrated in Figure 6.23.

It is important to note that this CAISE-based code age tool is not just a display tool but an actual editor as well. It currently only supports the Java language, but it can be easily extended to support the Decaf language as well. Additionally, the code age editor is collaborative: multiple users can edit the same artifact in real time, and any subsequent modifications will update the code age display as they happen.

The code age editor is provided as a demonstration of the potential of CAISE-based CSE tools. Numerous similar tools can be envisaged, such as

المنارة للاستشارات
www.manaraa.com

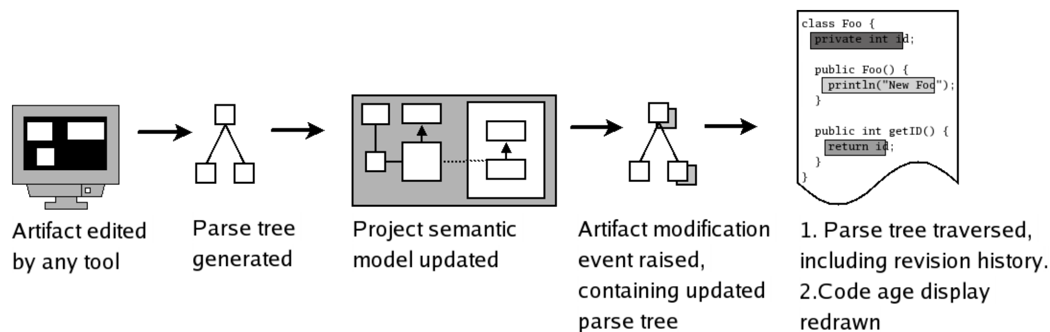


Figure 6.23: The event sequence for updating a code age display.

read-ware [56] applications, where declarations within artifacts are shaded according to the number of times they have been read by developers.

6.3.2 Diagramming Tools

Another common type of tool within SE development teams are diagrammers. Many different types of diagrams can be supported by SE tools today, such as class, sequence, state and use-cases. With the advent of round-trip engineering, often diagramming tools and code editors can be integrated, where a change in one tool will be reflected immediately in the other.

To demonstrate the capabilities of the CAISE framework, several diagramming tools have been implemented. As each of these tools adheres to the CAISE tool protocol, round-trip engineering is supported between these tools and all other tools within the CAISE framework.

A UML Class Diagramming Tool

A collaborative UML class diagrammer is presented in Figure 6.24. This diagramming tool supports many common operations such as add, delete and rename for classes, methods, properties, parameters and superclasses. The full semantic model is inspected by the UML diagramming tool, allowing a fine level of detail to be displayed, including interfaces, abstract classes and methods, and visibility information.

The UML class diagramming tool is fully collaborative. A change made

www.manaraa.com

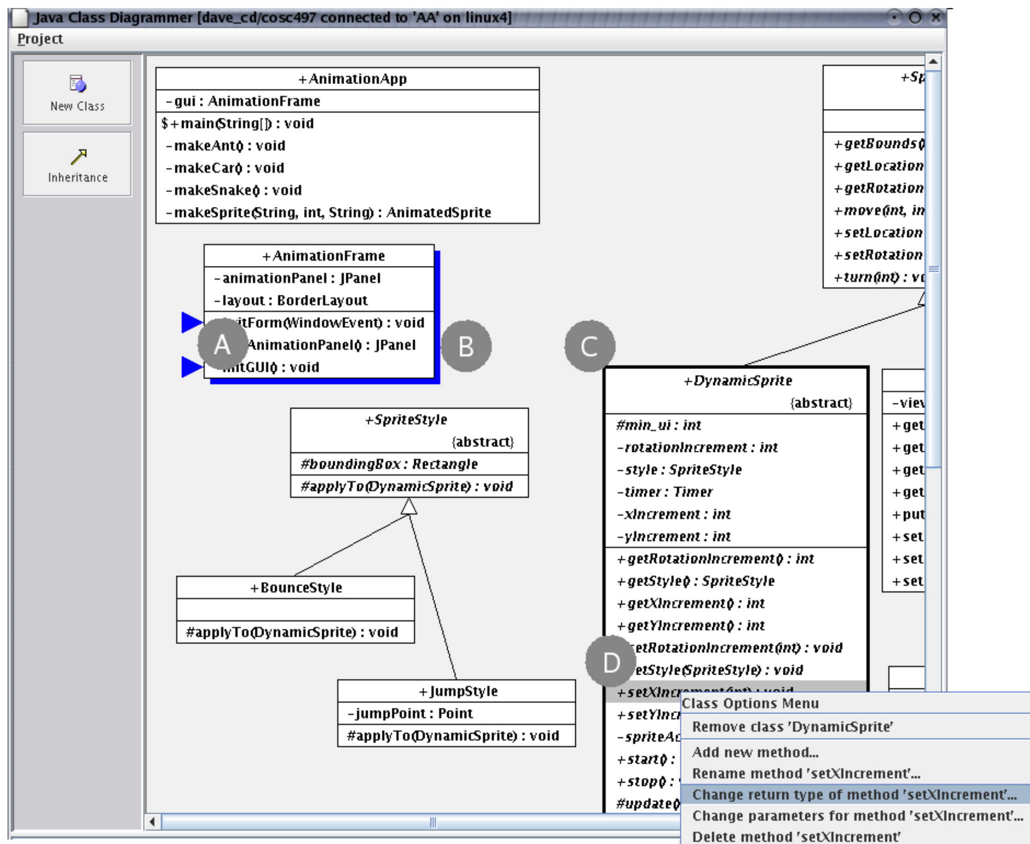


Figure 6.24: A collaborative UML class diagrammer.

from within the diagrammer is immediately propagated to all other diagrammers, and all other participating CAISE tools as well. Again, support for collaboration is provided by the CAISE framework and server, which helps reduce the tool development effort.

To support user presence feedback, the UML diagrammer includes annotations to indicate remote developer positions. These annotations are visible in a colour version of Figure 6.24; the blue shaded triangles (A) indicate remote user locations, and dynamic tool-tips are available to give further information. Classes currently visited by remote users are highlighted by a blue border (B), and classes selected within the tool are highlighted by a black border (C). All user presence information is derived from feedback events, reducing the development effort and workload of the diagramming tool.

المنارة للاستشارات
www.manaraa.com

When a region of a class is selected in the diagrammer, a black border and gray shading is drawn around the surrounding declarations (D). As part of the CAISE tool protocol, the diagramming tool informs the CAISE server about all changes in focus, which allows user proximity information to be calculated and broadcasted to all relevant tools.

Changes to the view of the diagram, such as a repositioning of a class, are sent to other instances of the class diagrammer, allowing each diagrammer's relaxed WYSIWIS view to remain synchronised. To implement this, each time a class diagrammer component is repositioned, the drag action is captured and sent to the CAISE server via the `fireToolEvent()` API method. The UML diagrammer tool manager plug-in responds to this event by adjusting its mapping of components and coordinates, and then broadcasts the drag event out to all tools registered for this event, which in turn update their local views of the semantic model. Tool managers are discussed further in Section A.3.3.

The UML class diagrammer has only been used within Java-based projects. There is no reason why it can not work within other languages, however, as its main interaction is with the semantic model of software, not language-specific parse trees or source files. Only minor modifications to support other languages are anticipated, such as ensuring that new source files are created using the correct language-specific file extensions.

A Class Diagrammer for a Custom Language

Another diagramming tool developed within the CAISE framework is presented in Figure 6.25. This diagramming tool was developed for the Decaf language, and served as a prototype and proof-of-concept prior to the development of the more powerful Java UML class diagramming tool.

The Decaf class diagramming tool is reasonably trivial, but it provides a further demonstration of one of the key properties of the CAISE framework—that of multiple language support. The Decaf class diagrammer operates on the same semantic model of software that Java tools are based upon, but on the assumption that the project is configured for Decaf source files rather than Java-based ones.

المنارة للاستشارات
www.manaraa.com

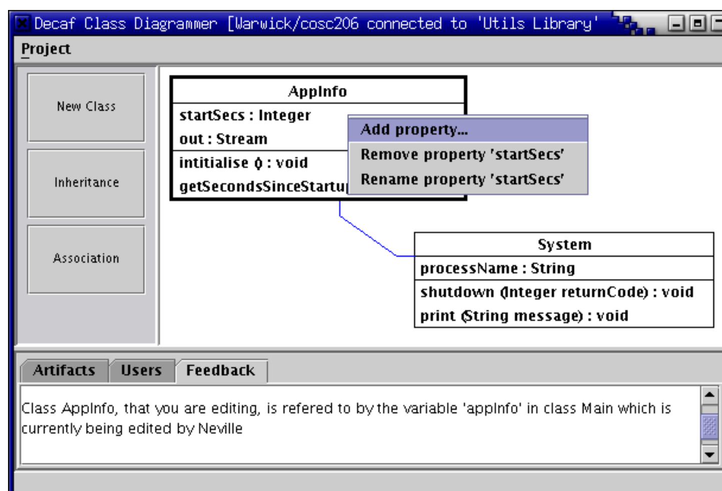


Figure 6.25: A Decaf collaborative code editor.

The Decaf class diagramming tool was again relatively simple to implement using the CAISE tool protocol and the existing functionality of the CAISE server. Feedback information related to user proximities, for example, is generated from the general semantic model of software within the CAISE server, without the need for any language-specific extensions. This information is sent to all relevant tools, including instances of the Decaf diagrammer.

Several IDEs, such at Together Architect [46], support various modes of operation. These include an analysis mode, where only a very restricted set of operations is supported, and development mode, where low-level views of the software project are presented. Within the context of the CAISE framework, different modes of operation can be supported by various types of CAISE-based tools. For example, the Decaf class diagramming tool can be extended to support a limited and high-level range of operations for Java-based projects such as adding a new class. For more comprehensive, low-level support, the UML class diagramming tool presented in Section 6.3.2 can be used.

للاستشارات المنارة
www.manaraa.com

6.3.3 IDE Integration

Fundamental to the motivation of Caise is that Cse tools and architectures should not be limited to a particular tool or programming environment. A strong test of the Caise approach is the integration of existing IDEs into the framework. It is important to be able to demonstrate that the Caise framework can support existing SE tools, otherwise the claim that the framework is genuinely useful and extensible is difficult to assert. Details of integrating Together Architect into the Caise framework are presented in Appendix E.

6.3.4 Constructing Collaborative Widgets

In this section, the focus has been on the construction of various new types of CSE tools. As the Caise framework provides open access to rich sources of project information and propagates fine grained user actions in an event-based manner, new types of collaborative widgets can also be easily constructed. Once implemented, these widgets can be used to further extend existing Caise-based tools.

The general strategy for most Caise-based widgets is to listen for relevant events, query the semantic model if required through the Caise tool API, update any local cache of information, and then redraw the current view.

The User Tree presented in Section 6.2.3, for example, simply listens to Caise events that specify user location changes. Caise events contain several fields of information, as discussed in Section 5.3.2, including identifiers to locate components within the project's semantic model, the user that caused the event to be fired, and the time that the event originated. The event data for user location changes includes the fully qualified name for the containing declaration and the name of the user that changed location. This information is all that is needed for the User Tree to update its local information and then redraw its tree view.

If a more complex collaborative widget is required, it is possible to inspect the semantic model to gain more detailed information based on specific events. For example, it is possible to display all subclasses immediately under the scope of the currently visited class within the User Tree. While this information is not contained directly within each user location event, the

المنارة للاستشارات
www.manaraa.com

User Tree widget simply needs to query the project's semantic model to obtain the current subclasses for each given declaration, and then display this information as desired.

Summary

In this chapter, the construction of CAISE-based tools has been discussed and demonstrated. Several different types of CSE tools have been presented, and such tools are typical of those likely to be of use in collaborative development settings. Given these various tools and the accompanying discussion on how new tools can be constructed, it can be asserted with reasonable confidence that the CAISE framework is complete for the purpose of supporting most collaborative development requirements.

In Chapter 7, detailed evaluations of CAISE-based tools are presented. The evaluations show software development scenarios where CAISE-based tools are preferable over their conventional counterparts. The evaluations also validate the quality of the CAISE-based tools presented in this chapter, and help confirm the suitability of the CAISE framework in supporting CSE tools.

المنارة للاستشارات
www.manaraa.com

Chapter VII

Evaluation of the Caise Framework and Tools

Within this thesis the use of the Caise framework has been advocated as an approach to supporting CSE. It has been demonstrated that many types of CSE tools can be constructed within the Caise framework.

Regardless of anecdotal reasoning, Caise-based CSE tools require empirical evaluation to verify claims of usability. Such evaluations will also allow the exploration of the perceived benefits of working collaboratively in real time.

In this chapter, various assessments of the Caise framework and supporting tools are made. These evaluations are made in order to determine the suitability of the Caise framework as an approach to supporting CSE, and to provide insight into the viability of the current set of Caise-based tools.

In Section 7.1, heuristic evaluations for CSE tools are proposed and related to the Caise-based CSE tools. In Section 7.2, tools to analyse activity within Caise-based projects are discussed. In Section 7.3, a detailed user evaluation of Caise-based tools is presented. In Section 7.4, the performance of the Caise framework is analysed.

7.1 Heuristic Evaluation

It is important to maintain a balance between development of CSE infrastructure and ongoing evaluation. If too long is spent performing detailed analyses of prototype systems, then not only are results likely to be of marginal use, but also the development process is likely to be delayed or misled. Conversely, to ignore evaluation is to risk failure because development is not guided and informed by empirical work.

المنارة للاستشارات
www.manaraa.com

Evaluating systems and techniques with typical user groups on realistic problems is difficult, time consuming and expensive. Various approaches have been developed.

User trials are well suited to evaluating HCI and usability aspects. These would be most useful towards the end of the development of Caise-based tools when a full range of industrial-strength tools is available. At that point it will be useful to quantify such factors as the relative merits of alternative feedback/feedthrough mechanisms and the balance between the benefits of awareness of others and the potential distractions from one's own tasks.

Field studies and case studies are long term undertakings. They are conducted in realistic industrial environments, in order to determine the domain-specific tasks which must be supported and to observe how particular systems are used in practice. As well as being expensive in terms of time and cost they also have difficulties such as provision of control groups; consequently, they are most useful when the systems to be evaluated are at a mature level. For Caise-based tools, group studies will be valuable to explore the patterns of collaboration amongst users and the effectiveness of individual techniques on particular categories of tasks.

In order to gain the most from costly evaluations it is important to be able to address the issues of assessing systems which are in early stages of development. This allows the developers to use results to improve the system rather than simply quantify its performance. A range of so-called 'discount' evaluation techniques have been developed to achieve this. These include heuristic evaluation [82, 80, 81], in which small groups of evaluators seek violations of a given set of heuristics. Results suggest that these techniques can be very effective in detecting faults, thereby enabling them to be corrected earlier in the development cycle.

This discussion of heuristic evaluations has appeared previously [29], and has been co-authored by Neville Churcher. These heuristics are not claimed to be complete and exhaustive; rather they are provided as motivational examples for software engineers when assessing the structure and rigour of CSE tools. These heuristics have been based upon our experiences when developing CSE tools, and further HCI/ethnographic field studies into the relevance and impact of these heuristics will be of benefit to CSE researchers.

المنارة للاستشارات
www.manaraa.com

7.1.1 Heuristic Evaluation of Groupware

A set of heuristics for evaluation of Groupware has recently been proposed [4, 5]. A summary is provided here, with a brief indication of how they relate to specific CAISE features.

CSCWi *Provide the Means for Intentional and Appropriate Verbal Communication.* CAISE provides text chat and an audio channel. External systems, such as telephone conferencing and web cams, may also be used.

CSCWii *Provide the Means for Intentional and Appropriate Gestural Communication.* CAISE provides a User Tree (see Figure 6.1) which indicates the location (scope) of each user. Individual tools may supplement this by implementing features such as telepointers.

CSCWiii *Provide Consequential Communication of an Individual's Embodiment.* This is currently implemented through the User Tree and Artifacts Pane.

CSCWiv *Provide Consequential Communication of Shared Artifacts (i.e. Artifact Feedthrough).* CAISE-based tools' buffers remain synchronised to reflect changes to the underlying artifacts. Individual tools may implement features such as colour-coding for the age of updates.

CSCWv *Provide Protection.* The default access policy in CAISE is to rely on social protocols, although collaborative undo is supported within the framework. Additional protection is provided by CAISE-based tools as required.

CSCWvi *Management of Tightly and Loosely-Coupled Collaboration.* The User Tree and Client Panel enable users to assess activities of interest. Feedback, tailored to reflect the users' interests, is used to alert users to potential conflicts.

المنارة للاستشارات
www.manaraa.com

Cscwvii *Allow People to Coordinate Their Actions.* Communication channels and other feedback mechanisms support coordination.

Cscwviii *Facilitate Finding Collaborators and Establishing Contact.* The Caise session management tools, such as the User Tree and Artifacts Pane, indicate which users, tools and artifacts are currently active.

It is useful to distinguish *taskwork*, task-specific actions, and *teamwork*, actions specific to group performance of tasks. Collaboration Usability Analysis (CUA) [88, 89] provides a technique for modelling domain-specific tasks in order to form a basis for heuristic evaluation.

7.1.2 Heuristics for CSE Evaluations

Heuristic evaluations are a valuable complement to other techniques for evaluating CSE systems, particularly for infrastructure and capability assessment—the areas in which experiment-driven feedback during development are most desired.

The heuristics and task modelling techniques proposed for CSCW [4, 5, 88, 89] are somewhat generic. They are extended for the purposed of CSE-based heuristic evaluations in two ways. Firstly, there is merit in establishing additional domain specific heuristics for CSE since this differs in many ways from the typical CSCW application area. Secondly, there is merit in the analysis and visualisation of Caise logs. This allows CSE researchers to mimic many of the beneficial aspects of case studies.

The current set of CSE-specific heuristics, to be considered alongside the generic CSCW heuristics discussed earlier, is presented here. A brief rationale for the inclusion of each heuristic is given, together with an indication of its relevance to the current Caise version.

CSEi *Support multiple views of artifacts.* A given Java class may be represented in different ways by individual client tools such as a text editor, folding editor, User Tree or UML class diagrammer. Changes made to the underlying artifact by any tool should be reflected appropriately in each view. Caise-based tools send updated artifacts

المنارة للاستشارات
www.manaraa.com

to the server. In return, they receive syntax trees corresponding to artifacts which have been updated by others.

CSEii *Support Degree of Interest based feedback/feedthrough.* Central to SE activities such as refactoring and comprehension is the notion of the neighbourhood (context) of a particular component or change (focus). The neighbourhood indicates the most relevant components to be taken into account from the viewpoint of the focus. For example, when modifying a method the neighbourhood might include the method itself, the methods it invokes and is invoked by, its host class and its parent class. This focus+context concept is familiar in visualisation. One common approach is the use of fisheye-view techniques [42, 97] to de-emphasise features not in the neighbourhood of the focus. When CAISE-based tools update artifacts, events are generated whose foci are located at the corresponding parse tree nodes. CAISE tailors feedback according to the neighbourhood of such nodes, determined by the semantic model, user preferences and specific client capabilities.

CSEiii *Support fine-grained integrity.* CSE requires more powerful approaches than simple CSCW applications in order to reflect the semantic and syntactic structures implied by the source code or other artifacts. CAISE uses parse trees as the basis for the semantic model it maintains.

CSEiv *Support multiple physical and logical granularities.* Physical granularity levels reflect physical partitioning (URL, directory, file, line, ...) while logical granularity levels (package, class, method, block, statement, expression, ...) reflect syntactic structure.

CSEv *Support deep syntactic- and semantic-based awareness and feedback.* The generic CSCW heuristics address issues such as notification of changes in the location of other users. In CSE, it is also important to be aware of changes at a semantic level (e.g. method foo() has been deleted from class Bar) or altered relationships involving components

المنارة للاستشارات
www.manaraa.com

and users (e.g. another user is editing a method which the method you are editing overrides). CAISE clients are notified of changes to the semantic model (including inferred relationships) and can reflect these as appropriate.

CSEvi *Support semantic relationships.* Updates to artifacts lead to *indirect*, and often subtle, changes in semantic relationships (extends, overloads, overrides, calls, uses, ...) which should be indicated to users.

CSEvii *Support private work and code integration.* Users can work against a snapshot of the project state and make experimental changes which will not be seen by others. In CAISE this simply involves client tools temporarily detaching from the server.

CSEviii *Support builds at different temporal granularities.* A rapidly evolving project, where developers make interleaved changes, could potentially spend much of its time in a broken state in which many components are unable to compile. CSE systems must accommodate artifacts that are temporarily un-parsable and projects that have unresolved code references. The CAISE framework propagates modification events to users directly accessing the same artifacts, ensuring that their views are synchronised at short timescales. The underlying semantic model is updated only when syntactic correctness is restored, so that on a coarser timescale, other users always build against a correct version.

Applying CSE Heuristics

CSE-specific heuristic evaluations, based on the combination of both sets of heuristics, leads to the identification and classification of problems and issues with CSE, CSE implementations and specific tools.

As an example, a problem identified in the current version of CAISE tools is a violation of heuristic CSCWiii: "The User Tree shows the user location, but does not indicate the transition from previous locations, making it hard to

المنارة للاستشارات
www.manaraa.com

decide what changes in location have occurred." Similarly, another problem identified as a violation of heuristic CSEv is: "Semantic feedback is delivered mainly via text messages. A metaphor more tightly coupled to the artifact representation would be more effective".

Deriving Data for Heuristic Evaluation

Empirical data from logs helps ensure that the sets of heuristics used are valid, representative and complete. Such data guides the ongoing process of refining sets of heuristics. In return, heuristics suggest patterns which should be observable in event logs.

Event logs can also reveal a great deal about collaboration patterns, system performance, task complexity and many other factors. In particular, they can indicate where refinement or extension of heuristics is appropriate, thereby improving the quality of subsequent heuristic evaluations.

In the next section, the use of visualisations based on analysis of Caise event logs is presented. Such visualisations can provide valuable information about patterns of collaboration, user activity profiles and sequences of operations in refactoring. This both complements and informs heuristic evaluations.

7.2 Visualisation Tools

The Caise framework incorporates an XML-based logging facility which unobtrusively records data about users and tools, events resulting from user activity and the artifacts affected. Server applications may process the logs in real time in order to obtain information, such as cumulative activity indicators, for propagation to users. Alternatively, logs may be processed off-line in order to perform detailed analyses.

The logging facility within Caise provides valuable insight as to how software projects develop over time. Not only is a fine-grained modification history available, information related to who made the changes and the impact of each individual change is also recorded. The ability to analyse a project's evolution at such a fine level of detail is new to the field of SE, both for collaborative and conventional tools.

المنارة للاستشارات
www.manaraa.com

7.2.1 The Visualisation Pipeline

The potential uses of even relatively unsophisticated visualisations in Groupware have been recognised [8]. Pipeline-based techniques developed for software and information visualisation [59, 23] are applicable to CAISE event log visualisation.

Figure 7.1 shows a typical log visualisation pipeline. Firstly, XSLT or other filters select and process the required data. In subsequent stages, layout tools produce 2D or 3D visualisations which are then rendered for user exploration. These may vary from spreadsheet graphics to virtual worlds.

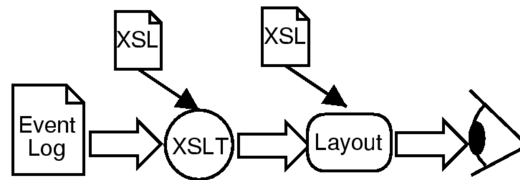


Figure 7.1: Visualising CAISE event log data.

The CAISE event logs conform to a DTD, `http://www.cosc.canterbury.ac.nz/dtd/CAISEEventLog.dtd`, enabling validation to be performed. Filters, typically implemented in XSLT, extract and format the data required for specific visualisations.

