

Software development processes and software quality assurance

Zsolt Dr. Ulbert

Copyright © 2014 University of Pannonia



A tananyag a **TÁMOP-4.1.2.A/1-11/1-2011-0042** azonosító számú „*Mechatronikai mérnök MSc tananyagfejlesztés*” projekt keretében készült. A tananyagfejlesztés az Európai Unió támogatásával és az Európai Szociális Alap társfinanszírozásával valósult meg.

Nemzeti Fejlesztési Ügynökség
www.ujszechenyiterv.gov.hu
06 40 638 638



A projekt az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósul meg.

Manuscript completed: February 2014

Language reviewed by: Bódis László, Kismódi Péter

Published by: University of Pannonia

Editor by: University of Pannonia

2014

Software development process and software quality assurance

Table of Contents

Preface.....	7
1 Introduction.....	8
1.1 Software.....	9
1.2 Software process and software process models.....	9
1.3 The attributes of good software.....	10
1.4 Exercises.....	11
2 System design.....	12
2.1 System engineering.....	12
2.1.1 System requirements definition.....	13
2.1.2 System architecture design.....	13
2.1.3 Sub-system development.....	14
2.1.4 Systems integration.....	15
2.1.5 System evolution.....	15
2.2 Exercises.....	15
3 Software processes.....	17
3.1 Software process models.....	17
3.1.1 The waterfall model.....	17
3.1.2 Evolutionary development.....	19
3.1.3 Component-based software engineering.....	20
3.2 Process iteration.....	20
3.2.1 Incremental delivery.....	21
3.2.2 Spiral development.....	22
3.3 Process activities.....	22
3.3.1 Software specification.....	23
3.3.2 Software design and implementation.....	23
3.3.3 Software validation.....	24
3.3.4 Software evolution.....	24
3.4 The Rational Unified Process.....	24
3.5 Exercises.....	26
4 Agile methods.....	28
4.1 Extreme programming.....	31
4.1.1 Testing in XP.....	34
4.1.2 Pair programming.....	35
4.2 Scrum.....	35

4.2.1	Scrum roles.....	36
4.2.2	Scrum artifacts.....	37
4.2.3	Scrum events	37
4.3	Feature driven development (FDD).....	39
4.3.1	Milestones	40
4.4	Exercises	41
5	Object-oriented development	42
5.1	Generalisation	44
5.2	Association	45
5.3	Object-oriented design process.....	46
5.3.1	System context and models of use	47
5.3.2	Architectural design	47
5.3.3	Object and class identification	47
5.3.4	Design models	48
5.3.5	Object interface specification.....	49
5.4	Exercises	49
6	Unified Modelling Language	50
6.1	Model, modelling.....	51
6.2	Diagrams in UML.....	51
6.3	Business models	52
6.3.1	Business use case diagrams.....	54
6.3.2	Business analysis model.....	58
6.4	Exercises	62
7	Requirements analysis.....	63
7.1	Analyse the problem.....	64
7.1.1	Definition of glossary	64
7.1.2	Find actors	65
7.1.3	Develop requirements management plan	65
7.1.4	Develop Project Vision document.....	66
7.2	Understand stakeholder needs	66
7.2.1	Elicit stakeholder requests.....	66
7.2.2	Find use cases	67
7.2.3	Manage dependencies	68
7.3	Define the system	68
7.4	Manage the scope of the system	69