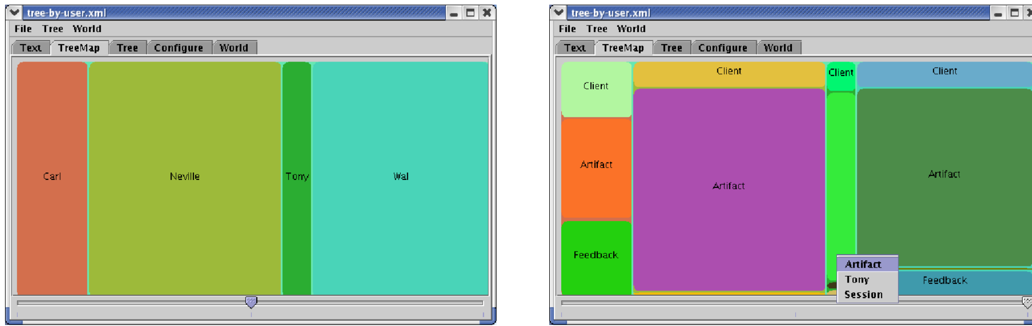
7.2.2 User Activity Visualisation

The visualisation process is illustrated with some analyses of log data from a CAISE-based development session of approximately 30 minutes, involving four users (including the author) working on a project consisting of ten Java classes.

Figure 7.2 shows two views of a visualisation based on treemaps [64]: in this case the pipeline ends with a treemap visualisation tool. The log is transformed into a tree representing a hierarchy of events structured as: session → user→ event type. Similarly, other structures (e.g. component → event type → user) may readily be obtained.

Figure 7.2(a) shows that two users, Neville and Wal, are responsible for most of the events generated in the session. Figure 7.2(b) provides additional

المنارة للاستشارات
www.manaraa.com



(a) Events originated by each user

(b) Events of each type originated by each user

Figure 7.2: Treemaps showing events in a CAISE session.

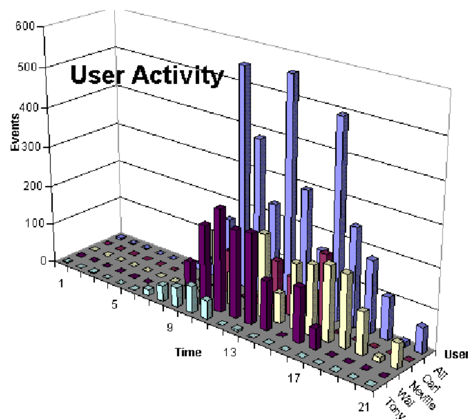
information about the proportion of events of each type resulting from individual users' actions. In this case, it can be seen that Carl and Wal have a greater proportion of feedback events than the other users.

From this information, it can be deduced that Tony was the least active user in this session (in fact he left before it ended); Neville and Wal were responsible for the bulk of the coding done during the session; Carl and Wal collaborated most closely (i.e. concurrently edited the same artifacts) while Neville and Wal worked more independently.

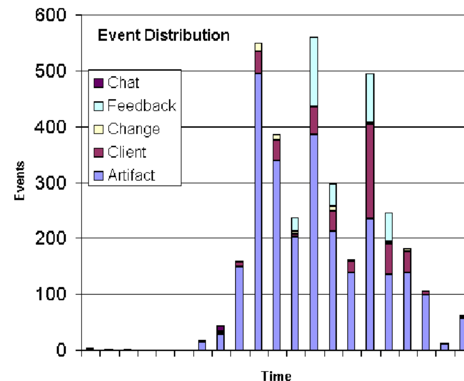
Figure 7.3 illustrates some temporal analysis options for the same data set. In this case the pipeline ends with a file readable by Excel. Figure 7.3(a) shows the number of events generated by the activity of each user during a 100 second interval as well as the overall totals. Peaks and lulls in activity can be seen clearly. In Figure 7.3(b) the events are broken down by type, irrespective of the user responsible for their generation.

Again patterns within the event log are evident. Most events in this session are Artifact events since significant text entry is occurring; Client events are mainly associated with location changes within artifacts; Change events arise from semantic changes such as altered inheritance relationships and are associated with Feedback events which alert other users to such changes.

المنارة للاستشارات
www.manaraa.com



(a) All events per user



(b) Events by type

Figure 7.3: Temporal analysis of user activity within a Caise-based project.

7.2.3 Artifact-Span Visualisation

Figure 7.4 shows the specific artifacts, in this case Java source files, modified by each user during the session. In this case, the pipeline produces a file for the popular `dot` layout tool [45]. In a dynamic version of this graph, edges are added and removed to reflect the current session state.

While the visualisation presented in Figure 7.4 appears somewhat trivial, it illustrates the potential of the Caise framework to supply fine-grained information related to the development of the software project over long periods of time—without impeding the participating developers. Without framework-based recording of developer activity, it is considerably more difficult to analyse and visualise the individual efforts of the development team.

7.3 User Evaluations

The heuristic evaluations for CSE presented in Section 7.1 assisted in keeping all CSE tools well designed during the prototype development of Caise. An empirical user evaluation, however, will help determine if developers are likely to embrace these new tools and provide objective measures related to SE tasks. Successful evaluations will also verify that the tools have a degree of robustness.

المنارة للاستشارات
www.manaraa.com

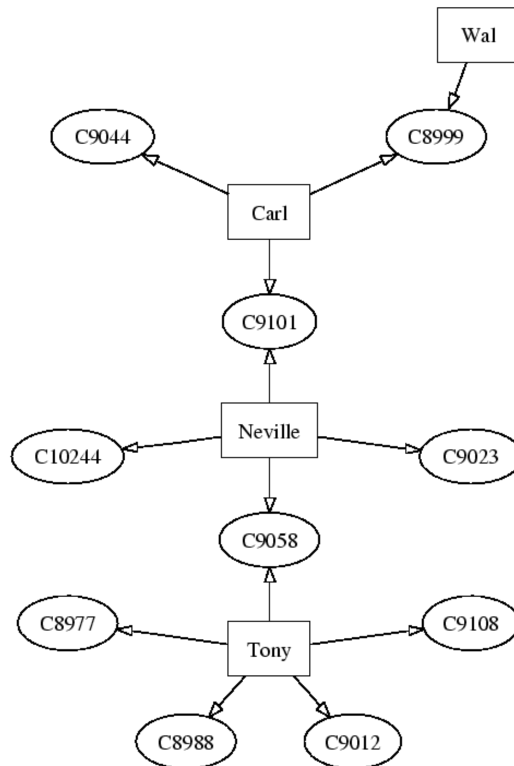


Figure 7.4: Artifacts accessed within a CAISE-based project.

The premise of the research in this thesis is that enabling more collaboration in SE through advanced tool support will in turn raise the restricted levels of communication and enable better development practices. To validate this premise, a comparison was made between CSE tools with their conventional counterparts, with the aim of showing scenarios where the collaborative tools are preferable.

The term *preferable*, however, is difficult to define objectively within empirical SE research. *Preferable* for CSE tools can have many meanings—faster task completion rates, ease of use, fewer bugs in resultant programs, encouragement of greater communication between programmers, greater program comprehension, greater awareness of other programmers' changes to name a few. Additionally, it is difficult to define the range of allowable values for external factors that affect the evaluation, such as size, type and difficulty of evaluation tasks, experience of participants, tools to be used within the control group, and features of the tools being evaluated.

www.manaraa.com

Accordingly, it is challenging to design a test that can evaluate all aspects of SE within a single context. Therefore, the evaluation presented in this section was limited to the objective measurement of task completion rates for mechanically scripted tasks between pairs of collaborating users, as well as gathering subjective measures such as user preferences. The hypothesis was that collaborative tools give task completion rates superior to those of their conventional counterparts for selected typical coding scenarios.

To the best of my knowledge, this is the first empirical evaluation of task completion rates and subjective measures for synchronous CSE tools. Other empirical studies have been performed previously that focus on tool support for collaborative software development [106, 77], but none have evaluated concurrent real time development tools.

7.3.1 Evaluation Method

A full and detailed description of the evaluation method is presented in Appendix F. In this section, only an overview of the evaluation method is given.

Experimental Design

A comparative randomised design, using one-way analysis of variance (ANOVA), was used to measure differences between SE tools operating in various modes of work. Pairs of co-located users working on adjacent workstations were the experimental units, with task completion rates the measured dependent variable. The independent variable was mode of tool operation. Two tool modes, or levels, were evaluated for the independent variable, consisting of collaborative and conventional tool support.

Each evaluation session was performed twice to compare task completion rates for each tool mode with a second factor—type of task. Two types of tasks were examined: between files and within files tasks. Between files tasks were such as renaming a method for one user while the second user made a new call to the method using the original name. Within file tasks were such as changing the structure of a control statement by one user while the second user changed a conditional within the statement.

المنارة للاستشارات
www.manaraa.com

Several control variables were identified, such as individual programmer ability, software development methodology followed, and difficulty of programming task given. As described in Appendix F, these control variables have been addressed by the experimental design.

The null hypothesis is that for each task type, no difference in task completion rates between the two modes of tools exists. The alternative hypothesis for each task type is that there is a difference in task completion rates. Rejection of the null hypothesis provides evidence that for a selection of simple but common SE scenarios, one type of tool is preferable over the over in terms of developer performance.

The interaction between type of task and tool mode was not assessed in this evaluation. Additionally, differences in task completion rates for the two types of tasks within each tool mode were not analysed, as this is of no immediate interest to the current research.

Experimental Environment

All evaluation tasks were based on a simple 1000 line graphical Java application. The program consisted of eleven classes within a package that displayed several animated sequences. While the program was relatively small, it contained some complex design idioms such as behavioural, creational and structural design patterns, use of collections classes, graphics code and event-based actions. It was trivial for participants to assert that their changes had taken effect; the program at startup would show the animations in their current state which could be immediately verified for correctness. A screen shot of the program in a typical working state is presented in Figure 7.5.

All tasks were performed primarily in the Java editor. The UML class diagrammer was available for visualisation of the changing program structure and for user presence awareness. In collaborative mode, the tasks could be performed using the real time file sharing support of the tools. In conventional tool mode, the participants were able to share and synchronise their files through the inbuilt code repository support. The code repository interface was minimal to avoid confounding the experiment, as explained in Section F.3.4.

المنارة للاستشارات
www.manaraa.com

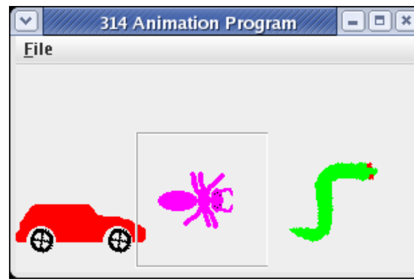


Figure 7.5: The graphical interface of the program under modification during the evaluation sessions.

Although two different modes of work are being compared—conventional versus collaborative—the comparison is fair. Code repositories are the mainstream technology for collaborative SE, the only other commercial option today is pair-programming with a shared keyboard and display.

For both types of tasks, there was a deliberate and unavoidable conflict between the instructions for both participants. The coding conflict was introduced to replicate the typical SE scenario of conflicting changes between a pair of developers. To eliminate any variance caused by differing programmer abilities between groups as team members attempt to resolve the coding conflict, all steps within each task were scripted for both participating users per session. Users were instructed to follow the scripted programming steps without deviation, regardless of their own opinions on how to complete the overall task or how to resolve the inevitable conflict once discovered.

Each task was timed, and participants were instructed to work as fast as possible without rushing; this ensured that the participants were focused on completion rates rather than collaboration. To complete the tasks, however, a degree of collaboration was inevitable, which suggests that the tasks were still realistic. The participants had a brief reading period before being timed, where they could clarify any questions related to the task. The participants were not permitted to discuss the task with each other at this stage, however. They could only communicate with each other when completing the task, both face-to-face and by observing the feedback from the tools.

When the inevitable conflict within each task was discovered, the partic-

المنارة للاستشارات
www.manaraa.com

ipants were then instructed to access an answer sheet which contained the predetermined resolution for the given task. Under normal conditions programmers would discuss and resolve the conflict themselves, but this factor had the potential to confound the experiment. Timing would stop as soon as the problem was corrected, the code compiled and synchronised, and the program was demonstrated to execute correctly on the workstations of both users.

Upon completion of each task a survey was given to each participant to answer in private. This allowed a comparison of each participant's perceived level of frustration, success and effort for each tool mode and task type. Finally, another survey was completed at the end of each evaluation session, providing a subjective summary of each user's preferences and comments for later comparison.

7.3.2 Evaluation Results

This section provides details on the findings of the user evaluation. The results are discussed further in Section 7.3.4. In Appendix F, a detailed discussion is provided on how the statistics in this section were derived and what their meanings and implications are.

Task Completion Times

The task completion times for the tools in collaborative mode were at least twice as fast as the times recorded for the tools in conventional mode. The comparative differences are presented in Figures 7.6 and 7.7. Error bars show the mean ± one standard error.

For within file tasks the difference between tool modes was highly significant ($F_{1,10} = 38.3$, $p<0.01$), as were the between file differences ($F_{1,10} = 34.2$, $p<0.01$). These significance levels give us confidence that the results were not obtained by chance; statistically, these results are expected for 99.9% of trials that repeat this experiment.

المنارة للاستشارات
www.manaraa.com

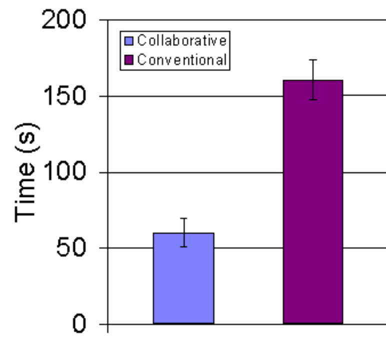


Figure 7.6: Mean task completion times for within file tasks.

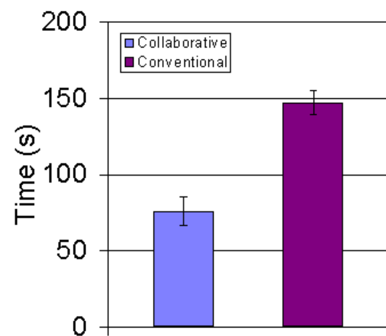


Figure 7.7: Mean task completion times for between file tasks.

Subjective Assessment

Table 7.3.2 presents the findings of the survey given at the end of each task within the evaluation sessions. The survey is based on the NASA Task Load Index [52] with a 20 point Likert scale. From the table it is apparent that for both task modes, participants felt strongly that they understood the changes of others better, and it was markedly easier to control source files using the tools in collaborative mode.

For perceived frustration, perceived effort and awareness of local changes, there was a statistically significant difference between the mean response in one of the two task modes in favour of collaborative mode. For the remaining task mode in each survey question, the difference was still favourable

المنارة للاستشارات
www.manaraa.com

	NASA Task Load Index: Within, Between					
	Understanding own changes	Understanding others' changes	Ease of File Control	Perceived Effort	Perceived Success	Perceived Frustration
Collaborative:						
Mean	14.7, 12.1	18.8, 9.2	16.3, 15.9	3.9, 2.9	17.9, 16.3	3.7, 4.3
(s.d.)	(4.9, 4.2)	(1.4, 5.9)	(3.5, 3.5)	(3.5, 2.5)	(2.2, 2.5)	(2.4, 2.8)
Conventional:						
Mean	9.0, 8.8	4.5, 1.8	8.4, 7.6	5.3, 7.5	15.5, 14.4	6.1, 8.3
(s.d.)	(4.5, 6.0)	(4.3, 1.4)	(4.3, 5.6)	(3.8, 5.5)	(3.6, 4.3)	(4.0, 5.7)
***<.01, *<.05	***,–	***,***	***,***	–,*	–,–	–,*

Table 7.1: Summary of the subjective measures for tasks: NASA-TLX workload ratings. Possible values range from 1 (low) to 20 (high).

towards collaborative mode, but the difference was not statistically significant. For the perceived success survey question, neither task mode gave a significantly difference in mean response, although the participants again showed a lenience towards the collaborative version of the tools.

User Preferences

Table 7.3.2 presents the findings of the survey given at the end of each evaluation session. This survey focused on general user preferences using a 20 point Likert scale. The questions within this survey are also presented in Table 7.3.2.

Order	Question	Response: mean (s.d.)
1	In a collaborative, distributed setting, how useful do you think this type of system will be?	15.7 (2.1)
2	In a collaborative, co-located setting, how useful do you think this type of system will be?	15.8 (1.9)
3	How much does it help to have the source code shared and managed for you?	16.4 (2.2)
4	How often would you like to work on collaborative tasks with a system such as this (a system that updates and shares source files in real time)?	14.3 (2.2)
5	How useful did you find the ability to know what the current global state of the project is?	14.8 (3.5)
6	How adequately was the awareness support provided (such as user location feedback)?	13.0 (4.1)

Table 7.2: Summary of the subjective measures for overall preference. Possible values range from 1 (low) to 20 (high).

المنارة للاستشارات
www.manaraa.com

The results of the user preferences survey were encouraging—all responses ranged from positive to extremely positive. The participants foresee the collaborative tools as useful in both co-located and distributed settings, they find the real time synchronisation of code helpful, the feedback support was also perceived as useful, and they claim they would use CSE tools such as those used in the evaluation often if made available.

User Comments

Examples of recurring comments made during and after the trials are listed in Table 7.3.2. Of the positive comments a conclusion can be drawn that all users enjoyed using the system, and they claim that they would use it for most situations given the opportunity. They also stated that they liked having the source code managed for most tasks. These comments are corroborated by the results of the user preferences survey reported in Section 7.3.2.

Type	Comment
✓	"The system made coding more enjoyable."
✓	"I liked the concept of real time development."
✓	"The collaborative [user] tree was really helpful."
✗	"The [editor] lag was a bit annoying."
✗	"A private work area is needed for offline [development] spikes."
✗	"The editor needs tele-scrollbars to give a better indication of where other users are within the same file."

Table 7.3: Post-session user comments.

Of comments to help improve the system, a private work facility was suggested if the tools are to be used in a commercial setting. The remaining comments for improvement were all related to usability issues that will be addressed in the next development phase.

7.3.3 Threats to Validity

While the CVS interface was not as complex as those typically supported in IDEs, the core facilities of check-out, check-in and merge were present within the control version of the trial tool. These facilities were well-aligned with

المنارة للاستشارات
www.manaraa.com

the experiment, which specifically aimed to compare real time shared code editing to the copy/modify/merge idiom of code repository systems.

For a fair user evaluation, the experiments had to be realistic yet measurable. If the experiment tasks were too sterile then there was the risk of having results that are valid but not genuinely useful in a global context. While the evaluation tasks were required to be simple to enable them to be repeatable and free from confounding factors, they still represented an approximation of tasks and conflicts that are likely to be encountered in everyday SE.

Another potential threat to validity is that of using students as evaluation subjects. The students selected for this experiment, however, all had strong interests in software engineering, and were conversant with SE aspects such as design patterns, test-driven design, software development methodologies and UML. Therefore, I feel that the evaluation subjects were representative of candidate users of CSE tools.

All other confounding factors that could cause a threat to the validity of this evaluation have been addressed by the experimental design. Full details of the experimental design are given in Appendix F.3.

7.3.4 Discussion

This experiment focused on pairs of collaborating users. While no inference can be confidently made as to how the CSE tools will perform when used by large groups of developers working concurrently, the experimental design does give us considerable insight as to how small groups of developers will react and perform when using real time CSE tools.

The results obtained for task completion rates and subjective measures were surprisingly good considering that no attention had been paid to making the tools particularly user friendly or refined. While it is reasonable to assume that some difference would exist between the two tool modes in favour of collaboration, it was surprising that the differences were so large. More pleasing, however, were the subjective results which showed that users liked using the system and agreed with the perceived benefits to SE given in this thesis. It was always a concern that even though the users could perform the tasks faster, they did not like using the tools in collaborative mode.

المنارة للاستشارات
www.manaraa.com

While the evaluation tasks involved at least a degree of collaboration between users, the tasks were not designed specifically in favour of a highly collaborative approach. Therefore, for tasks that are highly collaborative, such as debugging or demonstrating new ideas, it is reasonable to believe that the tools in collaborative mode would perform even better than in this experiment. Similarly, as the users only had ten minutes worth of training in collaborative tool mode, it is possible that the collaborative features of the tools were not used to their full potential. Given more experienced users, it is likely that the task completion rates could have been improved upon, and the feedback on the collaborative mode of work might have been even more positive.

When referring to the data presented in Figures 7.6 and 7.7, there was a considerably larger gain for collaborative within files tasks than collaborative between file tasks. The likely explanation for this is that it is not always possible to completely avoid transactional conflicts during between files tasks as it is to avoid merge conflicts during within file tasks. Programmers working without consideration for other users still have the potential to create transactional errors during between files tasks, which ultimately must be corrected. Regardless of the relative difference between the two task types, between files tasks are still a lot faster in collaborative mode than conventional mode because the error is detected as it is made, instead of waiting for the results of a file merge and project rebuild.

An interesting observation during the experiments was that when participants did not stop and talk with each other in collaborative mode for within files tasks, they still managed to accomplish their code changes without noticeable hindrance. They simply engaged in a brief 'editing war', where even though their changes were being interrupted, both users very soon had their code changes in place. Under normal circumstances, users are likely to pause development activity and discuss the collaborative edits that occurred in the same region of code. Participants in this experiment, however, were highly task oriented due to the nature of the evaluation.

المنارة للاستشارات
www.manaraa.com

Summary

Through this user evaluation, example coding scenarios have been given where the CAISE-based CSE tools not only outperform their conventional counterparts, but users prefer using them, their perceived success is higher, and their perceived effort and frustration levels are lower. The results strongly suggest that collaborative tools such as text editors can improve the productivity of software development. Subjective results also suggest that providing users with a constantly updated global project state appears to help developers rather than hinder.

The results of this evaluation give credibility to the assumption that computer mediated support for CSE can provide real benefits to software engineers. It has been demonstrated that the CAISE-based CSE tools stand up to testing with users that have had no previous exposure or experience to them, even when completing considerably comprehensive tasks within a non-trivial application. From the results of this evaluation, I am strongly encouraged to continue with further research and development of tools for CSE.

7.4 Framework Performance

Another important consideration when discussing the design and use of tools for CSE is that of performance. The performance of the tools must be satisfactory, and there should be no theoretical limitations of the framework that will prevent the tools from being useful in realistic environments. While the core response speeds and resource usage of CAISE and its supporting tools have proved acceptable over a long period of subjective testing and user evaluations, it is important to note the effects of code size and number of concurrent developers on server memory load and tool response times.

7.4.1 Memory Load

To provide features such as impact reports and user proximity feedback, the CAISE server maintains a semantic model of the software within the project. An immediate concern is that of memory usage; if a large amount of memory is required for each line of code added to the semantic model, projects of a

المنارة للاستشارات
www.manaraa.com

realistically large size might be beyond the scope of the CAISE framework.

Figure 7.8 presents the amount of memory used per line of code across a range of CAISE projects. For any CAISE-based project, the server first loads in all packages, classes, interfaces and methods directly accessible from any Java source file. This brings the initial project semantic model size to around 60 MB. From this point onwards, however, most of the components that the modelled software rely upon are now loaded, and the project semantic model size increases only linearly in relation to the number of classes and methods declared in each source file. Each subsequent line of code requires approximately only one kilobyte of server memory.

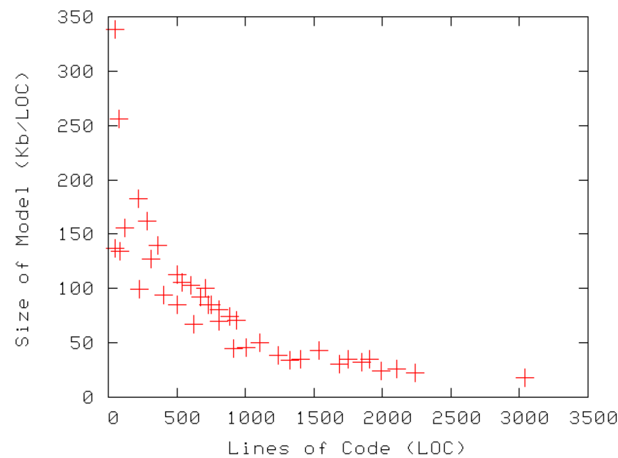


Figure 7.8: Lines of code versus server memory usage.