7.5	Refine the system definition	69
7.5.1	Details of use cases	70
7.5.2	Modelling and prototyping user-interface	71
7.6	Manage changing requirements	71
7.6.1	Structure the use case model	71
7.6.2	Review requirements	74
7.7	Exercises	74
8	Analysis and design	75
8.1	Define a candidate architecture	76
8.1.1	Architectural analysis	76
8.1.2	Use case analysis	78
8.2	Refine the architecture	88
8.2.1	Identify design mechanisms	88
8.2.2	Identify design elements	89
8.3	Analyze behaviour	89
8.4	Design components	90
8.4.1	Use case design	90
8.4.2	Sub-system design	90
8.4.3	Class design	91
8.5	Implementation	96
8.6	Deployment	96
8.7	Exercises	97
9	Software testing	98
9.1	Unit testing	98
9.1.1	Interface testing	99
9.2	System testing	99
9.2.1	Integration testing	99
9.2.2	Functional testing	100
9.2.3	Performance testing	101
9.3	Exercises	101
10	Embedded system development	102
10.1	Critical systems	103
10.1.1	System dependability	103
10.1.2	Safety	104
10.1.3	Security	105

10.2	Critical systems development	105
10.2.1	Fault tolerance	106
10.3	Real-time software design.....	107
10.3.1	System design.....	108
10.3.2	Real-time operating systems	109
10.4	Exercises	110
11	Project management	111
11.1	Management activities	111
11.2	Project planning	111
11.2.1	The project plan.....	112
11.3	Project scheduling.....	113
11.4	Risk management.....	113
11.4.1	Risk identification	114
11.4.2	Risk analysis.....	115
11.4.3	Risk planning.....	115
11.4.4	Risk monitoring.....	116
11.5	Exercises	116
12	Software quality management.....	117
12.1	Process and product quality	117
12.2	Quality assurance and standards	118
12.2.1	ISO	118
12.2.2	Documentation standards	119
12.3	Quality planning.....	119
12.4	Quality control	119
12.4.1	Quality reviews	120
12.5	Software measurement and metrics	120
12.5.1	The measurement process	120
12.5.2	Product metrics.....	121
12.6	Exercises	121
13	Software cost estimation	122
13.1	Software productivity estimation.....	122
13.2	Development cost estimation techniques.....	123
13.3	Algorithmic cost modelling	124
13.4	Exercises	124
14	References	126

Preface

In the last decades we have been witness to the rapid development in computer technology. Due to the technological development in production of hardware and software tools new challenges have emerged. The dynamic and global economic environment requires the development of new software development techniques that are suitable for producing high quality software meeting requirements of the business environment and high level of professional standards. The new approaches to software development techniques and methodologies make it possible to use the technique best fits the structure and business objectives of an organization. In addition, the software development processes have to support the development requirements of new types of systems and new applications that control these systems. For example they are embedded systems and their software, mobile systems and mobile applications, safety-critical systems and dependable system developments, etc. Today, the large number of available methodologies ensures the use of appropriate methodology and the successful completion of software development projects.

Emergence of agile methodologies are very favourable for flexibly and adaptive organizations. However, there are a number of software development projects when conventional development methodologies are preferable to use due to the nature of development task and organization.

Use of object-oriented development principles have become very popular in both system and software development projects. Over the object-oriented programming the object-oriented technology has become an effective development tool in the systems and software development methodologies. This course book deals with the phases of software development process, such as software requirements, analysis and design that are discussed in the frame of the Rational Unified Process (RUP) software development process in several chapters. RUP is based on the Unified Modelling Language (UML) as an object-oriented graphical modelling language, so theoretical knowledge and practical examples presented in this student book reflect the object-oriented design and development principles.

In chapters 1-4. the basic concepts of software development and the conventional and agile software development methodologies are presented.

Chapter 5. and 6. discuss the basic of object-oriented design and UML graphical modelling language. Chapter 7. and 8. describe in detail the requirements, analysis and design software development phases. These chapters are well illustrated with examples using the notations of UML graphical modelling language. In chapter 10. related to development issues of embedded systems the characteristics of the critical systems, development of critical systems, and development of real-time systems are discussed. The topics of the last three chapters are related to software projects management including project management, software quality management and software cost estimation.

Although this student book is about basic knowledge of software development process, the programming is not subject of any chapters, so the processing of chapters do not require any prior programming experience. Processing of chapters related to software development process requires object-oriented design approach, but it is helped by knowledge provided in chapter 5. and 6.