For large software projects where there can be potentially millions of lines of code within a single version, an alternative to an in-memory semantic model might be required. In commercial settings, it is likely that specialised hardware can support multiple gigabytes of memory. In other situations where mass memory capabilities are not available, the CAISE framework can easily be extended to incorporate an OO database for semantic models of potentially any size.

While the memory requirements for a CAISE-based project may seem significant, it is important to note that no other demands are placed on memory resources throughout the entire development environment. Unlike

المنارة للاستشارات
www.manaraa.com

other architectures including IDEs, each CSE tool can rely on the Caise server for all parsing, analysing and semantic modelling of the software; tools themselves do not necessarily have to store a replica semantic model.

7.4.2 Network Load

The design of the framework ensures that network loads are as low as possible, and analysis of traffic verifies that for small user groups, no considerable strain is placed on a 100 Mbps Ethernet local area network. Even as the number of concurrent users increases to that of large development teams, today's networks are capable of accommodating the load.

When testing on wide area networks, the data throughput requirements are low enough for clients to be connected to the server from dial-up networks, but the latency can cause edit delays of up to several seconds. To support low speed wide area network connections, an alternative distributed system might be necessary where the anticipated results of modification requests are immediately represented in the originating tool's display. In this case, a synchronisation routine will be required to run in a separate thread to resolve any modification discrepancies between tools.

At present, fault tolerance within Caise has not been addressed. User trials and experimentation have been limited to local networks, where error rates are low and are readily addressed by underlying communication protocols. For high-latency, high error network contexts such as global software development, techniques for fault tolerance may need to be identified.

Performance Details

When performing packet captures to analyse network data, a compressed semantic model of software containing a small project is approximately 150 Kb in size, or 100 Ethernet frames. This is not a significant amount of data to transmit given the operational capabilities of any modern network.

Caise-based events, such as keystrokes and feedback messages, typically use approximately 800 bytes of data when broadcasted. As a single Ethernet frame can carry up to 1500 bytes of data, the transmission of Caise events should not have any significant event on an existing network.

المنارة للاستشارات
www.manaraa.com

API calls were also analysed. Each call from a CSE tool to the Caise server was measured as a 50 byte packet. Responses from the server were the same size. This implies that even a large number of consecutive requests to the Caise server from CSE tools will not cause any serious networking issues.

7.4.3 Response Times

Performance measurements illustrate that as the number of users and the size of the project semantic model increases, response times will remain stable. The direct impact of increased numbers of concurrent users within a Caise project has been observed to be negligible; the number of connected users or opened files does not have a noticeable effect on server memory usage or response times. If all users are highly active at the same time the server response times will slow down temporarily, but in reality this is an unlikely scenario.

Even if a project has a large semantic model, this does not necessarily affect the response times of the server. Most operations such as adding a new method to a class or querying the semantic model for a specific relationship only require the traversal of a fixed subset of the entire semantic model space. Therefore, even as the semantic model grows in size, the response times will stay approximately constant.

Performance Details

Using the hardware described in Section F.3.3, code edits take approximately 100 milliseconds from the originating keystroke to being updated on all remote views. Semantic model changes, normally invoked through the class diagrammer, take approximately one second to be updated on all views. The delay is caused by the server processing the request and modifying the underlying semantic model, which text editors will not directly encounter.

To traverse the entire semantic model of the Animation Application program through a server application, 516 classes are encountered. This takes approximately 3.5 seconds, including the time to draw the results to a text pane.

المنارة للاستشارات
www.manaraa.com

7.4.4 Feedback Information versus Number of Users

Performance measurements presented in this section indicate that the CAISE framework and associated CSE tools can scale to moderately sized code-bases and development teams. With the addition of server-level hardware or a semantic model caching mechanism, the CAISE framework will be able to scale to large code-bases and number of users, as both of these aspects introduce only a linear increase in server resources. As the number of concurrent users increases, however, the amount of generated feedback may increase exponentially, as each user has the potential to work on regions of the software that are related to all other users, either directly or indirectly.

While the generation of large volumes of feedback information will have some impact on the CAISE server's processing load, the scalability issue most significant is that of management of feedback information per CSE tool from a user interface perspective. To address the possibility of feedback saturation, tailoring of feedback is discussed in Section 8.2.1.

Summary

Very few CSE research projects have conducted evaluations of any type for the resultant tools and technologies produced. As far as I am aware, no formal user evaluations of synchronous CSE tools have been carried out prior to the CAISE research project. As the focus for the CAISE framework was to produce practical solutions to current SE tool limitations, constant evaluation of the CAISE framework and tools has been central to the research process.

In this chapter the issue of tool suitability has been examined. The various forms of evaluations indicate that the CAISE-based CSE tools presented in this thesis are useful for the support of CSE. The performance of the CAISE framework has also been discussed and shown to be suitable for most development purposes within small teams and code-bases.

In Chapter 8, methods to expand the scope of the CAISE framework are presented. Open research problems are also discussed.

المنارة للاستشارات
www.manaraa.com

Chapter VIII

Caise in an Industrial Context

In this thesis, the Caise framework has been presented, discussed and evaluated. The work in this thesis demonstrates that the Caise approach is appropriate for the support of CSE, and that useful CSE tools can be constructed.

In this chapter, a discussion is given on how the scope of the Caise framework can be expanded for industrial use. In Section 8.1, a discussion is presented on how large groups of users can be accommodated within CSE systems. Areas of enhancement, in order for the benefits of Caise to be fully realised within an industrial setting, are listed in Section 8.3.

8.1 Managing Groups and Individuals

In this section, the key scalability issues that challenge CSE are presented. Techniques for addressing scalability are listed for when the limits of CSE tools are exceeded. These techniques are available to all CSE systems, not just the Caise framework.

For a small group of developers, a small set of source files, and a well defined SE process, real time CSE tools are likely to be readily suitable. However, as the number of developers increases, the project size in terms of lines of code grows, or the developers are adverse to continual collaboration, CSE tools may not be as suitable as some conventional tools that permit long periods of uninterrupted private work.

8.1.1 Working from a Source Code Repository

Figure 8.1 presents the typical version history for a software project being developed with real time synchronous CSE tools. As the entire project is

المنارة للاستشارات

www.manaraa.com

shared, concurrent modification of the project alters the single project trunk, and checkpoints are made only to provide an offline revision history. Individual source files and other artifacts may evolve over time but only one project branch ever exists.

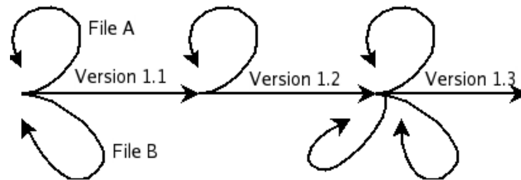


Figure 8.1: A typical revision trunk for a collaborative software project.

The jitter of concurrent changes within the same region of code may appear mildly distracting when compared to working in isolation, but this is offset by the added awareness of the actions of others, and the avoidance of costly merge processes.

From a viewpoint of real time CSE, source code repository systems may at first appear antithetical to CSE tools; the purpose of tool support for CSE is to enable developers to work together, not to partition themselves. While this argument is certainly true for small development groups, within the realm of open-source software development the use of source code repository systems is essential and unavoidable.

Users of real time systems are still able to work collaboratively within an environment controlled by a code repository. This is achieved by forming a group of collaborating users, and working collaboratively through CSE tools within this group. The set of source files within the shared project will be based from the latest version from the code repository, and this collaborative group will be required to periodically re-synchronise their code base with that of the central repository.

This approach still subjects the collaborative group to the same problems that the code repository users face: merge and transactional conflicts upon re-synchronisation with the code repository. However, the collaborative group benefits from having the ability to work together within the group. Additionally, if the area that the collaborative group is working on within

المنارة للاستشارات
www.manaraa.com

the project is loosely coupled from the rest of the project, merge conflicts should largely be avoided, and transactional conflicts are also likely to be low.

The success of this approach depends on the number of collaborative groups within the entire project, the size of each collaborative group, the ratio of collaborative to conventional developers, the degree of coupling between packages within the project, and the development approach of the programmers.

A CAISE project can operate on a working copy of source files checked out from a source code repository located elsewhere such as SourceForge [84]. In fact, the CAISE-based Java text editor, presented in Section 6.3.1, implements support for CVS code repositories [9]. Using this code editor tool, a developer within a collaborative group can upload the CAISE-based artifacts to a CVS server, and if required, refresh the CAISE-based project's artifacts with the latest version from the repository as well.

The Java text editor presented in Section 6.3.1 employs a *syntax directed* form of support for code repositories, which is illustrated in Figure 8.2. CVS is used as the underlying code repository system, but as CVS can be troublesome for new users to master, only valid and meaningful CVS operations for the given state of the repository are available.

(a) Download (b) Upload

Figure 8.2: A syntax-directed code repository interface.

To explain the syntax directed code repository interface further, if the source files within the current CAISE project are up-to-date with the code repository, no options are available from the repository menu visible in Figure 8.2. If the code repository has newer versions of any file compared to the CAISE tool's copy, the only code repository option available, as illustrated in Figure 8.2(a), is to download the new files to the Java editor. An automatic merge of the old and new files within the editor is then performed. If the

المنارة للاستشارات
www.manaraa.com

Java editor has a newer version of any file than the code repository, the only repository option available, as illustrated in Figure 8.2(b), is to upload the tool's files back to the code repository.

The repository interface provides an easy mechanism for using a source code control system, and avoids problems commonly associated with code repositories such as forgetting to upload all modified files, forgetting to download more recent files and losing synchronisation with the rest of the development team. A simple and effective repository interface was essential for the user evaluations presented in Section 7.3.1. During this evaluation, many users commented on how intuitive and easy-to-use the code repository interface was, and they would like the same interface on all of their usual SE tools.

It is important to note that CVS support is implemented in the CAISE tools; the CAISE server requires no knowledge of source code repositories in order for them to be used by CAISE-based tools. The CAISE server simply treats the live CAISE-based artifacts as the only version that exists, even if these files were originally downloaded from a source code repository. In the case of working with source files from a source code repository, the tools at startup typically download the latest version of the files from the code repository and then update the CAISE server with these new files.

8.1.2 Partitioning of Projects

In a well designed software project, there are likely to be separate areas for developers to focus on, and a natural partitioning of roles can take place. A simplistic example of such a project is presented in Figure 8.3. In this example, one group of users can work on the GUI, another group can work on the database, and very few conflicts are likely to occur between groups. It is important to note, however, that even in a well partitioned and highly modularised project, there will be a multitude of code relationships and dependencies between packages and classes.

Within professional development groups, well partitioned projects and structured development approaches are likely. In this situation, where relatively few changes within a partition should affect the development efforts of

المنارة للاستشارات
www.manaraa.com

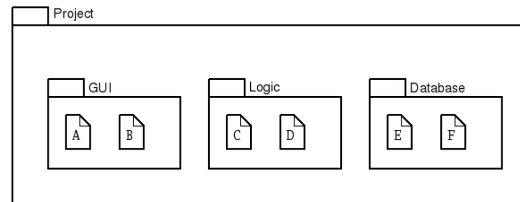


Figure 8.3: A simplistic example of a well partitioned software project.

those working on other areas of code, development *crosstalk* is likely to be at an acceptably low level. In this case, the use of real time CSE tools is also suitable.

In some projects, however, even if they are well partitioned, it is possible that the developers will prefer no crosstalk from programmer activity within other partitions of the project. In this situation, it is still possible to accommodate collaboration within each development partition by using the code repository mechanism discussed in Section 8.1.1.

There are three main types of configurations available for projects that are developed by collaborative groups, as illustrated in Figure 8.4. These configurations are:

Individual Each user works individually, each with their own local copies of source files, using a code repository to integrate changes

Partitioned The developers are split into sub-groups, and developers work collaboratively within each sub-group. A code repository is used to merge files between sub-groups

Global All developers work together in real time using CSE tools on a shared project semantic model. A code repository system is not required for file sharing

By partitioning a group of developers within a project into sub-groups, crosstalk between groups is eliminated. The trade-off, of course, is that communication between groups is likely to be reduced and a synchronisation process must take place at regular intervals between groups. However, if the

المنارة للاستشارات
www.manaraa.com

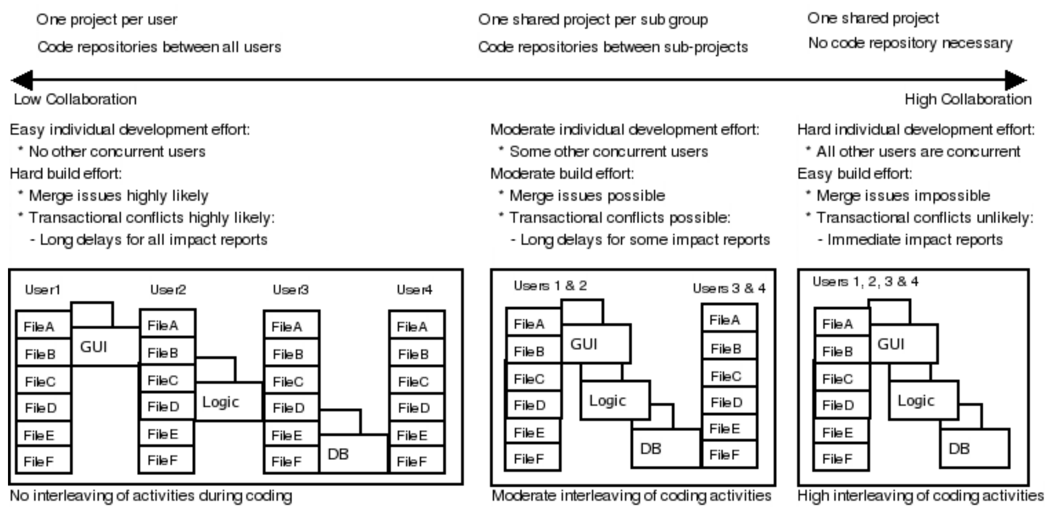


Figure 8.4: Various configurations for group work using CSE tools.

project is well designed and the developers use a structured SE approach, merge and transactional conflicts are likely to be infrequent.

The partitioned approach is illustrated in the middle segment of Figure 8.4. Crosstalk is likely to be less than in full collaborative mode, and merge conflicts and transactional conflicts are also likely to be less frequent than in the conventional, code repository mode. It should be noted, however, that this approach should only be used when groups of developers intentionally wish to separate themselves.

This choice of group configuration depends on how well the project is partitioned, how many developers are likely to work within each partition, and the programming methodology employed.

As illustrated in Figure 8.4, the conventional development end of the spectrum allows each user to have his or her own code base to work on. Usually, each programmer will try to modify only the subset of files within his or her current area of focus [51]. In this configuration, the individual programming effort is relatively easy, but transactional and merge conflicts are likely.

At the other end of the spectrum, using a single shared project negates all use of code repositories. This implies that while higher levels of develop-

المنارة للاستشارات
www.manaraa.com

ment crosstalk are possible, transactional conflicts are less likely and merge conflicts are completely avoided. In this thesis, it is argued that for most small and medium sized development groups it is better to work as one collaborating team—development jitter during spikes of activity is preferable to ongoing conflicts and reduced programmer communication between developers.

8.1.3 Compilation Crosstalk

Unexpected real time code modifications by other users, while surprising, do not significantly degrade a developers ability to work within a collaborative setting. Evidence of this was given in Section 7.3. If one developer is working on the same line of code as another developer, it is likely to be beneficial if both parties pause and discuss the current activities, although programmers may choose to ignore the presence of others and carry on development. A major problem with real time development, however, is that of compiling code during a time of concurrent development activity. This problem is hereby termed *compilation crosstalk*.

In a conventional development setting, the problem of compilation crosstalk does not occur. As each developer's code base remains isolated from the central repository and other developers' caches, system compilation can be performed without hindrance. The problem with conventional development, however, is that developers will not be made aware of concurrent modifications to the code base by other users; modifications which have the potential to significantly alter the semantics of the software project.

In both collaborative and conventional settings, if one developer makes a modification that is relatively isolated from all other areas of a program, all other developers should not necessarily be placed in a position where they are prevented from compiling. For CSE tools, however, if the first developer has not completed their changes, or their changes are syntactically or semantically incorrect, the project will fail to build even for other users, as the entire project is shared in real time. To resolve this problem, the project Build Pane has been refined with a special *collaborative scope* feature, which is presented in Figure 6.4.

المنارة للاستشارات
www.manaraa.com

The collaborative scope facility within the project Build Pane allows compilation to take place from within three different modes: *current*, *last parseable* and *last buildable*. In current mode, the pane attempts to build the latest version of the code, which will fail if any recent remote changes have broken the build. In last parseable mode, the build only takes into account the last syntactically correct version of each file. This way, if a remote programmer is currently editing a file, his or her changes will only take effect once the code is properly formed. In last buildable mode, the panel will produce an executable based on the last version of the program that has no build errors. The different types of collaborative scope within the project tools panel are depicted in Figure 8.5.

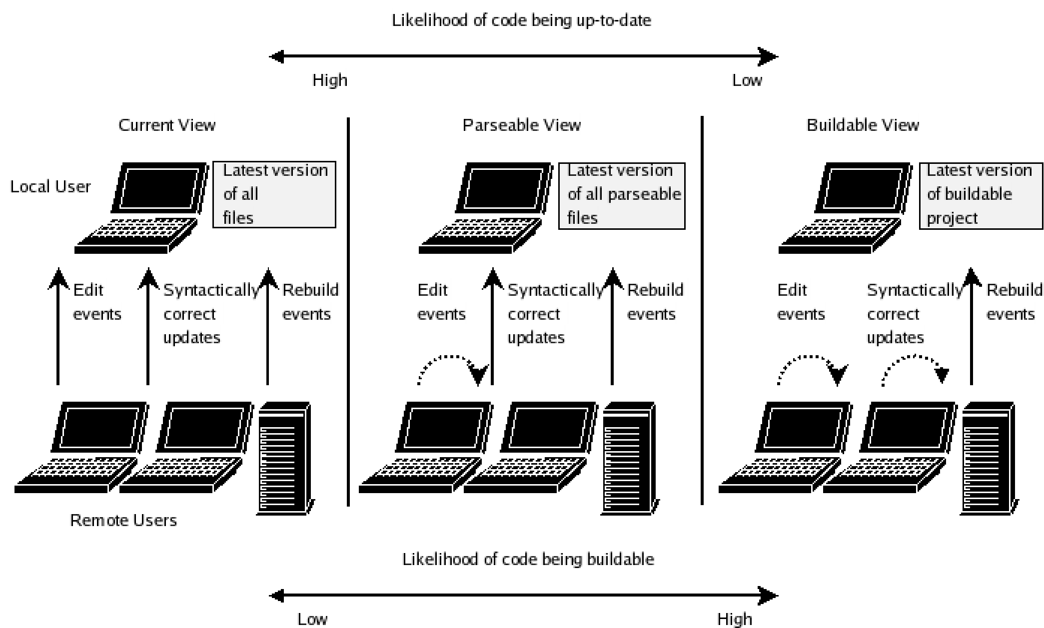


Figure 8.5: The various modes of collaborative view when compiling from within a CAISE tool.

From the previous discussion of Figure 5.7, the three conceptual layers of the CAISE framework are: CSCW, CSE, and SE. These three layers are also mirrored in the Build Pane's modes of collaborative scope. The current scope represents and immediate or synchronous view of the artifacts as they are edited by real time tools (the CSCW layer). The last parseable mode of

المنارة للاستشارات
www.manaraa.com

collaborative scope represents the server's view of all syntactically complete artifacts (the CSE layer). Finally, the last buildable mode of collaborative scope represents the latest version of the project that successfully builds (the SE layer).

The collaborative scope facility has proved to be a particularly useful feature for CSE tools. The collaborative scope facility for avoiding compilation crosstalk may provide an alternative to partitioning a group of users when the activity of remote users makes it difficult to compile the shared set of a project's source files. As the collaborative scope facility has the potential to be a general strategy for all CSE tools, it will be an ideal candidate pattern of CSE if widespread use eventuates.

Project Build Likelihood

Related to compilation crosstalk, Figure 8.6 presents the likelihood of build failures for conventional and collaborative modes of work. This data in this figure is anecdotal, based on observations made during the user evaluations presented in Section 7.3: the software project is likely to be in a buildable state more often when users are given immediate change impact information. CSE tools are self regulating where it is always likely that the project will build with minimal effort—presuming that developers always take appropriate action when feedback related to broken units of code is received.

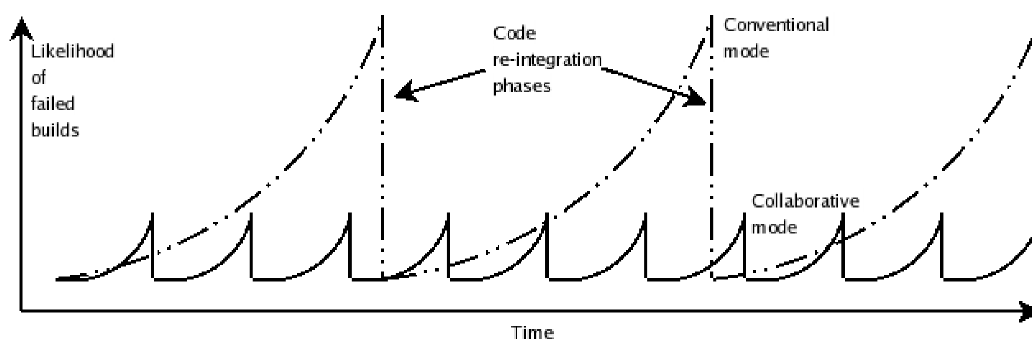


Figure 8.6: The likelihood of build failures: collaborative versus conventional modes of work.

Conventional modes of work, however, are subject to the use of the given

المنارة للاستشارات
www.manaraa.com

code repository system, which has the potential to involve large delays and development skews before getting the main branch into a buildable state again. This increasing effort of code integration is indicated by the spikes sketched in Figure 8.6. Therefore, even though it might take more initial coordination and collaboration to develop software using real time CSE tools, the likelihood of being able to build the project at any given point of time is relatively high.

8.1.4 Private Work

The main focus of CSE systems is to enable developers to work together. The concept of private work, where developers modify a copy of a code base independent of other concurrent developer activities, may appear antithetical to the principles of CSE, but it is on some occasions essential in real-world SE scenarios. This has been discussed as a pattern of CSE in Section 3.4.2.

If a given developer deems it essential to work in isolation for a considerable amount of time, he or she can work on an alternative branch of the code base using a code repository system, and merge the changes back into the main CSE project upon completion. Following the conventions of code repository-based SE, it is good practice to discourage all other users from modifying their version of the files checked-out for private use during this period.

All CSE tools have the potential to implement a private work mode, where the changes of others are prevented from being propagated to the tools of private developers. Unfortunately, such a tool mode has the potential to attract merge conflicts upon code integration with the main CSE project, and draws away from the ideals of collaborative work. Therefore, modes of private work within the Caise framework and associated tools are discouraged, but are trivial to implement if the group culture requires it.

8.2 Large Software Projects

The Caise framework is best suited to a small group of developers who wish to work collaboratively and in close contact on an entire software project. For the development of projects where large numbers of people are working

المنارة للاستشارات
www.manaraa.com

on many artifacts, there are no theoretical, technical or practical reasons why the Caise framework can not be used, as long as a structured approach to software development is taken.

It is unlikely that in a well-planned software development project, numerous people will work on the same set of heavily related files [16, 51]. Developers will usually work within separate areas of code, especially when the number of developers is large and several tasks can be performed concurrently. In these settings, the Caise framework and associated tools are expected to work well.

In the case, however, of large open source development projects, where coding efforts are not necessarily planned and coordinated in advance, and heavy moderation takes place, the Caise framework is not as well suited. The support of large development teams is a challenging aspect for all research towards CSE tools and technologies.

8.2.1 Tailoring Feedback

As discussed in Section 7.4.4, feedback information will become the dominant feature of Caise-based tools when numbers of concurrent users become large. To this end, it will become viable to introduce several mechanisms which tailor feedback per user. Failure to control the amount of feedback information generated and displayed may cause information overload for end users, and important feedback information may risk being ignored if it generated too frequently during development tasks that require a significant amount of concentration.

Two initial approaches to tailoring feedback can be easily made within the current version of the Caise framework and associated tools. Firstly, feedback plugins within the Caise framework could allow the registration of user-configured filters, allowing feedback information to be generated only for the types of feedback that each end user is interested in. Examples of semantic model feedback information that could be explicitly requested by end users include subclass/superclass connection, method callee/caller connection, and type declaration/association. Secondly, graphical sliders could dynamically configure the degree of interest within each running instance of

المنارة للاستشارات
www.manaraa.com

a CSE tool. A slider can simply filter feedback events at the client side of the framework, only displaying feedback events that are beyond a given severity.

8.3 Areas of Enhancement

The Caise framework has been designed to assist developers collaborate during everyday SE tasks. While the Caise framework has several advantages over conventional tools, and provides a strong proof-of-concept for the support of real time CSE, there are aspects of it that could be further enhanced. These aspects are not critical to the success of the Caise approach, but may require addressing for any commercial implementation.

8.3.1 CSCW Floor Control Policies

The Caise framework supports the lowest common denominator for all CSE tasks—unobstructed access to a shared set of artifacts and an underlying semantic model. Caise provides 'free for all' floor control with support for the Mêlée, Action/Reaction, Follow the Leader, Working Together, and Independent modes of development, as presented in Section 3.4.2. Changes can be made without moderation or restriction—it is intended that the awareness mechanisms of Caise-based tools and prevailing social protocols are adequate to prevent concurrent modification difficulties between users.

In some development scenarios, it may be desirable to restrict the levels of concurrent access, reducing the possibility of conflicting actions between developers. The design of floor control mechanisms to restrict access within collaborative applications is a difficult topic [116] that others are working on [101]. As determining the appropriate levels of floor control within software development teams is an open question, no restrictions on floor control have been implemented within the Caise framework to date.

Floor Control Policies for Commercial Tools

The de-facto standard at present for floor control within commercial CSE applications appears to be token-passing, as evident in Borland's JBuilder [12] and Sun's JSE [115] IDEs. While token passing ensures that only one developer at a time can modify the system, it may be overly-restrictive—floor

المنارة للاستشارات
www.manaraa.com

control mechanisms must suit the type of work being done and the SE processes that the development group follow. Floor control clearly is an area for future investigation.

Support for Model Locking

As discussed in Section 2.5.2, the Poseidon collaborative UML editor allows parts of the semantic model to be locked by users during times of concurrent development. Instead of using a token passing system to restrict concurrent access across the entire project, users may select specific areas of the model they wish to develop, which will lock all other developers out until the lock is yielded. This approach may also be incorporated with the Caise framework, where the server supports locking of model components, and tools disable regions of code and diagrams that are currently marked as read-only.

Implementing Floor Control Policies and Model Locking

While specific behaviour for a software methodology can be built into CSE tools without having to alter the structure of the Caise server, industrial-strength implementations of Caise-based tools may require global floor control policies and semantic model locking mechanisms. In this case, it is possible to extend the Caise framework to enforce CSCW floor control policies, providing mechanisms for all Caise-based tools.

Such floor control policies and semantic model-locking facilities could easily be implemented by a new 'security manager' plug-in, where the server checks the plug-in for write access on a per-user basis before allowing a modification request to be processed. The concept of server-based floor control policies is well suited to the design of the framework's event model. Only valid modification requests are processed as normal within the server; all unauthorised requests are rejected with appropriate tool notification events.

8.3.2 Atomic Operations versus Refactoring

The Caise server does not currently handle high-level modification operations such as refactoring. At present, however, it is still possible to create tools that support refactoring by directly manipulating the semantic model,

المنارة للاستشارات
www.manaraa.com

or by inspecting the semantic model and then issuing a sequence of artifact modification events. While the CAISE server will not currently identify such a sequence of events as a refactoring event, it is possible to write a server application to identify potential refactoring events, and even perform refactoring, if required.

Summary

In this chapter, various possibilities to extend the CAISE framework have been presented, allowing the framework to be applicable for a wider range of developers and SE methodologies. Open problems within the CAISE framework have also been discussed.

In Chapter 9, final conclusions about the CAISE framework and participating tools are made, including future work.

للاستشارات المنارة
www.manaraa.com

Chapter IX

Conclusions and Future Work

In Section 9.1, final conclusions for the Caise framework and associated tools are made. In Section 9.2, future work is outlined.

9.1 Conclusions

Real time support for CSE is an important emerging field of research. The size and complexity of today's software projects far exceeds the ability of conventional single-user tools to provide much-needed environments for fine-grained collaboration between developers.

Source code repository systems provide some control over constantly evolving software. There is both the demand and enabling technology, however, for more comprehensive tool support. CSE tools that operate within a shared semantic model of software and receive project updates in real time have the potential to raise the level of communication, cooperation and coordination between developers, improving the SE process.

Current approaches to supporting CSE have inherent limitations, including overly-restrictive floor control policies, reduced tool functionality, high implementation costs and an inability to scale or extend. The Caise framework has been designed to address these problems, with a particular focus on small, well-coordinated development groups.

I have proposed a new approach of shared semantic modelling for the support of CSE. This approach has been embodied within the Caise framework, where different types of powerful and previously unobtainable CSE tools can collaborate in real time upon a shared set of artifacts. The construction and operation of several Caise-based tools has been demonstrated in detail.

Evaluations have shown these new types of CSE tools to be useful to small

www.manaraa.com

teams of software engineers within common development scenarios. Subjective and heuristic evaluations also provide evidence that CAISE represents a highly-viable approach for the future progression of computer-supported CSE.

In this thesis, my contributions include:

- Illustrating the need for more comprehensive tool support within SE
- Introducing and classifying candidate patterns of collaboration within SE
- Describing the CAISE approach to supporting CSE, where different types of tools facilitate synchronous collaboration upon a shared semantic model within small groups of developers. The approach of a central server, a shared semantic model of software, a tool protocol and propagating atomic events is new to the field of research, and the infrastructure has been presented in sufficient detail to be replicated
- Demonstrating that the CAISE framework is suitable for the construction and support of such CSE tools
- Demonstrating new types of tractable semantic model-based tools that support various patterns of CSE
- Presenting candidate heuristics for evaluating CSE tools
- Presenting user evaluations where CSE tools give substantial objective and subjective improvements over conventional SE approaches. This is the first formal evaluation, to my knowledge, of task completion rates and subjective measures for synchronous CSE tools

9.2 Future Work

The basic framework and example tools for CAISE have been constructed and evaluated. Further research can now focus on improving the framework and investigating CAISE-based CSE in more detail.

المنارة للاستشارات
www.manaraa.com

9.2.1 *Areas of Investigation*

Aside from addressing the open problems discussed in Section 8.3, several avenues for future work within the CAISE research project exist.

Abstraction of CAISE into a General CSE Framework The CAISE prototype has proved useful for determining the capabilities and limitations of a framework-based approach to supporting CSE. The key components and interfaces of the framework can potentially be abstracted, allowing different vendors to implement collaborative frameworks and tools that can integrate globally.

User Awareness Mechanisms Determining appropriate types of awareness mechanisms for collaborative user activity is another challenging problem related to the CAISE framework and participating tools. Other research projects such as Maui [55] and GroupKit [95] are currently addressing awareness mechanisms for general CSCW, but awareness mechanisms specific to CSE have so far received little attention. Determining the appropriate volumes of feedback information within CSE tools will also be of significant value to the field of research.

Software Development Visualisations The visualisations presented in Section 7.2 contain considerable amounts of useful information, although they are still relatively simple. A range of more sophisticated visualisations may be developed. Candidate visualisation techniques include the use of colour and other metaphors to indicate user activity attributes. Such attributes may include the current rate of change within a project and the previous locations of developer activity within a set of artifacts.

Code Analysis Analysis of software within the CAISE framework is performed statically. It is of interest to also consider the run-time behaviour of source code when addressing software design. For example, profiling of method calls can indicate methods that should be declared in-line. The CAISE server could be expanded to incorporate dynamic code analysis, with feedback to CAISE-based tools.

المنارة للاستشارات
www.manaraa.com

9.2.2 Future Evaluations

Evaluations of CAISE-based tools presented in this thesis provide evidence that for concentrated tasks between small groups of users, the CAISE approach is suitable. An important further step for the progression of CSE is to investigate how developers interact with each other and CSE tools given more complex and open-ended sets of development tasks. Aspects to be considered include the frequency of communication and reactions to CAISE-based feedback.

Longitudinal Studies

SE tasks typically take days or weeks to complete. An investigation into the fine-grained actions of participants during collaborative tasks over short periods of development has been conducted [29], but a more thorough examination is warranted. Given the CAISE framework's event logging capabilities, a longitudinal study of collaborative development behaviour will be of considerable value. Aspects to consider include bug counts, design aspects and frequency of compilation attempts.

Comparisons to Other Tools

Another interesting evaluation would be the comparison of existing user evaluation results with other broadly comparable tools such as Moomba [92] and Borland's JBuilder [12].

Summary

The anecdotal, heuristic and empirical evaluations presented in this thesis provide strong evidence that the CAISE framework is a suitable approach for supporting CSE. Through the CAISE framework, it is possible and practical to extend the range of tools that support synchronous collaboration facilities. It is envisaged that new aspects of collaborative work within SE can now be explored, allowing the perceived benefits of CSE to be fully realised. I look forward to investigating CSE further, including observation of changes in software development given tools that are more collaboration-aware.

www.manaraa.com

Publications

A listing of all papers related to the work presented in this thesis is given here. Copies of each paper are available from the accompanying resources disc.

Core Articles

The following articles have been published in refereed international conferences. The papers are focused upon original material within this thesis.

APSEC'03

Cook and Churcher, An Extensible Framework for Collaborative Software Engineering, in *Proceedings of the Tenth Asia-Pacific Conference on Software Engineering*, Chang Mai, Thailand, December 2003 [28].

Summary: A description of the initial CAISE architecture and example CSE tools.

APSEC'04

Cook, Churcher, and Irwin, Towards Synchronous Collaborative Software Engineering, in *Proceedings of the Eleventh Asia-Pacific Conference on Software Engineering*, Busan, Korea, December 2004 [31].

Summary: An updated description of the CAISE architecture, with demonstration of a commercial IDE operating as a CAISE-based tool.

ACSC'05

Cook and Churcher, Modelling and Measuring Collaborative Software Engineering, in *Proceedings of the Twenty-Eighth Australasian Conference on Computer Science*, Newcastle, Australia, January 2005 [29].

Summary: A discussion of modelling user activity within the CAISE framework, and the introduction of heuristic evaluations for CSE tools.

Voted as one of the best papers for ACSC2005, and nominated as an invited publication for the Journal of Research and Practice in Information Theory.

المنارة للاستشارات
www.manaraa.com

APSEC'05

Cook, Churcher, and Irwin, A User Evaluation of Synchronous Collaborative Software Engineering Tools, in *Proceedings of the Twelfth Asia-Pacific Conference on Software Engineering*, Taipei, Taiwan, December 2005 [32].

Summary: A description of the CSE tool user evaluation.

ACSC'06

Cook and Churcher, Constructing Real-Time Collaborative Software Engineering Tools Using CAISE, an Architecture for Supporting Tool Development, in *Proceedings of the Twenty-Ninth Australasian Conference on Computer Science*, Tasmania, Australia, January 2006 [30].

Summary: A discussion of tool construction within the CAISE framework.

Related Articles

The following articles are based partially on material from this thesis.

ASWEC'05

Irwin, Cook, and Churcher, Parsing and Semantic Modelling for Software Engineering Applications, in *Proceedings of the Nineteenth Australian Software Engineering Conference*, Queensland, Australia, March 2005 [61].

Summary: This paper describes the semantic analysis of Java source code using the yacc-yacc parser and JST semantic modelling tool. To illustrate the use of the JST semantic modelling tool, this paper discusses the use of JST within the CAISE framework.

Work in Progress

JRPIT

A full version of the ACSC'05 paper [29] is being prepared as an invited paper for the Journal of Research and Practice in Information Technology. This paper is due for submission in March 2006.

Unpublished Articles

The following articles have been published as technical reports within the Computer Science Department, University of Canterbury.

المنارة للاستشارات
www.manaraa.com

Annotated Bibliography

Cook, Collaborative Software Engineering: An Annotated Bibliography, in *Technical Report TR-COSC 02/04*, Department of Computer Science and Software Engineering, University of Canterbury, New Zealand, June 2004 [26].

Summary: This is an annotated collection of papers that have proved relevant during the course of the research into the CAISE framework.

The CAISE *Messaging Framework*

Cook and Churcher, A Pure-Java Group Communication Framework, in *Technical Report TR-COSC 02/03*, Department of Computer Science and Software Engineering, University of Canterbury, New Zealand, July 2003 [27].

Summary: This paper describes the CAISE messaging framework, which can be used to provide asynchronous messaging between any set of Java applications. As the messaging framework is decoupled from the remainder of the CAISE framework, this paper provides a user manual for programmers wishing to develop generic collaborative applications independent of the CAISE framework.

المنارة للاستشارات
www.manaraa.com

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles Techniques and Tools*. Addison Wesley, Reading, MA, 1988. ISBN 0-201-10194-7.

[2] C. Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.

[3] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

[4] K. Baker, S. Greenberg, and C. Gutwin. Heuristic Evaluation of Groupware Based on the Mechanics of Collaboration. In M.R. Little and L. Nigay, editors, *Proceedings of Engineering for Human-Computer Interaction*, volume 2254 of *Lecture Notes in Computer Science*, pages 123–139, Toronto, Canada, May 2001. Springer-Verlag.

[5] Kevin Baker, Saul Greenberg, and Carl Gutwin. Empirical Development of a Heuristic Evaluation Methodology for Shared Workspace Groupware. In *Proceedings of the 2002 ACM Conference on Computer Supported Cooperative Work*, pages 96–105. ACM Press, 2002. ISBN 1-58113-560-2. doi: http://doi.acm.org/10.1145/587078.587093.

[6] Sergio Bandinelli, Elisabetta Di Nitto, and Alfonso Fuggetta. Supporting Cooperation in the SPADE-1 Environment. *IEEE Transactions on Software Engineering*, 22(12):841–865, 1996. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.553634.

[7] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, MA, 1st edition, October 1999.

[8] James Begole, John C. Tang, and Rosco Hill. Rhythm Modeling, Visualizations and Applications. In *Proceedings of the 16th annual ACM symposium on User interface software and technology*, pages 11–20. ACM Press, 2003. ISBN 1-58113-636-6. doi: http://doi.acm.org/10.1145/964696.964698.

المنارة للاستشارات
www.manaraa.com

[9] Brian Berliner. CVS II: Parallelizing Software Development. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 341–352, Berkeley, CA, 1990. USENIX Association.

[10] *BitKeeper User Documentation*. BitMover Incorporated, September 2005. URL `www.bitkeeper.com/UG`.

[11] Marko Boger, Thorsten Sturm, Erich Schildhauer, and Elizabeth Graham. *Poseidon for UML User Guide*. Gentleware AG, 2002. URL `www.gentleware.com`.

[12] *What's New In Borland JBuilder 2005*. Borland Software Corporation, September 2004. URL `www.borland.com/us/products/jbuilder`.

[13] Gerard Boudier, Ferdinando Gallo, Regis Minot, and Ian Thomas. An overview of PCTE and PCTE+. In *SDE 3: Proceedings of the third ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 248–257, New York, NY, USA, 1988. ACM Press. ISBN 0-89791-290-X. doi: http://doi.acm.org/10.1145/64135.65026.

[14] Lionel C. Briand, Christian Bunse, and John W. Daly. A Controlled Experiment for Evaluating Quality Guidelines on the Maintainability of Object-Oriented Designs. *IEEE Transactions on Software Engineering*, 27(6):513–530, 2001. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.926174.

[15] Felix C. Brodbeck. Communication and Performance in Software Development Projects. *European Journal of Work and Organizational Psychology*, 10(1):73–94, March 2001.

[16] Frederick P Brooks Jr. *The Mythical Man Month: Essays on Software Engineering*. Addison-Wesley, 2nd edition, 1995. ISBN 0-201-83595.

[17] W.J. Brown, R.C. Malveau, H.W. McCormick III, and T.J. Mowbray. *AntiPatterns: Refactoring Software, Architectures and Projects in Crisis*. John Wiley & Sons, 1998.

[18] Rich Burridge. *Java Shared Data Toolkit User Guide*. Sun Microsystems, Inc., October 1999. URL `java.sun.com/products/java-media/jsdt`.

المنارة للاستشارات
www.manaraa.com

[19] R. P. Carasik and C. E. Grantham. A Case Study of CSCW in a Dispersed Organization. In *CHI '88: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 61–66, New York, NY, USA, 1988. ACM Press. ISBN 0-201-14237-6.

[20] David Chappell. *Understanding .NET*. Independent Technology Guides. Addison Wesley, 1st edition, May 2002.

[21] Li-Te Cheng, Susanne Hupfer, Steven Ross, and John Patterson. Jazz: A Collaborative Application Development Environment. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 102–103, Anaheim, California, USA, October 2003. ACM Press.

[22] Neville Churcher and Carl Cerecke. GroupCRC: Exploring CSCW Support for Software Engineering. In *Proceedings of the 4th Australasian Conference on Computer-Human Interaction*, Hamilton, New Zealand, November 1996. IEEE Computer Society Press.

[23] Neville Churcher, Warwick Irwin, and Carl Cook. Inhomogeneous Force-Directed Layout Algorithms in the Visualisation Pipeline: From Layouts to Visualisations. In *Australasian Symposium on Information Visualisation, (invis.au'04)*, volume 35 of *Conferences in Research and Practice in Information Technology*, pages 43–51, Christchurch, New Zealand, 2004. ACS.

[24] Alistair Cockburn. *Agile Software Development*. Addison-Wesley, 1st edition, December 2001.

[25] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version Control with Subversion*. O'Reilly Media, 1st edition, June 2004. URL `http://svnbook.red-bean.com/en/1.0/svn-book.pdf`.

[26] Carl Cook. Collaborative Software Engineering: An Annotated Bibliography. Technical Report TR-COSC 02/04, Department of Computer Science and Software Engineering, University of Canterbury, Christchurch, New Zealand, June 2004. Work in Progress.

[27] Carl Cook and Neville Churcher. A Pure-Java Group Communication Framework. Technical Report TR-COSC 02/03, Department of Computer Science and Software Engineering, University of Canterbury, Christchurch, New Zealand, July 2003.

المنارة للاستشارات
www.manaraa.com

[28] Carl Cook and Neville Churcher. An Extensible Framework for Collaborative Software Engineering. In Deeber Azada, editor, *Proceedings of the Tenth Asia-Pacific Software Engineering Conference*, pages 290–299, Chiang Mai, Thailand, December 2003. IEEE Computer Society.

[29] Carl Cook and Neville Churcher. Modelling and Measuring Collaborative Software Engineering. In Vladimir Estivill-Castro, editor, *Proceedings of ACSC2005: Twenty-Eighth Australasian Computer Science Conference*, volume 38 of *Conferences in Research and Practice in Information Technology*, pages 267–277, Newcastle, Australia, January 2005. ACS.

[30] Carl Cook and Neville Churcher. Constructing Real-Time Collaborative Software Engineering Tools Using CAISE, an Architecture for Supporting Tool Development. In Vladimir Estivill-Castro and Gill Dobbie, editors, *Proceedings of ACSC2006: Twenty-Ninth Australasian Computer Science Conference*, volume 39 of *Conferences in Research and Practice in Information Technology*, Tasmania, Australia, January 2006. ACS.

[31] Carl Cook, Neville Churcher, and Warwick Irwin. Towards Synchronous Collaborative Software Engineering. In *Proceedings of the Eleventh Asia-Pacific Software Engineering Conference*, pages 230–239, Busan, Korea, December 2004. IEEE Computer Society.

[32] Carl Cook, Neville Churcher, and Warwick Irwin. A User Evaluation of Synchronous Collaborative Software Engineering Tools. In *Proceedings of the Twelfth Asia-Pacific Software Engineering Conference*, pages 230–239, Taipei, Taiwan, December 2005. IEEE Computer Society.

[33] Donald Cox and Saul Greenberg. Supporting Collaborative Interpretation in Distributed Groupware. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, pages 289–298, Philadelphia, PA, December 2000. ACM Press.

[34] Bill Curtis, Herb Krasner, and Neil Iscoe. A Field Study of the Software Design Process for Large Systems. *Communications of the ACM*, 31 (11):1268–1287, 1988. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/50087.50089.

[35] Edsger W. Dijkstra. The Humble Programmer. *Communications of the ACM*, 15(10):859–866, 1972. ISSN 0001-0782.

المنارة للاستشارات
www.manaraa.com

[36] Stephen G. Eick, Joseph L. Steffen, and Jr. Eric E. Sumner. Seesoft—A Tool for Visualizing Line Oriented Software Statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, 1992. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.177365.

[37] Jacky Estublier. *The Adele Configuration Manager*. John Wiley & Sons, Inc., New York, NY, USA, 1995. ISBN 0-471-94245-6.

[38] Marin Fowler. *UML Distilled: A Brief Guide To The Standard Object Modeling Language*. Object Technology Series. Addison Wesley, Reading, MA, 3rd edition, 2004.

[39] Martin Fowler. *CruiseControl: Continuous Integration Toolkit*. ThoughtWorks Incorporated, November 2005. URL `cruisecontrol.sourceforge.net/overview.html`.

[40] Martin Fowler and Matthew Foemmel. *Continuous Integration*. ThoughtWorks, Inc., October 2005. URL `www.martinfowler.com/articles`.

[41] Jon Froehlich and Paul Dourish. Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams. In *6th International Conference on Software Engineering (ICSE'04)*, pages 387–396, Edinburgh, Scotland, United Kingdom, May 2004. IEEE.

[42] G.W. Furnas. Generalised Fisheye Views. In *Proc ACM SIGCHI '86 Conference on Human Factors in Computing Systems*, pages 16–23, 1986.

[43] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[44] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison Wesley, 1995. ISBN 0201633612.

[45] Emden R Gansner and Stephen C North. An Open Graph Visualization System and its Applications to Software Engineering. *Software—Practice and Experience*, 30(11):1203–1233, September 1999.

المنارة للاستشارات
www.manaraa.com

[46] Christopher Garrett. *Software Modeling Introduction: What Do You Need from a Modeling Tool?* Borland Software Corporation, May 2003. White Paper.

[47] James Gosling, Bill Joy, and Guy Steele. *Java Language Specification*, chapter 18.1. The Java Series. Prentice Hall, 2nd edition, 2000. URL `java.sun.com/docs/books/jls/second_edition/html/syntax.doc.html#44467`.

[48] Nicholas Graham, Hugh Stewart, Authur Ryman, Reza Kopaee, and Rittu Rasouli. A World-Wide-Web Architecture for Collaborative Software Design. In *Software Technology and Engineering Practice*, pages 22–32, Pittsburgh, Pennsylvania, August 1999. IEEE.

[49] Saul Greenberg. The 1988 Conference on Computer-Supported Cooperative Work: Trip Report. In *SIGCHI Bulletin*, volume 20 of *5*, pages 49–55. ACM, July 1989. Also published in Canadian Artificial Intelligence, **19**, April 1989.

[50] Jonathan Grudin. Why CSCW Applications Fail: Problems in the Design and Evaluation of Organizational Interfaces. In D. Marca and G. Bock, editors, *Groupware: Software for Computer-Supported Cooperative Work*, pages 552–560. IEEE Press, Los Alamitos, CA, 1992.

[51] Carl Gutwin, Reagan Penner, and Kevin Schneider. Group Awareness in Distributed Software Development. In *CSCW '04: Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work*, pages 72–81, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-810-5. doi: http://doi.acm.org/10.1145/1031607.1031621.

[52] S. G. Hart and L.E. Staveland. Development of NASA-TLX (Task Load Index): Results of Empirical and Theoretical Research. In P.A. Hancock and N. Meshkati, editors, *Human Mental Workload*, pages 139–183. Elsevier Science, 1998.