1 Introduction

Computer software has become a driving force in our life, everyone uses it either directly or indirectly. The role of computer software has undergone significant change over the last 50 years. Software affects nearly every aspect of our lives and has become unavoidable part of commerce, our culture and our everyday activities. It serves as the basis for modern scientific investigation and engineering problem solving. It drives business decision making. It is embedded in systems of all kinds: entertainment, office products, transportation, medical, telecommunications, industrial processes, etc. Software as a product delivers the computing potential embodied by computer hardware. It can be considered an information transformer producing, managing, acquiring, modifying, displaying, or transmitting data. Software provides a connection to worldwide information networks Internet and makes it possible to acquire information in all of its forms [1,4,14,15].

As the importance of software grows, the software community continually attempts to develop technologies that will make it easier, faster, and less expensive to build high-quality computer programs. Some of these technologies are targeted at a specific application domain such as web-site design; others focus on a technology domain (for example object-oriented systems); there are technologies that deliver dependable software. However, we have yet to develop a software technology that does it all. The technology encompasses a process, a set of methods, and an array of tools that we call software engineering. Software engineering has been an engineering discipline, it focuses on the cost effective development of high-quality software systems.

The notion of software engineering was first proposed in 1968 at a conference held to discuss what was then called the software crisis. This software crisis resulted directly from the introduction of new computer hardware based on integrated circuits. Dramatic improvements in hardware performance, profound changes in computing architectures, increases in memory and storage capacity, have all led to more complex computer-based systems. The resulting software was orders of magnitude larger and more complex than previous software systems. Due to the high demand for complex software systems the individual programmers of the earlier periods has been replaced by a team of software developers, each focusing on one part of the technology required to deliver a complex application. At large-scale systems developments, that requires extensive co-operation of developers to complete developments within a prescribed duration of time, it has become increasingly apparent that the existing development methods were not good enough to efficiently drive the development processes.

Development projects were sometimes years late. The development cost was over the budget, software was unreliable, difficult to maintain and performed poorly. Hardware costs were consumedly decreased while software costs were rising rapidly. This phenomenon was recognised by software developers and they stated that the software development practice was in crisis. To solve problems it was necessary to recognize that software has become a product, and similarly to other products a technology was required to develop it.

What does it means that software is a product? It is that:

1. Software has to provide services and functionalities as it specified.
2. Software has quality attributes.
3. There is a cost of its production.
4. There is a time limit for developing software.

New techniques and methods were needed to control the complexity inherent in large software systems. These techniques have become part of software engineering and are now widely used. Software industry has made tremendous progress since 1960s and the

development of software engineering has significantly improved the quality of software. The activities involved in software development have been much better understood. Software engineers and development organizations have developed effective methods of software specification, design and implementation. However, the large number of different types of systems and organizations that use these systems means that we need different approaches to software development.

1.1 Software

Computer software is the product that software engineers design and build. A software product is composed of computer programs, data and documents. A definition of software can be given by these items as follows:

1. Software consists of computer programs that when executed provide desired function and performance.
2. It includes data structures that enable the programs to adequately manipulate information.
3. It has a documentation that describes the operation and use of the programs.

There are two fundamental types of software product:

1. *Generic software*. These software are produced by a development organization and sold on the open market to any customer.
2. *Customized software*. These software are developed especially for a particular customer by a software contractor.

The main difference between these two types of software is the following. In the case of generic software products the organization that develops the software specifies the software requirements. For custom products, the specification is usually developed and controlled by the organization that is buying the software.

The breadth of software applications can be indicated by the following software areas: system software, real-time software, business software, engineering and scientific software, embedded software, personal computer software, web-based software and artificial intelligence software.