[53] Andrew S. Hatch, Michael P. Smith, Christopher M.B. Taylor, and Malcolm Munro. No Silver Bullet for Software Visualisation Evaluation. In *International Conference on Imaging Science, Systems, and Technology (CISST)*, Nevada, USA, June 2001. Computer Science Research, Education, and Applications Press.

[54] C. Helberg. Pitfalls of Data Analysis (or how to avoid lies and damned lies). In *Third International Applied Statistics in Industry Conference*,

المنارة للاستشارات
www.manaraa.com

Dallas, Texas, U.S.A.,, June 1995. URL `my.execpc.com/~helberg/pitfalls`.

[55] Jason Hill and Carl Gutwin. Awareness Support in a Groupware Widget Toolkit. In *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work*, pages 256–267, Sanibel Island, Florida, USA, November 2003. ACM Press.

[56] William C. Hill, James D. Hollan, Dave Wroblewski, and Tim McCandless. Edit Wear and Read Wear. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 3–9, New York, NY, USA, 1992. ACM Press. ISBN 0-89791-513-5. doi: http://doi.acm.org/10.1145/142750.142751.

[57] Scott E. Hudson. *LALR Parser Generator for Java*. Visualization and Usability Center, Georgia Institute of Technology, Atlanta, GA, July 1999. URL `www.princeton.edu/appel/modern/java/CUP`.

[58] Warwick Irwin. *Understanding and Improving Object Oriented Software through Static Analysis*. PhD thesis, University of Canterbury, Christchurch, New Zealand, January 2006. Work in Progress.

[59] Warwick Irwin and Neville Churcher. XML in the Visualisation Pipeline. In David Dagan Feng, Jesse Jin, Peter Eades, and Hong Yan, editors, *Visualisation 2001*, volume 11 of *Conferences in Research and Practice in Information Technology*, pages 59–68, Sydney, Australia, April 2002. ACS. Selected papers from 2001 Pan-Sydney Workshop on Visual Information Processing.

[60] Warwick Irwin and Neville Churcher. Object Oriented Metrics: Precision Tools and Configurable Visualisations. In *9th International Software Metrics Symposium*, Sydney, Australia, September 2003.

[61] Warwick Irwin, Carl Cook, and Neville Churcher. Parsing and Semantic Modelling for Software Engineering Applications. In *Proceedings of ASWEC2005: Nineteenth Australian Software Engineering Conference*, Queensland, Australia, March 2005.

[62] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-57169-2.

المنارة للاستشارات
www.manaraa.com

[63] James O. Coplien and Neil Harrison. *Organizational Patterns of Agile Software Development*. Pearson Prentice Hall, 2005. ISBN 0131467409.

[64] Brian Johnson and Ben Shneiderman. Tree-Maps: A Space-Filling Approach to the Visualization of Hierarchical Information Structures. In G.M. Nielson and L. Rosenblum, editors, *Proceedings of Visualisation '91*, pages 284–291, Los Alamitos, CA, October 1991. IEEE Computer Society Press.

[65] Jim Keogh. *J2EE: The Complete Reference*. McGraw Hill/Osborne, California, USA, 1st edition, 2002.

[66] Scott Lewis. *Eclipse Communication Framework*. Eclipse Foundation, April 2005. URL `www.eclipse.org/ecf/goals.html`.

[67] Adrian Mackenzie and Simon Monk. From Cards to Code: How Extreme Programming Re-Embodies Programming as a Collective Practice. *Computer Supported Cooperative Work*, 13(1):91–117, 2004. ISSN 0925-9724. doi: http://dx.doi.org/10.1023/B:COSU.0000014873.27735.10.

[68] David Martin and Ian Sommerville. Patterns of cooperative interaction: Linking ethnomethodology and design. *ACM Transactions on Computer-Human Interaction*, 11(1):59–89, 2004. ISSN 1073-0516. doi: http://doi.acm.org/10.1145/972648.972651.

[69] Steve McConnell. *Code Complete*. Microsoft Press, Redmond, Washington, 2nd edition, 2004.

[70] T. Mens. A State-of-the-Art Survey on Software Merging. In *Transactions on Software Engineering*, volume 28 of *5*, pages 449–462. IEEE, May 2002.

[71] *Windows SharePoint Services Overview*. Microsoft Corporation, November 2005. URL `www.microsoft.com/windowsserver2003/techinfo/sharepoint/overview.mspx`.

[72] *Visual Studio Developer Center*. Microsoft Corporation, March 2007. URL `msdn2.microsoft.com/en-gb/vstudio`.

[73] *Visual Studio Team System*. Microsoft Corporation, March 2007. URL `msdn2.microsoft.com/en-gb/teamsystem`.

المنارة للاستشارات
www.manaraa.com

[74] Nilo Mitra. SOAP Version 1.2 Part 0: Primer. Technical report, W3C Consortium, June 2003. URL `www.w3.org/TR/soap12-part0`.

[75] David S. Moore and George P. McCabe. *Introduction to the Practice of Statistics*. W H Freeman and Company, New York, 2nd edition, 1993. ISBN 0-7167-2250-X.

[76] *Bugzilla*. The Mozilla Organization, March 2007. URL `www.bugzilla.org/docs`.

[77] Bonnie A. Nardi and James R. Miller. An Ethnographic Study of Distributed Problem Solving in Spreadsheet Development. In *Proceedings of the Conference on Computer Supported Cooperative Work*, pages 197 – 208, Los Angeles, CA, October 1990. ACM.

[78] P. Naur and B. Randell, editors. *Software Engineering: Report of a conference sponsored by the NATO Science Committee*, Garmish, Germany, October 1968. NATO Science Committee.

[79] Blair Neate. *An Object Oriented Semantic Model for .Net*. Honours Report 06/05, Department of Computer Science and Software Engineering, University of Canterbury, Christchurch, New Zealand, November 2005.

[80] Jakob Nielsen. Finding Usability Problems Through Heuristic Evaluation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 373–380. ACM Press, 1992. ISBN 0-89791-513-5. doi: http://doi.acm.org/10.1145/142750.142834.

[81] Jakob Nielsen and Thomas K. Landauer. A Mathematical Model of the Finding of Usability Problems. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 206–213. ACM Press, 1993. ISBN 0-89791-575-5. doi: http://doi.acm.org/10.1145/169059.169166.

[82] Jakob Nielsen and Rolf Molich. Heuristic Evaluation of User Interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 249–256. ACM Press, 1990. ISBN 0-201-50932-6. doi: http://doi.acm.org/10.1145/97243.97281.

[83] *Eclipse Platform Technical Overview Version 2.1*. Object Technology International Incorporated, February 2003. URL `www.eclipse.org/articles`.

المنارة للاستشارات
www.manaraa.com

[84] *SourceForge.net Home Page*. Open Source Technology Group, July 2003. URL `www.sourceforge.net`.

[85] Martin Ott, Martin Pittenauer, and Dominik Wagner. *SubEthaEdit*. The Coding Monkeys, July 2005. URL `www.codingmonkeys.de/subethaedit/collaborate.html`. Website Article.

[86] D.E. Perry, N.A. Staudenmayer, and L.G. Votta. People, Organizations, and Process Improvement. In *Software Magazine*, volume 11, pages 36–45. IEEE, 4th edition, July 1994.

[87] Dewayne E. Perry, Harvey P. Siy, and Lawrence G. Votta. Parallel Changes in Large-Scale Software Development: an Observational Case Study. *ACM Transactions on Software Engineering Methodology*, 10 (3):308–337, 2001. ISSN 1049-331X. doi: http://doi.acm.org/10.1145/383876.383878.

[88] David Pinelle and Carl Gutwin. Groupware Walkthrough: Adding Context to Groupware Usability Evaluation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 455–462. ACM Press, 2002. ISBN 1-58113-453-3. doi: http://doi.acm.org/10.1145/503376.503458.

[89] David Pinelle, Carl Gutwin, and Saul Greenberg. Task analysis for groupware usability evaluation: Modeling shared-workspace tasks with the mechanics of collaboration. *ACM Transactions on Computer-Human Interaction*, 10(4):281–311, 2003. ISSN 1073-0516. doi: http://doi.acm.org/10.1145/966930.966932.

[90] Atul Prakash and Michael J. Knister. Undoing Actions in Collaborative Work. In Marilyn Mantel and Ron Baecker, editors, *Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, pages 273–280, Toronto, Canada, November 1992. Acm Sigchi Siggroup, Acm Press, New York.

[91] Eric S. Raymond, editor. *The Cathedral and the Bazaar*. O'Reilly, 1999. ISBN 1-56592-724-9.

[92] Michael Reeves and Jihan Zhu. Moomba A Collaborative Environment for Supporting Distributed Extreme Programming in Global Software Development. In Jutta Eckstein and Hubert Baumeister, editors, *Lecture Notes in Computer Science*, volume 3092, pages 38–50. Springer-Verlag, January 2004.

المنارة للاستشارات
www.manaraa.com

[93] R. Reichwald, K. Moeslein, H. Sachenbacher, H. Englberger, and S. Oldenburg. *Telecooperation – Distributed Work and Organisational Forms*. Springer-Verlag, Berlin, 1998. German Text.

[94] Steven P. Reiss. *The FIELD Programming Environment: A Friendly Integrated Environment for Learning and Development*. Kluwer Academic Publishers, Norwell, MA, USA, 1995. ISBN 0792395379.

[95] Mark Roseman and Saul Greenberg. Building Real Time Groupware with GroupKit, A Groupware Toolkit. *ACM Transactions on Computer-Human Interaction*, 3(1):66–106, March 1996.

[96] W. W. Royce. Managing the Development of Large Software Systems: Concepts and Techniques. In *ICSE '87: Proceedings of the 9th International Conference on Software Engineering*, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press. ISBN 0-89791-216-0.

[97] M. Sarkar and M.H. Brown. Graphical Fisheye Views. *Communications of the ACM*, 37(12):73–84, December 1994.

[98] Anita Sarma and André van der Hoek. Palantír: Coordinating Distributed Workspaces. In *26th Annual International Computer Software and Applications Conference*, Oxford, England, August 2002. IEEE.

[99] Anita Sarma and André van der Hoek. A Conflict Detected Earlier is a Conflict Resolved Earlier. In *Collaboration, Conflict, and Control : The Proceedings of the 4th Workshop on Open Source Software Engineering*, pages 82–86, Edinburgh, United Kingdom, May 2004.

[100] Till Schümmer. Lost and Found in Software Space. In *34th Annual Hawaii International Conference on System Sciences*, Maui, Hawaii, January 2001. IEEE Computer Society.

[101] Till Schümmer. *Patterns For Groupware*. Groupware Patterns Community, October 2005. URL `www.groupware-patterns.org`.

[102] M. Shooman. *Software Engineering : Design, Reliability, and Management*. McGraw-Hill, New York, 1983.

[103] J. Short, E. Williams, and B. Christie. *The Social Psychology of Telecommunications*. John Wiley & Sons, New York, 1976.

المنارة للاستشارات
www.manaraa.com

[104] Ian Sommerville. *Software Engineering*. Addison Wesley, Reading, MA, 6th edition, 2000. ISBN 020139815X.

[105] Blake Stone. *OpenTools API*. Borland Software Corporation, October 2005. URL `homepages.borland.com/bstone/opentools/doc/ref/OpenToolsAPI`.

[106] M. A. Storey, K. Wong, and H. A. Müller. How Do Program Understanding Tools Affect How Programmers Understand Programs? *Science of Computer Programming*, 36(2–3):183–207, 2000.

[107] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, 39(1), 2000.

[108] Chengzheng Sun. Undo as Concurrent Inverse in Group Editors. *ACM Transactions on Computer-Human Interaction*, 9(4):309–361, 2002. ISSN 1073-0516.

[109] *Remote Procedure Call Protocol Specification*. Sun Microsystems, Inc., April 1998.

[110] *JINI Technology Architectural Overview*. Sun Microsystems, Inc., January 1999. URL `www.sun.com/jini/whitepapers/architecture.html`.

[111] *Java(TM) Message Service Specification Final Release 1.1*. Sun Microsystems, Inc., March 2002. URL `java.sun.com/products/jms/docs.html`.

[112] *RMI Architecture and Functional Specification*. Sun Microsystems, Inc., 2002. URL `ftp://ftp.java.sun.com/docs/j2se1.4/rmi-spec-1.4.pdf`. White Paper.

[113] *Java Software Development Kit*. Sun Microsystems, Inc., November 2004. URL `java.sun.com/j2se/1.4.2/docs`.

[114] *NetBeans IDE*. Sun Microsystems, Inc., November 2005. URL `www.netbeans.org/kb/50`.

[115] *Sun Java Studio Enterprise Edition*. Sun Microsystems Inc., July 2005. URL `www.sun.com/software/index.jsp`.

المنارة للاستشارات
www.manaraa.com

[116] William Tolone, Gail-Joon Ahn, Tanusree Pai, and Seng-Phil Hong. Access Control in Collaborative Systems. *ACM Computing Surveys*, 37(1):29–41, 2005. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/1057977.1057979.

[117] Iris Vessey and Ajay Paul Sravanapudi. CASE Tools as Collaborative Support Technologies. *Communications of the ACM*, 38(1):83–95, January 1995.

[118] J. Christopher Westland. The Cost Behavior of Software Defects. *Decis. Support Syst.*, 37(2):229–238, 2004. ISSN 0167-9236. doi: http://dx.doi.org/10.1016/S0167-9236(03)00020-4.

[119] Timothy Wright. Hierarchical Adaptive Concurrency Control for Synchronous Groupware Applications. Master's thesis, Queens University of Kingston, Ontario, Canada, 1999.

[120] James Wu, T. C. N. Graham, and Paul W. Smith. A Study of Collaboration in Software Design. In *ISESE '03: Proceedings of the 2003 International Symposium on Empirical Software Engineering*, page 304, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2002-2.

[121] Marvin V. Zelkowitz. Perspectives on Software Engineering. *ACM Computer Surveys*, 10(2):197–216, 1978. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/356725.356731.

[122] M.V. Zelkowitz, A.C. Shaw, and J.D. Gannon. *Principles of Software Engineering and Design.* Prentice Hall, 1979.

المنارة للاستشارات
www.manaraa.com

Acknowledgments

I am indebted to Neville Churcher for his patience, guidance and belief throughout the entire course of this research. His ability to remain enthusiastic and attentive every time I walked into his office with yet another new problem was genuinely appreciated. I also appreciated his sensible and subtle ability to bring the project back to its original goals every time I found a new tangent to spend a few years looking at.

Special thanks must also go to Warwick Irwin for so much valuable help along every stage of this project. Warwick has a vast amount of knowledge, skill and energy that he is happy to share with anyone camped outside his office. Without his input into this project, both the research and framework implementation would be of a far lower quality than it is today. Additionally, without the generous contribution of his most excellent semantic analyser software to my project, I would still be wrestling shift/reduce conflicts in Bison right now.

My gratitude must also go out to my co-supervisor, Andy Cockburn. Even though I usually tried to avoid the gaze of his methodical eyes, I truly appreciated his mantra of adhering to correct scientific methods at all times.

I also thank Carl Gutwin for his positive feedback and encouragement during the phase of tool development and evaluation, even from half a world away at times.

Finally, thanks must also go out to all the people along the way who have foolishly asked my how my Ph.D. is coming along, been bored to tears by whatever problem it was that I was working on at the time, and yet replied with supportive comments, genuine belief in my abilities and encouragement.

للاستشارات المنارة
www.manaraa.com

Appendices

المنارة للاستشارات
www.manaraa.com

Appendix A

The Caise Server

A brief description of the Caise server was given Section 5.3.4, which provides details of how the Caise server supports SE functions for CSE tools. More detailed implementation details of the Caise server are provided in this appendix.

A.1 Overview

The Caise server is responsible for the storage of all software artifacts, change history and the semantic model for each Caise-based project. It is also responsible for controlling collaborative access to information that it houses. A further role of the Caise server is to generate and broadcast feedback events to all interested listeners when appropriate, based on user activity.

The internal structure of the Caise server is illustrated in Figure A.1. The components in this figure that have not been described previously in Section 5.3.4 are discussed in this Appendix.

A.2 Language Support

By default, the Caise framework is independent of languages—it is an empty shell only capable of managing groups of collaborating tools and relaying events as they occur. To support a specific programming language, a parser must be available to convert modified source files into parse trees. A semantic analyser must also be in place that can update a semantic model of the project's software from the latest version of parse trees. Finally, source code formatters must be available to generate source files from the semantic model for when it is modified directly.

Multiple languages can be supported within the Caise framework. To introduce a new language, a language-specific parser, semantic analyser and source code formatter are required.

Within Caise, two languages are currently supported. The first, as mentioned previously, is Java 1.4. The second language, created for demonstration purposes, is named Decaf. Decaf is a subset of the Java language, which

www.manaraa.com

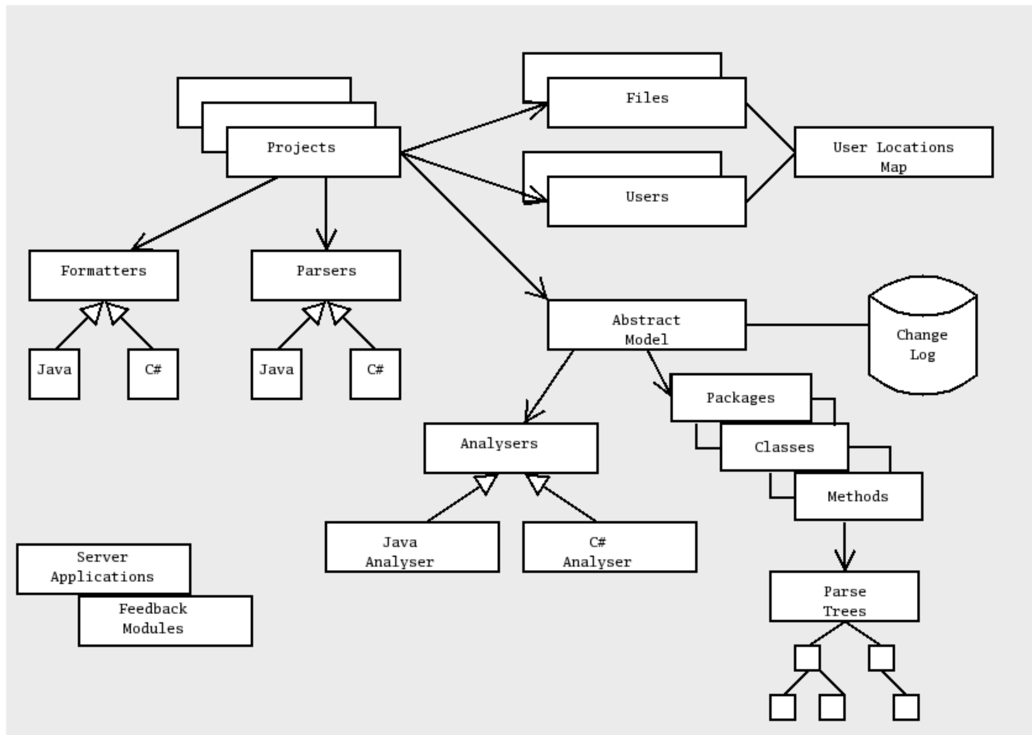


Figure A.1: The CAISE server architecture.

is used to build prototype tools as proofs of concept, prior to extending the tools for Java. The Decaf language is described in Appendix B, and some tools to support Decaf are presented in Chapter 6.

The CAISE framework can also support multiple languages within the same project. To achieve this, multiple parsers and semantic analysers are likely to be required. Additionally, the semantic model must be complete enough to accommodate the conventions of each language.

The role of parsers, analysers and formatters within the CAISE framework is now discussed in detail.

A.2.1 Parsers

Parse trees are used extensively within CAISE as a form of information interchange. Analysers use them internally to determine source code changes. When accepting direct semantic model modification commands, analysers internally construct new versions of parse trees as a convenient way to modify the semantic model—the new parse trees are simply analysed in the same manner as parse trees generated from updated source files. CAISE-based

المنارة للاستشارات
www.manaraa.com

tools also have uses for parse trees, such as determining which line and column terminals appear on. Other tools will also accept parse trees for input into their own local copy of the project semantic model, if one is maintained locally.

The role of parsers within the CAISE framework is to convert source code into well-formed tree structures, based on the corresponding language's grammar. The general role of a CAISE-based parser is illustrated in Figure A.2. The CAISE framework declares a standard parse tree data type, allowing any number of different parsers, even for different languages, to conform to a common standard for subsequent semantic analysis and semantic model integration.

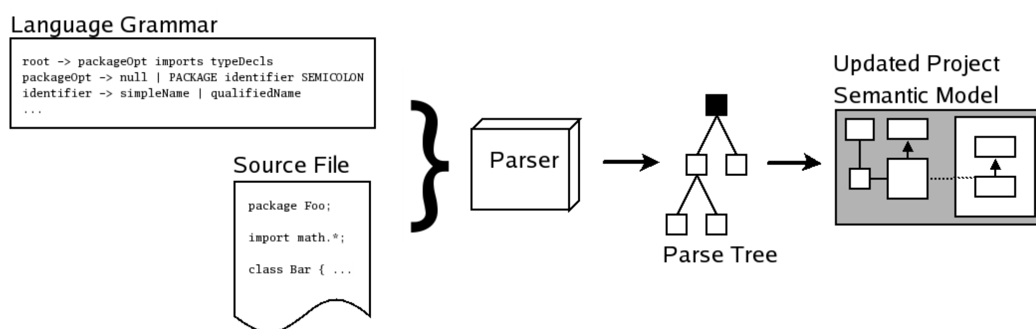


Figure A.2: Typical role of a CAISE-based parser.

There are several techniques for constructing a parser for a given language. Conventionally, tools such as Bison, Yacc or Cup [57] are used to generate a compiler for a specific language. The Decaf parser for the CAISE framework was generated by Cup, and the Java 1.4 parser was generated by the Tomita-based YakYacc tool [61].

A discussion of parsing techniques for OO languages is beyond the scope of this thesis, but is presented elsewhere [58]. For the purposes of the CAISE framework, a CAISE-compliant parser simply needs to output parse trees of the standard CAISE format and conform to the CAISE ParserPlugin interface, as presented in Appendix D.

A.2.2 Source Code Formatters

From within a CAISE project, it is possible that some types of tools such as class diagrammers will directly alter the project semantic model. This is opposed to the manner that text editors operate, where modified source files are parsed, and the resultant parse tree is analysed to actuate the semantic model

المنارة للاستشارات
www.manaraa.com

change. In this case, source files are required to be reverse-engineered from the semantic model of software, allowing text editors to receive an updated view of the project's artifacts subsequent to semantic model modification.

To reverse engineer source files from the semantic model of software, a component known as a source code formatter is employed. A source code formatter is required for each specific language supported within a Caise-based project. Each source code formatter receives the complete parse tree for a source file and emits a sequence of characters for each terminal it traverses. An additional role of source code formatters is to format the source code according to a predefined coding convention.

At present, the Caise framework has been designed so that a source file will only be reformatted when required, such as when a class diagramming tool adds a new method to a file. In the case of a source file that is only modified by a text editor, the source formatter plug-in will not be invoked. It is possible, however, to enforce code formatting standards by configuring Caise to run the source code formatter over source files upon every artifact modification.

All Caise-compliant source code formatters must conform to the Caise FormatterPlugin interface, as presented in Appendix D.

A.3 Artifacts

Artifacts displayed by Caise tools may have multiple representations such as source code buffers and class diagrams. The corresponding artifacts mirrored within the Caise server, however, have no physical representation; they are units of information storage only. As explained in Section 5.3, when users modify an artifact from within Caise tools, the tools themselves relay this request to the Caise server, which updates the authoritative version of the artifact and the underlying semantic model, and sends a modification event to all tools, allowing them to update their local artifact views.

Artifacts within the Caise server have facilities to represent the current source code listing in plain text and a corresponding parse tree. Artifacts also contain a list of users that currently have the artifact opened, and a reference to the user who made the last modification of any kind. Additionally, artifacts within the Caise server also record a complete change history. To support this, the previous version of each altered node within a parse tree is stored in a revision list.

Access to artifacts stored within the Caise server is available through the Caise tool API. While tools typically generate artifact modification events and send these to the server to update Caise artifacts, copies of all project artifacts are available for download and inspection.

www.manaraa.com

A.3.1 The CAISE *Document Buffer*

For single-user tools, an artifact can simply be represented as a plain text file with trivial routines to support the insertion and deletion of characters at given offsets as modifications are made. For collaborative documents the task of supporting modification is considerably more complicated. Mechanisms must be in place to ensure that the edits to the document remain consistent, and that the view of each tool is kept synchronised with the version on the server.

To assist in the support of collaborative artifact modification, a component internal to CAISE artifacts known as a document buffer is employed, as illustrated in Figure A.3. A document buffer is a collaboration-aware text repository that maps the locations of all users within the body of text. Requests for insertions and deletions of text are performed by the document buffer using the last known position of the requesting user, rather than using an explicit file location. When the buffer is modified, user positions are updated accordingly.

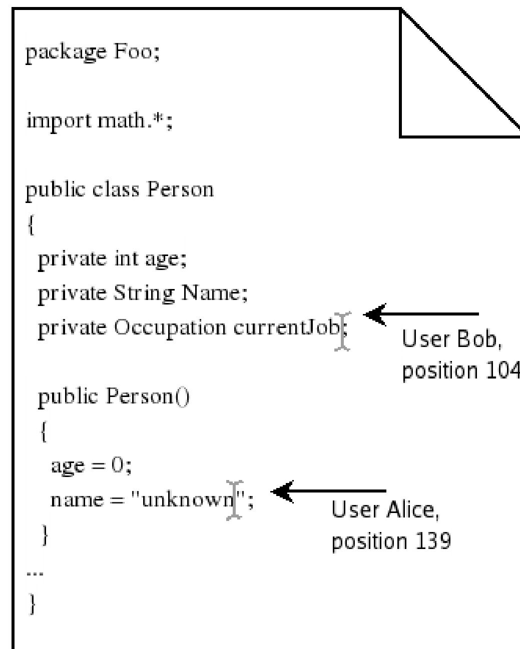


Figure A.3: User positions within a CAISE document buffer.

The reason for performing artifact modifications on known positions rather than explicit locations is that when making modification requests, tools can not always determine that the absolute position will remain fixed until the server has accepted the modification request. For example, if two separate

www.manaraa.com

requests are made to modify the document at the same time, the second request, when processed, will have an incorrect absolute file position. By using known positions within the document buffer, the second request will be processed correctly, as all user positions will have already been updated appropriately.

Upon artifact modification within the CAISE server, events are broadcast to all artifact viewers, allowing them to update their copies of the affected artifact accordingly. This distributed Model-View-Controller [44] design of the CAISE document buffer is discussed further in Section 6.2.4.

The Collaborative Text Pane within the CAISE widgets package, as presented in Section 6.2.3, uses the CAISE document buffer as the underlying repository and controller for the text that it displays. This approach is in a manner very similar to the Model-View-Controller design of the standard Java Swing text widgets.

Fine-Grained Tool Synchronisation

A problem occurs when a partial, or syntactically-incomplete, change is being made in a source file, and someone else makes a full change from a tool that does not share the same artifact view, such as a modification in a class diagram. In this case, either the CAISE server or participating CSE tools are required to merge the in-progress partial modification with the newly updated artifact in order to preserve the work efforts of both parties. Appropriate feedback information between highly related developers also helps keep this type of problem to a minimum.

A.3.2 Implementing Collaborative Undo

Collaborative undo is well known to be a very challenging problem [108, 90, 119]. For genuinely useful CSE tools, however, collaborative undo must be supported.

Two types of undo are possible within collaborative tools. The first type, that I call global undo, is where an undo request from any user will undo the last action that modified the artifact, independent of which user actually made the change. Global undo is virtually as easy to implement as standard, single-user undo, but it is not intuitive to use in a collaborative setting. For example, it is likely to be out of place for one user to see his or her code changes disappear when another user presses the undo button. Similarly, it is not desirable to have to delete other users' later modifications before having an opportunity to erase some of one's own work.

Local undo, as I name it, is a more intuitive form of undo, where only one's own actions can be undone. This gives the perspective of working somewhat

المنارة للاستشارات
www.manaraa.com

independently, knowing that regardless of the modification activity, pressing the undo button will only reverse the latest changes of the local user. The difficulty, of course, is that local undo is challenging to design and implement.

The Caise document buffer maintains a per-user artifact modification stack, which provides local undo capabilities. When a user invokes an undo command, the document buffer determines which artifact modification event is to be reversed, updates the underlying text, and broadcasts this text modification event out to all viewers of the artifact for local artifact adjustment.

A.3.3 Tool Manager Plug-Ins

The Caise framework provides generic support for the collaborative editing of text documents, the parsing of source files, and the semantic analysis of parse trees derived from source code and UML diagrams. In some cases, however, tools require further functionality from the server, including the support of new artifact types. To accommodate extensibility within the Caise server, plug-ins known as *tool managers* can be integrated through the Caise plug-ins interface. The tool manager interface specification is given in Appendix D.

For a UML class diagrammer, it is apparent that such a tool requires information beyond what is contained within the core semantic model of the project. As well as displaying all classes, methods and relationships, a class diagrammer contains class layout information that must be shared every time any instance of the class diagram is modified. Therefore, when implementing the UML diagramming tool presented in Section 6.3.2, an additional diagrammer-specific type of artifact to store layout information was introduced, which was managed and shared by a UML-specific tool manager.

For the UML class diagramming tool, whenever a user changes the location of a displayed class, a tool-specific event is thrown to the Caise server via the Caise tool API, and this event is proxied to the UML diagrammer tool manager. The tool manager will then access and update the new artifact that stores the class location information, and then broadcast this change out to all tools in accordance with the Caise tool protocol, allowing all users to update their local view of the UML class diagram.

The Caise server has no knowledge about the structure or semantics of new types of artifacts that tool managers introduce. Rather, the Caise server relies upon the tool manager to handle all modification requests as invoked by the UML diagramming tool. The Caise server's role in this case is only to store and control concurrent access to the artifact.

The Caise tool API allows new artifacts to be loaded against a project with a specific tool ID. This ID is used by CSE tools to retrieve the artifact and update it by way of the associated tool manager plug-in.

المنارة للاستشارات
www.manaraa.com

A.4 Server Applications

The most common type of CAISE-based tools are those that allow distributed editing, building and inspection of collaborative software projects. Other more static types of tools can be envisaged, however, such as visualisation generators and metrics gathering tools. For this class of tool, the CAISE framework supports *server-based* applications. These applications run within the server process itself, providing fast and efficient access to the software project, its artifacts and the underlying semantic model.

An example of a typical server application could be a project management tool that creates an entry in a log whenever certain code metrics have been violated. Another example of a simple server application is presented in Figure A.4. This is a simple example of a server application, where a project semantic model is inspected and the names of all classes within the package structure are displayed. The code segment listed in Figure A.5 shows how simple it is for the above application to walk the semantic model programmatically through an instance of CAISE's *Model Visitor* class.

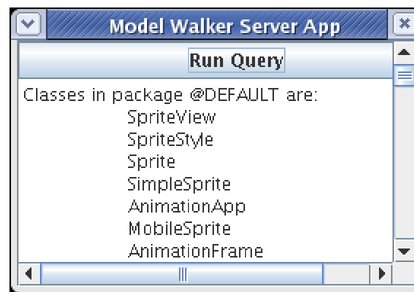


Figure A.4: A server application which inspects the semantic model of a CAISE-based software project.

Although an uncommon requirement, server applications have the ability to modify a project's semantic model directly as well, causing semantic model change events to be issued to all listening tools, which will in turn adjust their local views accordingly. An example of one such tool could be an automatic refactoring plug-in that periodically repairs easily identifiable violations of software design.

All CAISE-compliant server applications must conform to the CAISE Server-Application interface, as presented in Appendix D.

The Model Visitor Class

The Model Visitor class, demonstrated in Figure A.5, is used by server applications and CAISE-based tools to traverse the project semantic model.

المنارة للاستشارات
www.manaraa.com

```
public class SimpleModelWalker extends CAISEServerApp {

    // called once server is ready
    public void run() { initGui(); }

    // called upon events
    public void update(Collection events) { /* do nothing */ }

    /* InitGui() method omitted */

    public void jButton1ActionPerformed(ActionEvent e) {

        // get the given project
        Project project = Engine.getEngine().getProject("AA");

        // get the default package from the project's model
        PackageDecl pkg = project.getModel().getDefaultPackage();

        // print out header
        setText("Classes in package " + pkg.getSimpleName());

        // create an instance of a subclassed model visitor
        new ModelVisitor(pkg) {

            // override the visit ClassType routine
            public void visitClass(ClassType classType) {

                // write the class name out to the text panel
                addText("\n\t" + classType.getSimpleName());
            }
        }.visit(); // fire up the adapter
    }
}
```

Figure A.5: The code listing for a simple server application.

المنارة للاستشارات

www.manaraa.com

Following the Visitor idiom [44], user routines are supplied for when a declaration is reached, such as a package, class, or method. The Model Visitor class is similar in design to the Model Adapter supplied with Borland's OpenTools API [105].

The visitor is initialised with the root declaration that is to be inspected. This could be the top-level package of the entire semantic model, or something as specific as a method body. Any user methods supplied to the visitor will be invoked when appropriate.

It is important to note that the visitor does not perform an exhaustive search of the semantic model from each declaration, as this may cause infinite loops and repeated search paths. Rather, only the direct declarations contained in each parent declaration are inspected. For example, a class will only have its directly declared fields and methods inspected. The superclass and any subclasses will not be inspected by default, but they can be by adding explicit code to do so in the user routines.

A.5 The CAISE Event Log

As discussed in Section 5.3.2, the CAISE framework is based upon sequences of small, highly frequently occurring events. These event sequences facilitate the synchronous sharing of artifacts and the incremental updating of the underlying semantic model. Additional events may also be spawned from artifact modifications, such as feedback events related to user presence and changing of the semantic model's state.

The role of the CAISE event log is to capture each event raised within a CAISE project in chronological order. There are many different types of events within a project, such as project events, text chat events, artifact modification events, project compilation attempts and specific user actions such as changing location within a file. Event types were discussed in detail in Section 5.3.2.

CAISE tools can download a project's full event log for inspection, analysis and visualisation. Visualisations of the event log for a sample CAISE project were presented in Section 7.2.

A.6 Project Administration

To create a new CAISE project, configure it for use with particular languages and feedback plug-ins, or to inspect the state of an existing project, the Project Manager Panel is used. This panel is presented in Figure A.6.

The Project Manager Panel is a stand-alone application that connects to the specified CAISE collaboration server. All information for each project currently stored on the server is available from this panel. Information on

www.manaraa.com

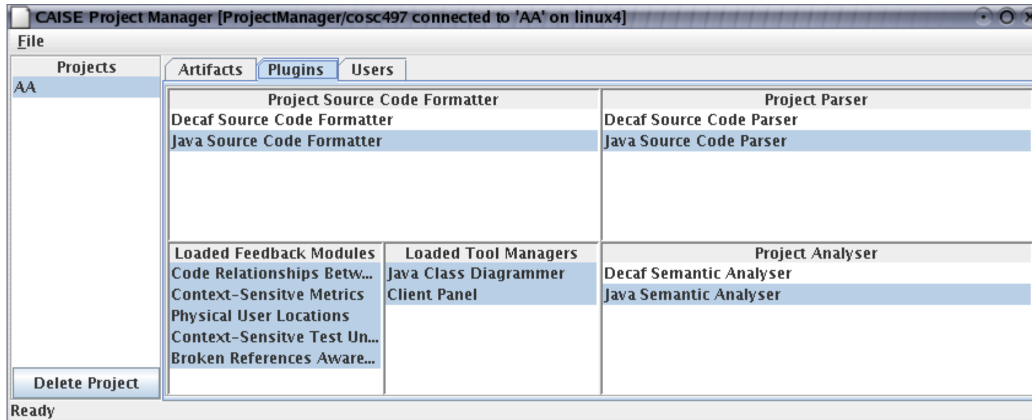


Figure A.6: The plug-ins configuration panel of the Project Manager Panel.

each artifact can be displayed, including modification date, current editor and current viewers. Text messages can be sent to members of the project group from within this panel on behalf of the project manager user.

To configure a newly created CAISE project, the relevant analysers, formatters and parsers for the project must be selected. Typically, one of each will be selected according to the language that the project is written in. As can be seen in Figure A.6, the Project Manager Panel also highlights the feedback plug-ins available for each project. For the situation where some tools within a project require feedback information and others do not, each tool can simply elect whether or not to register for custom feedback during project connection.

As server plugins are loaded independent of individual CAISE projects, they are not displayed within the Project Manager Panel. Additionally, in the current version of the CAISE framework there is no option to disable server plugins for specific projects, but this is trivial to support if required.

Another task supported by the Project Manager Panel is that of server shut-down. By invoking the shutdown function, the server disconnects all client connections, and serialises its entire collection of projects to disk before exiting. At this stage, the server also issues an event to all CAISE tools requesting them to shut down.

A.7 The Plug-Ins Interface

The plug-ins interface provides a means for extending the CAISE framework without modification of the server. CAISE-based plug-ins are compiled Java classes that implement a CAISEPlugIn interface.

المنارة للاستشارات
www.manaraa.com

As discussed previously in this section, CAISE plug-ins include parsers, analysers, source code formatters, feedback plug-ins and server applications. There is one other type of plug-in within the CAISE framework, known as a tool manager. Tool managers were discussed in Section A.3.3.

All plug-ins are loaded within the CAISE server on startup. As discussed previously, the project manager tool is used to associate specific plugins with projects. The CAISE server inspects a predefined library directory for plug-in plug-ins, and loads them into the CAISE server process. The CAISE server determines which type of plug-in they are, and adds them to the appropriate list of available plug-ins.

The interface specifications for each type of plug-in are presented in Appendix D.

A.8 Interprocess Communication

CAISE is a concurrent system, where it is likely to receive multiple interleaved requests to modify a set of artifacts within a short period of time. Invocations of CAISE tool API methods are treated fairly at the CAISE server. In the underlying distributed system that CAISE employs for its client interface, each incoming method call is queued and then processed in sequential order. For all other pending method calls in the queue, a low-CPU blocking mechanism is used on the client side.

While the CAISE tool API makes the CAISE server appear directly accessible via conventional method calls, in reality the server is shared by an unbounded number of collaborating CAISE tools. Therefore, the CAISE tool API is designed to be thread safe, allowing any number of threads from any number of processes to access the server concurrently. It was essential to implement a multi-threaded API for the CAISE server, and it simplifies application programming—developers do not need to be concerned with calling the API from only a specific thread within their application.

A.8.1 Asynchronous Communication

Within CAISE, a purpose-built messaging framework is used for communication between the server and all participating applications. The messaging framework, known as `caise.messaging`, is presented in an accompanying technical report (see Appendix H). The design of the messaging framework allows any group of applications to send asynchronous messages of any data type to each other using the Observer/Observable design idiom [44]. Within the messaging framework, all data is compressed between endpoints, reducing the network demands of any system that requires collaborative capabilities. This is particularly useful for collaborative applications on a slow network.

www.manaraa.com

The messaging framework is implemented as a simplified version of the Java Shared Data Toolkit (JSDT) [18]. As the messaging framework consists of a single pure-Java package, it is easily used within applications without high programmatic or architectural overhead. As such, the CAISE tool API employs the messaging framework to deliver asynchronous messages to distributed CAISE-based tools.

A.8.2 Synchronous Communication

The messaging framework only provides broadcast, or asynchronous communication. Within the CAISE framework, synchronous messaging is also required between CAISE-based tools and the CAISE server to allow API methods to be invoked. Therefore, for synchronous communication within the CAISE tool API, RMI [112] is employed. The only architectural consideration in using RMI is that an RMI server process must be running on each machine in the network that uses the CAISE framework; therefore the CAISE API that all CAISE-based tools use loads the RMI server process on tool startup.

Many tools can compete for the CAISE server's attention at any point in time. As part of CAISE's RMI-based implementation, the CAISE server uses a round-robin mechanism to process one incoming command from each tool at a time. Therefore, even if one tool issues a large number of commands in a row, a second tool will not have to queue long before having its first command processed. While this might seem surprising, it ensures fairness between tools. If queuing of multiple requests was based entirely on order of arrival, fairness between tools could not be ensured.

A.8.3 Selection of Distributed Communication Technologies

Many competing communication technologies are available to implement systems such as the CAISE framework. RPC [109] was an obvious first consideration, but it was too complicated and low-level compared to other distributed systems available today. Conversely, SOAP [74] and other web services appeared too high-level and generic, with considerable programmatic and architectural overheads compared to frameworks such as RMI. Other emerging technologies considered included JMS [111] and JINI [110]; both of which may have provided adequate communication facilities to implement the CAISE server and support CAISE-based tools.

At the time of framework development, RMI was the most suitable candidate distributed system. The JSDT was also considered, but its overhead was too high and a more lightweight system was required. RMI's support for synchronous communication was also significantly stronger than JSDT's.

www.manaraa.com

The well established GroupKit [95] architecture could have also been used to provide asynchronous communication facilities within the CAISE framework. Unfortunately, support for Java-based applications within GroupKit is limited. The Maui [55] toolkit was also considered, but like GroupKit, its limited support for synchronous communication made it unsuitable for use within the CAISE framework.

Additionally, by having all services accessed through one central server, including facilities for communication provided internally by RMI, the CAISE framework was simplified—no dependencies on external libraries or toolkits are required.

Summary

The details in this appendix provide a starting point for CSE tool development using the CAISE framework. Examples and descriptions of CAISE-based CSE tools are presented in Chapter 6.

المنارة للاستشارات
www.manaraa.com

Appendix B

Language Specification for Decaf

B.1 Overview

The Decaf language is a subset of Java. It has been derived for testing purposes within the CAISE framework. Decaf is a pure object oriented language; it has no primitive types, predefined operators or complicated language constructs such as pointers.

Decaf is currently only supported by a parser, semantic analyser and prototype development tools. A compiler specifically built for the language does not exist, but it would be trivial to process and pipe Decaf programs into a Java compiler to produce working applications. Additionally, as Decaf is modelled by the CAISE framework's general model of object oriented software, Decaf programs will be able to be compiled once a generic compiler for CAISE-based projects has been introduced.

B.2 An Example Source File

For demonstration purposes, an example Decaf source file is given in Figure B.1.

B.3 An Example Grammar

The full Decaf grammar is presented in Figure B.2.

المنارة للاستشارات
www.manaraa.com

```
class Foo {
    Integer i;
    Collection wibbles;

    Bar getBar() {
        return Bar.makeBar(this);
    }

    void addWibble(Wibble w) {
        wibbles.add(w);
    }
}

class Bar {
    String getName() {
        return "Bar object";
    }

    Bar makeBar(Class caller) {
        return this;
    }
}
```

Figure B.1: An example source file for the Decaf language.

للاستشارات المنارة
www.manaraa.com

```
Artifact             →  ϵ
                        | Class Artifact ;
Class                →  CLASS ID LBRACE Members RBRACE ;
Members              →  ϵ
                        | MemberDecl Members ;
MemberDecl           →  PropertyDecl
                        | MethodDecl ;
PropertyDecl         →  TypedDecl SEMICOLON ;
TypedDecl            →  ID ID ;
MethodDecl           →  TypedDecl LPAREN OptionalParameters RPAREN Body ;
OptionalParameters   →  ϵ
                        | ParameterDecl ;
ParameterDecl        →  TypedDecl
                        | TypedDecl COMMA ParameterDecl ;
Body                 →  LBRACE OptionalStatements RBRACE ;
OptionalStatements   →  ϵ
                        | Statement OptionalStatements ;
Statement            →  Expression SEMICOLON
                        | VariableDecl SEMICOLON
                        | ReturnStatement SEMICOLON ;
Expression           →  MethodCall
                        | Assignment
                        | ID
                        | Literal ;
VariableDecl         →  TypedDeclaration ;
ReturnStatement      →  RETURN Expression ;
MethodCall           →  ID DOT ID LPAREN OptionalArguments RPAREN ;
Assignment           →  ID EQUALS Expression ;
Literal              →  INTEGER_LITERAL
                        | STRING_LITERAL ;
OptionalArguments    →  ϵ
                        | Argument OptionalArguments ;
Argument             →  Expression
                        | Expression COMMA Argument ;
```

Figure B.2: The full grammar for the Decaf language.

المنارة للاستشارات
www.manaraa.com

Appendix C

CAISE Event Log DTD

This DTD represents the structure of the CAISE event log. The CAISE event log is stored in XML format, and is validated against this DTD for consistency. The event log consists of three core sections:

Components Each declaration within the model is listed as a component within the DTD. Components incorporate a unique identifier to assist locating it within the live CAISE-based semantic model. Components can represent packages, types, methods and blocks as well as low-level declarations such as parameters and local variables

Users Each user within the CAISE project is also uniquely identified

Events Every event within the lifetime of the CAISE project is recorded. Event types consist of actions from tools such as artifact modifications, actions of users such as a compilation attempt, and semantic events such as the resolution of a method invocation to a given method declaration.

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!-- DTD for representing events in a CAISE project -->

<!ENTITY version "V0.010alpha">

<!-- An event log consists of a header, followed by the used components in the model,
     the users within the project, and the events generated by users -->
<!ELEMENT CAISEEventLog (Header, Components, Users, Events)>
<!ATTLIST CAISEEventLog version CDATA #IMPLIED>

<!ELEMENT Header (ProjectName, Generator?, TimeStamp?) >
<!ELEMENT ProjectName (#PCDATA) >
<!ELEMENT TimeStamp (#PCDATA) >
<!ELEMENT Generator (#PCDATA) >

<!-- Only components involved in the logged events. See model
     dump for all components and relationships.
     A component is a decl in the semantic model -->
```


www.manaraa.com

```
<!ELEMENT Components (Component*)>
<!ELEMENT Component EMPTY>
    <!ATTLIST Component
        id ID #REQUIRED
        type (interface | class | constructor | field | method | package | parameter
        | sourcefile | variable | block | catchblock) #REQUIRED
        name CDATA #REQUIRED
        owner IDREF #REQUIRED
    >

<!ELEMENT Users (User*) >
<!ELEMENT User (Tool, Host) >
    <!ATTLIST User
        name ID #REQUIRED>
<!ELEMENT Tool (#PCDATA) >
<!ELEMENT Host (#PCDATA) >

<!-- An event is anything that the server processes and informs caise tools about
     within the feedback loop. Incidentally, changes to Auxillary artifacts are
     logged via ToolManager events, but the creation and opening of auxillary
     artifacts are not. -->
<!ELEMENT Events ((Project | Client | Artifact | Change | Chat | Feedback | ModelUpdate
        | ToolManager)*) >

<!-- Project event such as a new project being created -->
<!ELEMENT Project EMPTY >
    <!ATTLIST Project
        user IDREF #REQUIRED
        action (added | deleted) #REQUIRED
        timestamp CDATA #REQUIRED
    >

<!-- Chat message sent via client panel -->
<!ELEMENT Chat (Message) >
    <!ATTLIST Chat
        from IDREF #REQUIRED
        to IDREF #REQUIRED
        timeOffset CDATA #REQUIRED
    >

<!-- Feedback plugin response to an event -->
<!ELEMENT Feedback (Message) >
    <!ATTLIST Feedback
        type CDATA #REQUIRED
        srcUser IDREF #REQUIRED
        destUser IDREF #REQUIRED
        srcEntity IDREF #REQUIRED
        destEntity IDREF #REQUIRED
        timeOffset CDATA #REQUIRED
    >
```


المنارة للاستشارات
www.manaraa.com

```
<!ELEMENT Message (#PCDATA) >

<!-- Client has requested that the model be directly changed -->
<!ELEMENT ModelUpdate (EMTPY) >
    <!ATTLIST ModelUpdate
        user IDREF #REQUIRED
        type (addition | deletion | modification) #REQUIRED
        component IDREF #REQUIRED
        timeOffset CDATA #REQUIRED
    >

<!-- Tool manager events - such as component moved in class diagrammer -->
<!ELEMENT ToolManager (CustomData)?>
    <!ATTLIST ToolManager
        pluginID CDATA #REQUIRED
        user IDREF #REQUIRED
        timeOffset CDATA #REQUIRED
    >
<!ELEMENT CustomData (#PCDATA)>

<!-- a Client event is an action such as a project opened, a location changed, etc -->
<!ELEMENT Client (Object?, Location?, Result?) >
    <!ATTLIST Client
        user IDREF #REQUIRED
        action (location_changed | opened_artifact | closed_artifact | opened_project |
        closed_project | rebuilt_project) #REQUIRED
        timeOffset CDATA #REQUIRED
    >

<!ELEMENT Object (#PCDATA) >

<!ELEMENT Location (#PCDATA) >

<!ELEMENT Result EMPTY >
    <!ATTLIST Result
        rebuild (succeeded | failed) #REQUIRED
    >

<!-- an Artifact event denotes something that happend to an artifact -->
<!ELEMENT Artifact (AppendInfo?) >
    <!ATTLIST Artifact
        user IDREF #REQUIRED
        action (added | appended | replaced | saved | save_failed | deleted) #REQUIRED
        component IDREF #REQUIRED
        timeOffset CDATA #REQUIRED
    >

<!ELEMENT AppendInfo EMPTY >
    <!ATTLIST AppendInfo
```


المنارة للاستشارات
www.manaraa.com

```
        type (keyTyped) #REQUIRED
        keyChar CDATA #REQUIRED
        asciiValue CDATA #REQUIRED
        fileOffset CDATA #REQUIRED
    >

<!-- A Change event represents a semantic change to a component in the model.
     This is normally generated from a user input event, such as a modfication to
     a source file. -->
<!ELEMENT Change (ReferenceInfo?) >
    <!ATTLIST Change
        user IDREF #REQUIRED
        type (addition | deletion | modification | reference_resolved | reference_unresolved |
        references_fully_resolved) #REQUIRED
        component IDREF #REQUIRED
        timeOffset CDATA #REQUIRED
    >

<!ELEMENT ReferenceInfo (#PCDATA) >
```


www.manaraa.com

Appendix D

CAISE Server Plug-Ins Specification

In this appendix, the interface details for each type of CAISE plug-in are highlighted. A user manual for the CAISE framework, including a Javadoc description of each plug-in interface, is available from `www.cosc.canterbury.ac.nz/clc/cse`. The listings given here provide an introduction for each interface.

D.1 CAISEAnalyser

The CAISEAnalyser interface supports the adding of parse trees to the semantic model. A parse tree is mapped to a given source file. New parse trees are also capable of being generated by a CAISEAnalyser, which is typically the result of making a direct semantic model modification request through the CAISE tool API.

CAISEAnalysers must also be able to cross-reference the current semantic model, which is normally requested by the CAISE server after a parse tree has been added or removed.

CAISEAnalysers also provide facilities for searching the semantic model, according to the location of the search starting point and the scope rules of the corresponding language.

Method:	`void addParseTree(SourceFile sourceFile)`
Description:	Adds the given parse tree to the semantic model
Parameters:	`SourceFile sourceFile`, the source file containing the parse tree
Returns:	Nothing

Method:	`Artifact updateParseTree(CAISEEvent event)`
Description:	Construct an updated parse tree according to the requested change to the semantic model
Parameters:	`CAISEEvent event`, the event holding the requested semantic model change information
Returns:	A copy of an `Artifact` that holds the new parse tree

www.manaraa.com

Method:	`void crossReference(SourceFile sourceFile)`
Description:	Cross reference all currently known symbols in file by running through the parse tree, resolving references to named types
Parameters:	`SourceFile sourceFile`, the file to be cross-referenced within the semantic model
Returns:	Nothing

Method:	`TypeDecl lookupType(Scope currentScope, String typeName)`
Description:	Look for a given type within the semantic model
Parameters:	`Scope currentScope`, the scope at the point of lookup `String typeName`, the name of the type being looked up
Returns:	The `TypeDecl`, if found

Method:	`PackageDecl lookupPackage(String packageName)`
Description:	Look for a given package within the semantic model
Parameters:	`String packageName`, the name of the package being looked up
Returns:	The `PackageDecl`, if found

Method:	`MethodDecl lookupMethod(Scope currentScope, String methodName, List paramTypes)`
Description:	Look for a method within the semantic model
Parameters:	`Scope currentScope`, the scope at the point of method lookup `String methodName`, the name of the method being searched `List paramTypes`, the parameter list of the method being searched
Returns:	The `MethodDecl`, if found

D.2 *CAISEFeedback*

The CAISE server invokes CAISEFeedback modules upon the occurrence of any CAISE tool event. CAISEFeedback modules typically inspect the current project and create a collection of relevant feedback events for each user.

Method:	`Map getFeedback(Project project)`
Description:	Generate custom feedback given the current state of the semantic model
Parameters:	`Project project`, the project to inspect
Returns:	A `Map` where each key is a specific user, and each corresponding value is a collection of custom CAISE feedback events for that user

المنارة للاستشارات
www.manaraa.com

D.3 CAISEFormatter

CAISEFormatters are given a CAISE-based parse tree, and their role is to produce a corresponding source file. Source files are returned as plain text. Each CAISEFormatter will format code according to a user-defined specification, which is implemented internally.

Method:	`String format(Object parseTree)`
Description:	Translate a parse tree into a well-formatted source file
Parameters:	`Object parseTree`, the parse tree to be formatted
Returns:	A String representation of the newly created source file

D.4 CAISEParser

CAISEParsers translate plain-text source files into CAISE-based parse trees. Parse trees can be constructed in any means possible; typically parser generator tools are used to implement CAISEParsers.

Method:	`Nonterminal parseBuffer(String buffer)`
Description:	A CAISE parser converts source code into a CAISE-compliant parse tree
Parameters:	`String buffer`, the source code in plain text
Returns:	A CAISE-compliant parse tree, with a `Nonterminal` as the root

D.5 CAISEServerApp

CAISEServerApps are loaded at start-up by the CAISE server. Upon loading, the `init` method is invoked, allowing CAISEServerApps to perform any implementation-specific start-up routines. Upon successful invocation of the initialisation method, each CAISEServerApp is executed via the `run` method in a separate thread.

During operation of the CAISE server, CAISEServerApps are notified each time a CAISE event is generated. CAISEServerApps are not permitted to create additional events for distribution to CAISE-based tools, but they do have direct access to the semantic model of each project contained within the CAISE server, and may modify it as required.

Method:	`boolean init()`
Description:	Initialise the server application. Called on server startup
Parameters:	None
Returns:	Returns true if the server application was successfully initialised

المنارة للاستشارات
www.manaraa.com

Method:	`void run()`
Description:	Starts the server application if it has been successfully initialised
Parameters:	None
Returns:	Nothing. This method will only return if the server application shuts itself down

Method:	`void update(Collection events)`
Description:	A call-back method from the CAISE server upon any event being raised within the framework. Server applications may require knowledge of specific events as the occur
Parameters:	`Collection events`, The current cache of events within the framework
Returns:	Nothing

D.6 CAISEToolManager

CAISEToolManagers are responsible for the management of auxiliary artifacts specific to a given tool. CAISE-based tools may request an artifact to be opened, at which time a copy of the artifact is returned to the calling tool via the CAISE server.

To modify an artifact, CAISE tools call the `fireToolEvent()` API method, which is passed to the CAISEToolManager's `processToolEvent()` method. The CAISEToolManager updates the specified artifact in an implementation-specific manner, and then propagates the resultant change to all tools through the return value of the `processToolEvent` method.

Method:	`Object openAuxilaryArtifact(String artifactID, Client requestor)`
Description:	Open an artifact that is controlled by this tool manager
Parameters:	`String artifactID`, a unique identifier for this artifact `Client requestor`, the user that requests access to the artifact
Returns:	An `Object` that represents a copy of the auxiliary artifact

Method:	`CAISEEvent processToolEvent(CAISEEvent event)`
Description:	Process an input event from one tool
Parameters:	`CAISEEvent event`, the event thrown from the participating CAISE tool
Returns:	A corresponding `CAISEEvent` which will be sent to all listeners

المنارة للاستشارات
www.manaraa.com

Method:	`void closeAuxilaryArtifact(String artifactID, Client requestor)`
Description:	Close access to an artifact that is controlled by this tool manager
Parameters:	`String artifactID`, a unique identifier for this artifact `Client requestor`, the user that no longer requires access to the artifact
Returns:	Nothing

المنارة للاستشارات
www.manaraa.com

Appendix E

IDE Integration

At the time of framework development, Together Architect for Java [46] was chosen for framework integration. This was arguably the most comprehensive Java IDE on the market, and had a wide user base. Together Architect offers a plug-ins API, allowing tool developers to add new components within the IDE, and to listen for events such as file activity.

Using a plug-in that acted as a proxy between Together Architect and the Caise server, a means was established to incorporate the Together IDE as a Caise-based tool. Adhering to the Caise tool protocol, any change made within Together Architect was sent back to the Caise server and propagated to all other participating instances of the IDE and other Caise-based tools.

The synchronous editing of the same file between multiple users was implemented to the best ability that the Together Architect API allowed. Fine-grained events such as individual keystrokes, however, are not raised by Together Architect's API, which means that source files could only be updated on a per-save basis. Better API support for low level operations, or access to the source code is required if Together Architect is to be integrated as a fully-synchronous Caise-based tool. Using social protocols to moderate concurrent file access, however, allowed the IDE to be used successfully within the current suite of Caise-based CSE tools.

By the time of writing, the Eclipse IDE [83] had matured into a powerful and widely-used alternative platform for Java development. Therefore, Eclipse is an ideal candidate tool to incorporate within the Caise framework, especially given its extensive APIs and full access to source code.

The Together Architect IDE operating as a Caise-based tool is presented in Figure E.1. This uses the services of the Caise framework to provide information about what other users are doing in the project. The Change Graph (A) keeps track of cumulative remote user modifications. The User Tree (B) displays the current location of each developer relative to the project's semantic model. Together Architect's message panel (C) is used to display feedback information related to user proximities and impact reports.

The Caise-based version of Together Architect (A) working alongside a Caise-based text editor (B) within the desktop of a single user is presented in Figure E.2. Both tools are being used to edit and view the same source

www.manaraa.com

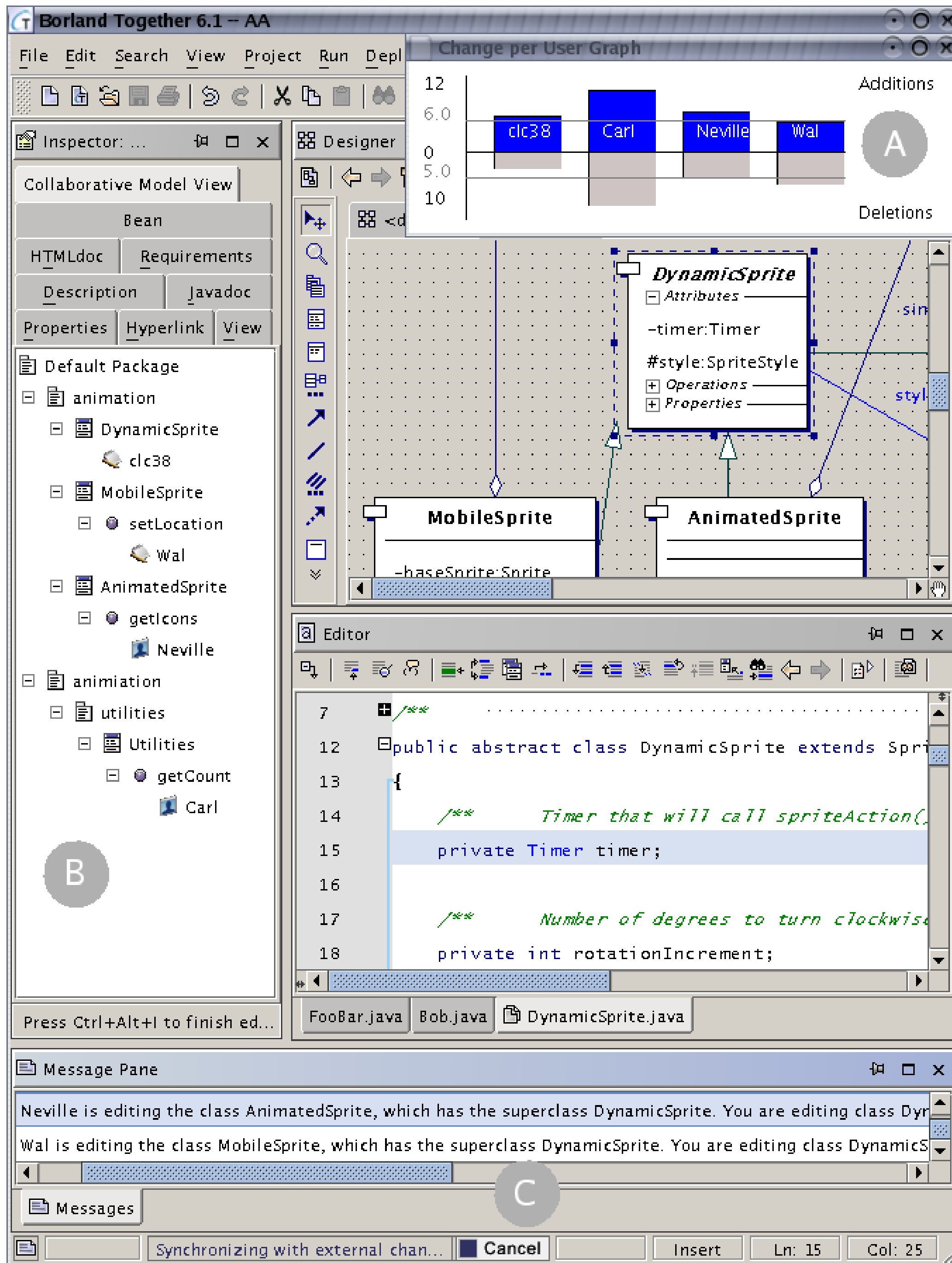


Figure E.1: The Together Architect IDE operating within CAISE.

المنارة للاستشارات
www.manaraa.com

file, and the User Tree in the bottom left segment of the IDE window keeps track of current user locations.

The CAISE-enabled version of Together Architect presents a comprehensive test of the strength and viability of the CAISE framework. By using an industrial-strength tool such as Together Architect, it is illustrated that CAISE is suitable for commercial tool use, and can accommodate multiple users joining and leaving at any time.

A key aspect of incorporating Together Architect within the CAISE framework is that it removes the requirement of using a code repository between instances of the IDE within a collaborating group of developers. For software engineers who wish to work together in real time using professional tools, the CAISE framework provides a mechanism for doing so on the premise that programmatic access to such tools exists.

المنارة للاستشارات
www.manaraa.com

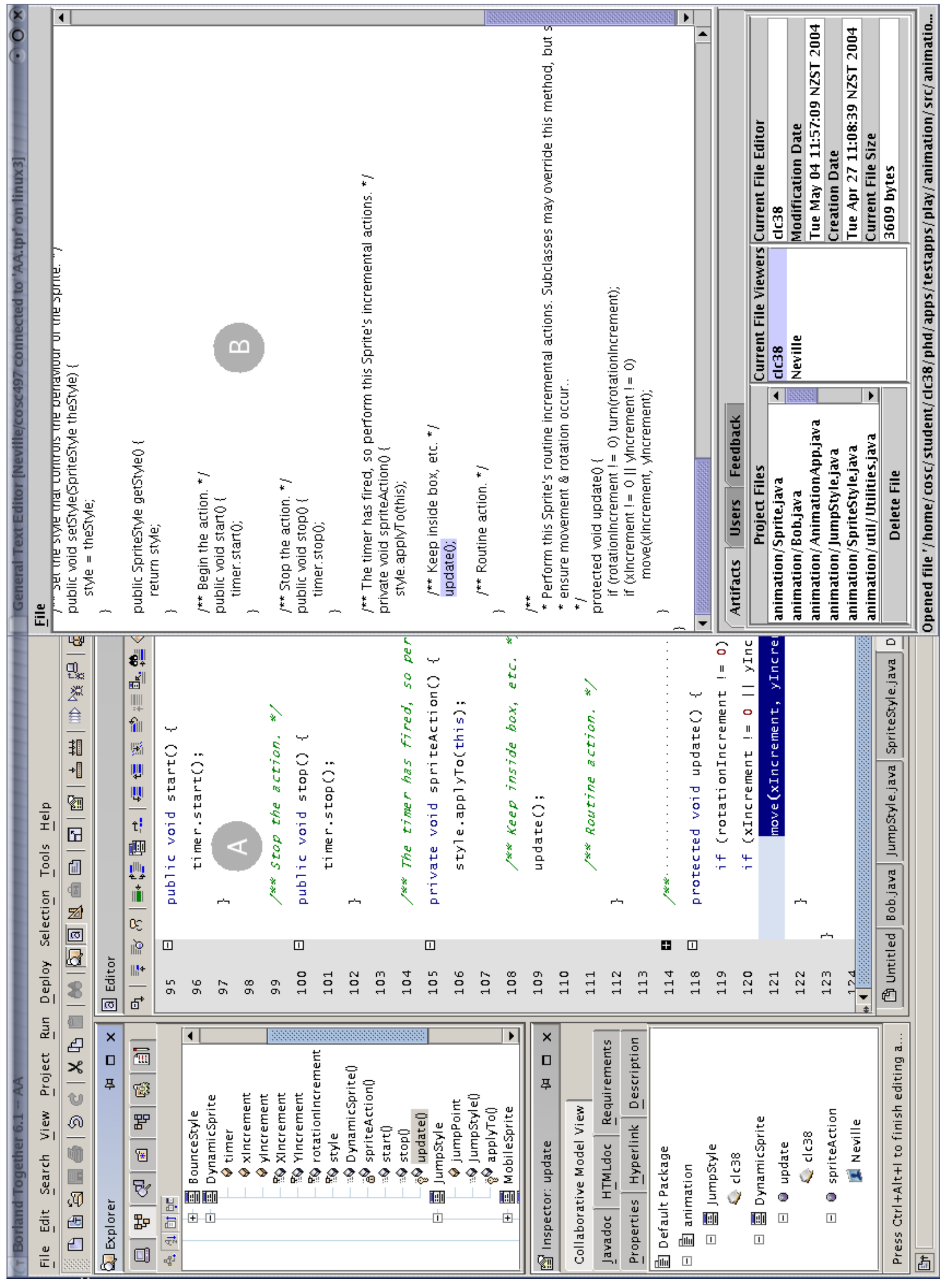


Figure E.2: The Together IDE operating alongside a CAISE-based editor.

المنارة للاستشارات
www.manaraa.com

Appendix F

User Evaluation Design

This appendix gives full details of the user evaluation presented in Section 7.3, including the methodology, environment configuration and results. It is intended that this appendix provides enough details so that the experiment can be reproduced by other researchers for comparison against other systems and types of users.

F.1 Overview

It is a challenging task to design a valid SE experiment of any kind. An example of the detail that must be addressed is given elsewhere [14]. For this reason many software engineers leave the task of empirical evaluations to that of other disciplines within computer science, where the number of variables to address is fewer and the difficulty of isolating them is considerably less.

To design a credible SE experiment, the first aspect is to determine precisely what it is to be measured: task completion times, software quality and bug rates, robustness and quality of design, and other subjective measures such as perceived effort and frustration. Following that, it may be necessary to either isolate or explicitly control independent variables such as the scope of the task, participants' familiarity with the tool set, team size and individual roles, and the type of task being performed. Additionally, confounding factors such as programmer abilities and learning effects need to be addressed. Without isolating the independent variables and addressing confounding factors, a vast number of variables could affect and distort any findings.

Given that it is possible to identify a dependent variable, isolate and control the independent variables, and remove all confounding secondary factors, there are still two considerable issues to consider for SE experiments: is the experiment still at a level realistic enough to show an effect that is globally useful; and if an effect is observed, is it possible to claim causality rather than just a correlation. An even harder aspect to consider within CSE research is the types of systems to compare CSE tools against in order to provide an objective and useful comparison.

www.manaraa.com

F.2 Aim and Purpose

A concern shown elsewhere is that programmers do not use collaborative systems as much as they can and arguably should [51]. The experiment detailed here aims to demonstrate that a set of real time collaborative tools not only provide a more efficient alternative to concurrent program editing with code repository systems, but participants also prefer using the new tools.

There are many perceived benefits of using CSE tools: faster task completion rates, greater levels of team efficiency, greater understanding of local and remote changes, less or no delay between file updates, fewer or no merge conflicts, and higher levels of communication between programmers. These are all anecdotal claims however; very little empirical research has been conducted to support these claims.

Only a small number of claims can be asserted in any one trial. Therefore, the primary goal of this experiment is to illustrate faster task completion times using collaborative tools when compared to an equivalent set of tasks using conventional code repository practices.

As a subjective measure, this experiment also surveys perceived levels of change understanding, frustration, success and effort for both modes of tool operation.

Finally, users are asked how much they envisage using such CSE tools in a range of settings. It will be most beneficial to discover if the participants embrace or dismiss the concepts behind the tools. It has been a fear that even though the tools may appear superior from a design perspective, 'real' users will not like them regardless of the actual efficiency levels. Empirical evaluations of other tools have not always produced results that match the researchers' expectations [19].

An additional outcome of this experiment, if completed satisfactorily, is that an assertion can be made that the tools are robust enough and appropriately designed to accommodate use by complete newcomers.

If the tools reduce task completion times and are favoured by the users, this provides confirmation that their design, implementation and user interfaces are at least satisfactory in terms of suitability for broad-scale CSE.

F.3 Evaluation Methodology

An overview of the evaluation methodology was given in Section 7.3.1. In this section, specific details are given.

As explained in Section F.2, it is difficult to design an evaluation where a fair comparison of conventional and collaborative tools can be made. Details are given here of an experiment that negates as many confounding effects as

www.manaraa.com

possible for a set of realistic programming tasks, and isolates the dependent variable of task completion rates for objective measurement.

It is important to emphasise that core speed in terms of task completion rates are being investigated. To allow this, equalisation of effects must take place such as programmer ability, physical and mental effort of tasks, task types and scope, and effects of different tools within the evaluation. By isolating these effects, the evaluation may appear somewhat mechanical, but a fair measure of the core comparative speeds of the tools is possible. A number of external factors will affect the overall efficiency and effectiveness of collaborative tools, but the experiment will still provide a useful and reliable insight of the CSE tools.

F.3.1 Participants

For this experiment, 12 postgraduate Computer Science students were used, which represented the entire class for an advanced OO design course. By selecting this class, it was assured that each participant had an interest and experience in SE. All participants had at least minimal operational experience in source code repositories and group work. The students possessed grade point averages ranging from satisfactory to excellent. All participants were male with an even spread of ages from 21 to 30, which is an approximately representative sample of the professional software development population.

After selecting the participants, they were left to organise themselves into groups of two. Upon formation of the six groups, most pairs had worked with each other in some SE context over the last two years.

F.3.2 Physical Layout

Each evaluation session involved a pair of participants, with the layout of participants and equipment presented in Figure F.1. As the physical location of participants had the potential to alter the task completion rates, the environment for the evaluations was kept constant for the duration of this experiment. For this experiment, the configuration of the environment was designed to be representative of a typical co-located programming setting.

The participants worked within two meters of each other, but the monitors were not in direct line of sight of each other. In this configuration, the participants were able to directly observe and possibly circumvent the activities of each other, but only if they made the conscious effort to draw themselves from their own work. They were at all times able to communicate with each other orally without impairment. Finally, the experiment was held in an isolated office without any risk of interruption or interference.

المنارة للاستشارات
www.manaraa.com

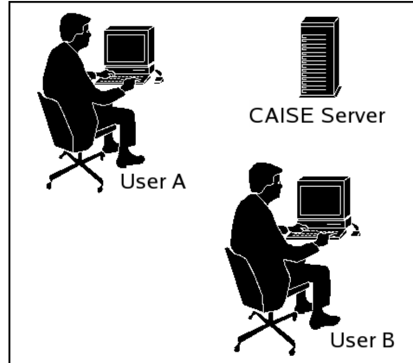


Figure F.1: Evaluation layout of CSE tools experiment.

F.3.3 Apparatus

Both the desktop workstations and the CAISE server ran the Fedora Core 3 operating system with the Linux 2.6.9-SMP kernel. The CAISE framework was compiled with Sun's Java 1.5.0 Standard Edition compiler and executed with the corresponding Hot-spot virtual machine.

The desktop workstations were 32-bit Dells with a single Intel Pentium-4 2.8GHz CPU. The CAISE server was a 64-bit ASUS machine with dual AMD Opteron 2.0 GHz CPUs. All machines had 2 Gb of primary memory and 10 Gb of swap space.

The desktop workstations hosted the standard Gnome X desktop running at 1600 by 1200 resolution. On each machine ran a local instance of the CAISE server process and the two CAISE tool applications. No other applications ran on the workstations apart from the core Linux services. The server ran without a display, again running only the core Linux services and the CAISE server process.

The communication medium between the participating machines was a 100 Mbps switched Ethernet network, dedicated for the use of the participants' desktop machines and the CAISE server.

F.3.4 Supporting a Minimal Code Repository Interface

For this experiment, it was very important to make interactions between the source code repository and the client tools as simple as possible. If the interface to the repository was cumbersome or complicated, it would greatly skew the task completion rates that are being measured.

CVS was used as the underlying repository, but the users were not aware of this technicality. For the CAISE-based tools, CVS was encapsulated simply

المنارة للاستشارات
www.manaraa.com

as a high level and generic code repository. This is similar to the simple manner in which a Wiki Web supports different versions of files, even though a complex code repository system is employed on the Wiki server.

Advanced users of CVS and other source code configuration systems will know how to use its features to avoid potential merge conflicts and transactional errors [51]. This experiment is not designed for advanced users however, although this type of user is addressed in Section 7.3.3. Therefore, participants in this experiment require only an average programming ability and a simple command of a code repository system.

For this evaluation, code repository support was incorporated into the Caise-based tools. Accordingly, when the tools are started in conventional mode, they keep all changes to files isolated from other users. File modifications can be exchanged through a code repository menu. This menu is presented in Figure 8.2.

The code repository was designed to be a simple as possible to use, and accordingly as fast as possible. Safeguards were also required to ensure that participants did not make errors when synchronising their source files. Many users struggle with systems such as CVS; typical problems include forgetting to download and work on the latest version of the repository, forgetting to upload new or modified files back to the repository, and checking-in a subset of files that build locally but will break the repository's version of the program. A syntax directed interface to the repository was implemented, as discussed in Section 8.1.1, where only valid repository actions could be made given a set of local files and a central code repository.

It took many prototype design sessions and pilot studies to create the code repository interface presented here. This type of syntax directed interface is a highly appropriate mechanism to support making a comparison between conventional and collaborative SE. It allows comparison of the essential differences between the two modes of work, and mitigates factors such as the time it can take to type in code repository commands or to navigate through a cumbersome repository interface, determining the correct code repository commands for the current state of the local files and the global repository, and the imbalance of repository experience levels between participants.

F.3.5 Tool Modes

As this experiment compared conventional and collaborative SE task completion rates, the set of Caise-based tools were required to run in both collaborative and conventional modes. In this manner, as long as the code repository interface is minimal, there should be no confounding factors in terms of tool type—the participants can use the same tool for both tasks, with very little practical difference between the two tool modes.

المنارة للاستشارات
www.manaraa.com

When the tools were operating in collaborative mode, a central server was responsible for supporting communication between all participating tools and for keeping code synchronised in real time between participants. In conventional mode, the central server was not employed; a local instance of the server process was used to maintain code at the scope of each individual workstation. To synchronise the code modifications between participants, the code repository interface would access a CVS server on a local network file system partition.

The learning effects of individual tool modes is addressed in Section F.3.7. Additionally, to keep the workstation memory loads constant between tool modes, a local copy of the server process ran on each workstation in collaborative mode, albeit redundant.

F.3.6 Task Types

This evaluation assessed two types of task completion rates. One was for task completion rates of between files changes; the other was within files changes. CSE is normally a combination of both types of tasks, but both cases were needed to be treated separately in order to remove any effect of interaction on task completion rates.

All tasks within both sets were designed to generate some sort of conflict between the two participants. For between-files tasks, a *transactional* conflict would occur, meaning there is a problem with the program semantics as a result of the concurrent modifications. A transactional conflict is one where the syntax of the changes is legal, but a semantic error would result once both participants' modifications were synchronised. For example, one user might rename a method while the second participant would make a new call to the method by the original name. Only when the files are synchronised and the resulting code is rebuilt will the error be exposed.

For between-files tasks, a *merge* conflict would result after each participant had made their change and synchronised their code, meaning that there is a problem with the program's syntax as a result of the concurrent modifications. A merge conflict results from overlapping modifications to separate local copies of a source file; when the code repository system attempts to synchronise the changes from multiple users it fails because of there is no deterministic way of forming a final, conflict-free version of the file.

As an example of a merge conflict, one participant could be editing a sequence of statements within a method so that instead of evaluating several complicated conditionals, the code is refactored as a more comprehensible switch/select block. At the same time, the other participant might be editing a second copy of the original file so that one of the conditionals is simplified syntactically. In this case, most code repository systems would give a merge

www.manaraa.com

conflict error where it is up to the participants to resolve the merge conflict manually and resubmit the final version to the code repository.

F.3.7 Order of Groups and Tasks

The order of groups in which the evaluations were held and the order of tasks that each group performed are presented in Table F.3.7 (Cv stands for conventional mode, Cb stands for collaborative mode. T1 and T3 are between-files tasks, T2 and T4 are within-files tasks). Careful consideration was given to the design of task ordering between groups; the main objective was to negate or minimise any learning effect of tool mode and task type.

Group	Task Configuration				Order			
1	CvT1	CvT2	CbT3	CbT4	1	2	3	4
2	CvT1	CvT2	CbT3	CbT4	4	3	2	1
3	CvT1	CvT2	CbT3	CbT4	1	3	2	4
4	CbT1	CbT2	CvT3	CvT4	4	2	3	1
5	CbT1	CbT2	CvT3	CvT4	3	2	4	1
6	CbT1	CbT2	CvT3	CvT4	1	4	2	3

Table F.1: Task types, tool modes and order of tasks.

Each pair of participants was used for both the treatment and the control group. Separate yet similar tasks were employed for each tool mode; this is the reason why there are two tasks for each task type. By using each group as a treatment and control, any imbalance between individual groups was negated. If one group was exceptionally good or bad at a given task, they were likely to produce the same result for both tool modes.

To reduce the risk of a learning effect on task type or tool mode, each group had a different order of task type and tool mode. Group one, for example, first did both tasks in conventional mode and then collaborative mode. Group two did both tasks in collaborative mode first, followed by conventional mode. From Table F.3.7, task modes were also alternated between groups. If there was any learning effect from task type or tool mode, it was likely to be countered by the nature of the group and task assignments.

Since participants acted as both the control and treatment group, the only other confounding factor could come from differences within the sets of tasks. While it may at first seem relatively simple to create two similar tasks for each task type, in practice this was quite challenging to achieve. Ensuring that the tasks were distinct was important to reduce any learning effect, yet the tasks had to be nearly identical in terms of syntax, semantics, typing effort and conflict resolution actions to ensure that the tasks were objectively comparable.

المنارة للاستشارات
www.manaraa.com

In Section F.3.11, analysis of the experiment results shows that there was no significant difference between any pair of task types for each tool mode. Again, if there was any difference within a set of tasks of the same type, due to the design of the group and task order, the impact would be largely negated.

F.3.8 Training Manual

Participants were given a 30 minute training session prior to completion of the evaluation tasks. It is hoped that by giving a thorough training, learning effects of tools and task types were minimised. To assist the training period, a training manual was given to each participant a few days prior to the evaluation session. This gave the participant a chance to gain an overview of the tools and tasks, and allowed him or her to prepare questions for the training session.

The training manual provided the participants with an overview of the code repository system, how to operate the code repository within the evaluation tools, how the real time editor and awareness support components operate, and how the basic editing and compiling functionality works for both tool modes, such as cut, copy, paste, undo, compile and run. An excerpt from the training manual is given in Appendix G.1.1. This appendix also gives details on how to obtain a full electronic copy of the training manual.

Training Tasks

Within the training manual there were four mechanically-scripted tasks to complete. Two of the tasks are conflict-free, the remaining two contain inevitable conflicts. Two of the tasks involve within files changes, the remaining two involve between files changes. The four tasks are performed by each pair of users firstly using the tools in conventional mode, and then again with the tools in collaborative mode. An excerpt from the training tasks sheet is given in Appendix G.1.2.

Answer Sheet

Each conflicting task within the training session has a prescribed resolution. When the participants encounter a conflict, they were instructed to refer to the training tasks answer sheet for the correct resolution. The answer sheet simply details which parts of the code need to be replaced, and what these lines of code need to be replaced with. Upon correct resolution of the conflicting code changes, the program should compile again, and the task is then considered to be complete. An excerpt of the training tasks answer sheet is also given in Appendix G.1.2.

www.manaraa.com

F.3.9 Evaluation Tasks

The evaluation tasks were again mechanically scripted for each participant, with check-boxes on the manuscripts to help prevent participants from skipping instructions or performing operations in an incorrect order. As in the training tasks, an answer sheet was provided to resolve the resultant conflict from the two sets of changes. An excerpt of the evaluation tasks sheet and answers is given in Appendix G.1.3.

Each participant worked as fast as they could on their set of instructions, and the first group member to complete his or her work, including recompiling the code and verifying that the application still worked properly, could submit his or her updated files to the code repository first and not have to deal with any potential transactional or merge conflicts. In all cases, the participant who finished his or her tasks second had the duty of correcting the now exposed conflict. For the tasks that were performed in collaborative mode, the issue of which participant did the code correction was determined by whoever discovered the conflict.

F.3.10 User Survey

A survey was given to each participant to complete in private at the end of each task. The aim of the survey was to provide a comparison of the tools running in both modes of work for each task type.

For this survey, NASA-TLX questions [52] were used to determine and compare the perceived effort, success and frustration levels of each participant. By using the standard NASA-TLX questions, results are made available for comparison against any other related studies that use the same survey technique. Additional subjective questions were also asked, related to the understanding of code changes and the perceived ease of file control.

A survey was also given at end of each session. This was purely for feedback on the underlying concepts of the Caise framework, such as did the participant like the concept of real time code sharing and editing, and would the participant consider using such a system if it was made available outside of the evaluation.

The questions and aggregated responses for both surveys were given in Section 7.3.2, and excerpts of the surveys are given in Appendix G.1.4.

F.3.11 Statistical Validity

The use of statistics must be valid and justifiable before the results are analysed and reported. Aspects to consider when looking at the statistical validity of empirical SE tasks include the choice of statistical test, design of the

المنارة للاستشارات
www.manaraa.com

experiment to eliminate confounding factors, and post-evaluation analysis of the data to ensure it fits with the test.

One-way analysis of variance (ANOVA) was selected for this evaluation. Tests are designed to detect any statistically significant difference between the sample means, where separate tests are conducted for within files and between files tasks. In the case of this evaluation there were only two means to compare, so a two-sample t-test could have been used to give identical results.

Interactions (two-way ANOVA) of tool mode and task type were not investigated. This would be an interesting aspect to explore, but it is not the focus of this study. While it can not be ruled out that there could be some interaction between the task type and tool mode, the evaluation presented here specifically isolated each task type.

From literature related to the use of empirical statistical analysis [54], it is safe to assert a statistically significant, valid and meaningful difference between two means if:

- The power of the test is not too high. A high power test is susceptible to asserting that a negligible difference between two means is statistically significant

- All samples are a simple random survey (SRS) of the population, where the population follows a normal or near-normal distribution. This also implies that the samples should follow normal or near-normal distributions with similar standard deviations to each other

- The sample sizes are the same or similar to each other

- The measures of both samples are independent of each other

- There is no bias in the experimental design

The evaluation presented in this appendix has not breached any of the above guidelines, and therefore the results of the statistical tests are significant, uncompromised and applicable to the field of CSE research. Justification of this claim is provided in the remainder of this section.

Design of Trial

A common criticism of statistical tests is that unless the number of observed values is large, the results are not valid due to the low statistical power of the test. This criticism is only valid when asserting similarities between a set of means, not differences. For the evaluation presented in this thesis, attention

المنارة للاستشارات
www.manaraa.com

is focused on finding a statistically significant difference between the task completion rates for the two tool modes; if any difference is found then it is a valid difference.

To discover a difference is a challenging task, however. Means are required that are distant from each other, with standard deviations small enough that they do not overlap significantly. To achieve both of these characteristics from the data, normally a large, high power sample is required to reduce the standard deviation size, or data is required from samples that genuinely are from populations with well separated means.

For this evaluation the sample sizes were all the same within each statistical test. Additionally, it is reasonable to claim that the pool of participants was representative of the population, and can be considered as a SRS. This is discussed further in Section 7.3.3.

If the two tool modes were tested on the entire population, an approximately normal distribution of completion rates would be expected—most users would complete the tasks near the population mean, with a decreasing number of outliers either side of the mean. In other words, no skew or flatly uniform distribution is expected if the entire population were to be sampled.

It can be safely asserted that the task completion rates taken from both samples were independent of each other. In the case of this evaluation, the two samples actually consisted of the same set of participants, but being examined under different tool modes. As long as the learning effect was negligible, then independent measures could be assumed.

As discussed previously, strong steps have been taken to eliminate or reduce any bias within the experimental design. Potential sources of bias include learning effects on tool mode and task type, but methods have been introduced to eliminate this. Steps have also been taken to remove any other confounding factors such as programmer ability and scope of tasks by isolating and mechanising the experiment as much as possible. Another common source of bias in evaluations is where participants are self-selected. This risk was eliminated by ensuring that the entire class took part in the evaluation, not just the students who showed interest.

It was also important to have a working implementation of collaborative undo for the tools used in this experiment. Without such undo facilities, a mistake could be very costly to correct, which would confound the task completion rates and would also be likely to negatively affect the participants' survey answers. To implement collaborative undo, where the local user's changes in a file are treated differently from all remote users, is an extremely challenging task, as discussed previously in Section A.3.2.

المنارة للاستشارات
www.manaraa.com

Post-Test Data Analysis

To confirm statistical correctness, post-data analysis was also performed. After completing the evaluations and collecting the raw task completion rates and survey responses, it was possible to verify the assertions of normally distributed samples and equivalent sample standard deviations.

The first step in any post-data analysis is to plot the results and confirm that the distribution looks normal and the standard deviations are also of approximately the correct magnitude. In the case of this experiment the data for both the objective measures and the subjective measures appeared satisfactory.

To formally test for equivalence between standard deviations, the rule

$$max(s.d.) <= 2 \times min(s.d.)$$

is often followed [75]. All statistical tests conformed to this rule for task completion rate comparisons. As it was desirable to test for significant differences within the survey questions as well, the survey results were also checked statistically. All but three of the twelve survey tests for statistical differences passed this rule. For the three tests that failed in terms of having equivalence, the p values were all so small that it is safe to assume that the results were still significant in determining a statistical difference [75].

A final concern that could be dismissed by statistical investigation was that of unfair variance within tasks of the same type. As the experimental design required two unique tasks for each type, it was important to ensure the completion times were similar for each task within both tool modes. If no significant difference is found in times between both tasks within each task set, this reduces speculation of a confounded experiment due to non-equivalent tasks. While any disparity between tasks is negated by the order of the groups and tasks, it is beneficial to assert that there is no disparity in the first instance.

The hypothesis is that there is no difference between the means of the groups that completed the two different tasks for each given task type and tool mode. For all one-way ANOVA tests, the F statistic, which is the ratio of variance between and within groups, is computed. The probability that this F value would occur if the two means truly were the same is checked. When testing the F statistic, this value is compared to the $F(I-1, N-I)$ distribution, where I is the count of groups and N is the count of all samples taken. The null hypothesis that the means are equal is rejected if the F test statistic is too large in comparison to the critical value of the F distribution for the corresponding degrees of freedom (I and N).

When it is said, for example, that a $F_{1,4}$ statistic of 0.13 with a p value of 0.74 has been computed, this implies that for a measure of two groups with

www.manaraa.com

six samples in total, it is expected that no real difference is detected 74 times out of every 100 trials if the means were truly equal. This high likelihood of detecting no difference reflects two distributions that are centered around a similar mean. Alternatively, the F test statistic of 0.13 is considerably lower than the critical value of 2.42 for $F_{1,4}$ at the 5% significance level.

After performing the test, it was not possible to show differences between any of the means within a set of tasks for a given tool mode. In collaborative mode, the between files test gave $F_{1,4}$=1.81, p=0.25 and the within files test gave $F_{1,4}$=0.13, p=0.74. In conventional mode, the between files test gave $F_{1,4}$=4.69, p=0.10 and the within files test gave $F_{1,4}$=0.37, p=0.58. This gives evidence that there may not be any difference between tasks for a given task type and tool mode, as suggested, but to claim outright no significant difference with a test of such a low power would be considered unwise.

Simple Random Sampling

Another aspect that is open for discussion for many evaluations is that of assuming the trial group is in fact a SRS of the global population. This judgment can be made by software engineers, statisticians, or perhaps more suitably both groups together. A statistical purist might argue that a SRS has not been made in the case of this evaluation, as the entire population of the class has been sampled. Alternatively, a software engineer can argue that this class is a SRS from the population of typical every-day software engineers. *Typical* programmers are hard to define, but experienced and competent SE students are probably a suitable average.

Summary

The experimental design took considerable effort, with duties including the production of a precise and unambiguous training manual and task sheet, formation of a correct evaluation methodology and plan, and verification of statistical validity. Two similar yet distinct conflicting tasks for both within files and between files experiments also had to be derived, and numerous pilots of the evaluation were undertaken to ensure that the session plan ran smoothly. Accordingly, it is envisaged that this experimental design can be replicated to save the time of others, perhaps even using the same set of tasks. Additionally, by using the same experimental design and set of tasks in other studies, an objective comparison between different tools can be made.

المنارة للاستشارات
www.manaraa.com

Appendix G

User Evaluation Documents

G.1 *Evaluation Documents*

The full task sheet, training manual, answer sheets and questionnaires are available from the accompanying resources disc. Excerpts are given here.

G.1.1 Training Manual

Excerpt from the introduction

Basic Training Manual for the SEVG's Collaborative Software Engineering Tools Evaluation

Preliminaries:

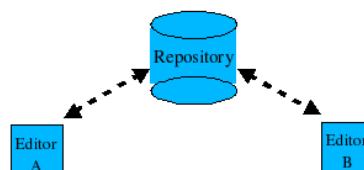
- The upcoming evaluation is to test our software, not your ability to program.
  - We are interested in the task completion rates and your feedback related to the tools.
  - The tasks are very simple, and require no actual software engineering experience or ability. Each task will be mechanically scripted for each user in both the training and evaluation session.
  - There are many aspects of the tools and underlying framework that need to be evaluated. For this evaluation, however, we only have time to look at task completion rates for specific types of scripted tasks. In future evaluations we will address usability issues, programmer comprehension, levels of inter-programmer communication and interaction, levels of software quality, etc.

Excerpt from the CVS interface section

Conventional mode and the code repository

When the tools are running in conventional mode, source files are only shared between the tools on your desktop. To share your source files with your other team member, you will need to use the code repository system (which is built into the text editor).

The following is a brief description of the code repository system, but *you don't need to memorise this or fully understand how it works in detail.* **We will shortly go through some examples which use the code repository, which should provide you with the basic working knowledge to operate the repository for this evaluation.**



المنارة للاستشارات
www.manaraa.com

G.1.2 Training Tasks

Excerpt from a conventional between files training task

Task 1: *Working on two separate files*

User A opens the file `Foo.java`
User B opens the file `FooTester.java`
User A:

- Renames the method `Foo.saySomething()` to `Foo.sayAnything()`
- Replaces the method call to `saySomething()` with `sayAnything()` from within the method `sayHello()`.
- User A then selects Upload, posting the final version of the file back to the repository. He then closes his file.

User B:

- Add a new call to `foo.saySomething("hiya")` from `FooTester.test()`.

User B now can not select Upload, so selects Download instead

- This downloads the latest version of the files that the other user has been working on

User B compiles his code:

- A transactional conflict is discovered.

User B then consults the answer sheet (task 3) to find out the resolution for the problem.
When user B is happy with the code again, he selects Upload and closes his file.
At this point, User A could select Download to bring his files back in sync with the latest version.

Excerpt from the training tasks answer sheet

Task 1

In `FooTester.test()`:

rename:

```
foo.saySomething("hiya");
```

to:

```
foo.sayAnything("hiya");
```

Task 2

In `FooTester.test()`:

The conflicting statement should be merged as:

```
Foo myFoo = new Foo(Foo.MAORI);
```


المنارة للاستشارات
www.manaraa.com

G.1.3 Evaluation Tasks

Excerpt from a conventional within files evaluation task

Task 1 [Conventional]

- User A
  - Open the file `AnimatedSprite.java` and locate the method `AnimatedSprite.advanceImage()`
  - In the method `advanceImage()`, change the conditional:
    `iconList.size() > 0`
    to:
    `!iconList.isEmpty()`

❒ ***Build and run your code to make sure the program still works.***

❒ ***When ready, upload your code to the repository.***

- ***If the upload option is not available, this means that your partner has already uploaded newer versions of the source files to the repository. In this case:***

  ❒ ***Select download to obtain the latest version of the source files (this will automatically update your copies of the source files with the latest versions held in the repository)***

  - ***If you encounter merge conflicts during the download:***

    ❒ ***Fix the merge conflicts by referring to the help sheet***

  ❒ ***Compile your code again to make sure that there are no problems***

  - ***If there are build problems:***

    ❒ ***Fix the problem by referring to the help sheet***

  ❒ ***Upload your code to the repository***

  ❒ ***Ask your partner to download the repository again***

❒ ***Close the file within the editor once finished***

Excerpt from the evaluation tasks answer sheet

Task 2

In `AnimationApp.makeAnt()`:

The method call should be changed to:

```
ant.getSimpleSprite().setBoxed(true);
```

Task 3

In `AnimatedSprite.advanceImage()`:

The conflicting statements should be merged as:

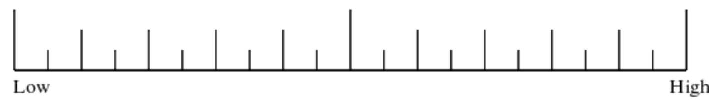
```
if (!iconIterator.hasNext())
    iconIterator = iconList.listIterator();
```


المنارة للاستشارات
www.manaraa.com

G.1.4 Surveys

Excerpt from the end of task survey

2.) How thorough was your understanding of the location of other users, and what other users were doing in the project?



3.) How easy was it to manage the source files amongst the other users?

Excerpt from the end of session survey

2.) In a collaborative, co-located setting, how useful do you think this type of system will be?

3.) How much does it help to have the source code shared and managed for you?

G.2 Source Code

The Java source code for both the training and the evaluation applications are also available from the resources disc or can be downloaded from `www.cosc.canterbury.ac.nz/clc/cse`

المنارة للاستشارات
www.manaraa.com

Appendix H

Accompanying Resources

A compact disc has been compiled that contains full versions of various articles referenced within this thesis. The disc is located in an envelope inside the back cover.

The disc contains:

- A copy of this thesis, in pdf format
- Copies of all the papers published as a result of work from this thesis, in pdf format
- Demonstrations of the CAISE-based tools, embedded within a web page in Motion Network Graphics (MNG) format. A plug-in to support MNG viewing within Internet Explorer is also provided
- The user evaluation documents and sample applications, as presented in Appendix G
- A user manual for the CAISE framework, including API documentation for tool creation and framework extension
- The CAISE Event Log DTD

These resources are also available online from `www.cosc.canterbury.ac.nz/clc/cse`

المنارة للاستشارات
www.manaraa.com