1.2 Software process and software process models

A *software process* also known as a *software development life-cycle* is composed of a number of clearly defined and distinct work phases which are used to design and build software. Development phases represent four fundamental process activities that are common to all software processes:

1. *Software specification*. During this activity customers and engineers define the functional and non-functional requirements of software to be produced.
2. *Software development*. It is the development activity for designing and implementing software.
3. *Software validation*. In this activity the software is checked that it works failure free and to ensure that it provides the functionalities what the customer requires.
4. *Software evolution*. The aim of software evolution is to implement changes to the system due to the changes in user and market requirements.

These fundamental activities may be organized in different ways and described at different levels of detail for different types of software. Use of an inappropriate software process may reduce the quality of the software product to be developed and may increase the development costs.

A *software process model* is a simplified description of a software process that presents an overall view of the software process. Process models may include activities that are part of the software process, software products and the roles of people involved in software engineering. Most software process models are based on one of the next general models of software development:

1. *The waterfall approach.* The waterfall model is a sequential design process in which progress is seen as flowing steadily downwards through the phases of software development such as requirements specification, software analysis and design, implementation, testing, etc. Development only goes on to the following phase when the preceding phase is finished, reviewed and verified.
2. *Iterative development.* The basic idea of this method is to develop a system iterative through repeated cycles in smaller portions at a time. At each iterations, design modifications are made and new functional capabilities are added. This approach interleaves the activities of specification, development and validation. An initial system is rapidly developed from very abstract specifications. The goal for this initial implementation is to create a product to which the user can react. The initial system is then refined with customer input to produce a system that satisfies the customer's needs.
3. *Component-based software engineering.* It is a reuse-based approach of software development. This technique assumes that parts of the system already exist. The system development process focuses on integrating these parts into a system.

1.3 The attributes of good software

Functional quality of software reflects how well it complies with a given design, based on functional requirements or software specifications. Besides functional quality, software has a number of other associated attributes that also reflect the quality of that software. These attributes are usually called non-functional attributes. They reflect the behaviour of software while it is executing and the structure and organization of the source program. Examples of these attributes are the maintainability, reliability, efficiency, etc. The specific set of attributes expected from a software system depends on its application. Some essential quality attributes of a software system are the following [13]:

- *Maintainability.* Maintainability is defined as the ease with which changes can be made to a software system. These changes may be necessary for the correction of faults, adaptation of the system to a meet a new requirement, addition of new functionality, removal of existing functionality or corrected when errors or deficiencies occur and can be perfected, adapted or action taken to reduce further maintenance costs.
- *Dependability.* Dependability of software includes characteristics such as availability, reliability, safety and security. Dependable software should not cause physical or economic damage in the event of system failure.
- *Efficiency.* Efficiency is the ability of the software to do the required processing on least amount of system hardware such as memory or processor cycles.
- *Usability.* Usability is the ability of a software to offer its interfaces in a user friendly and elegant way.
- *Reusability.* Reusability defines the capability for components and subsystems to be suitable for use in other applications and in other scenarios. Reusability minimizes the duplication of components and also the implementation time.
- *Testability.* Testability is a measure of how easy it is to create test criteria for the system and its components, and to execute these tests in order to determine if the

criteria are met. Good testability makes it more likely that faults in a system can be isolated in a timely and effective manner.

1.4 Exercises

Explain software crisis!

Give the definition of software!

What is the difference between generic and customized software?

What is the software process?

List the main activities of software process!

What does the software process model mean?

List the main types of software process model!

What is the meaning of component based software development?

List five quality attributes of a good software!

What is the meaning of software maintainability?

What is the meaning of software reusability?

2 System design

The term *system* is one that is universally used. *A system is a purposeful set of interrelated components that form an integrated whole and work together to achieve some objective.* Systems are usually hierarchical and so include other systems. These other systems are called *sub-systems*. Some systems share common characteristics, including [1]:

1. A system has *structure*, it contains parts or components that are directly or indirectly related to each other.
2. A system has *behaviour*, it contains processes that transform inputs into outputs.
3. A system has *interconnectivity*, the parts and processes are connected by structural and/or behavioural relationships.
4. Structure and behaviour of a system may be decomposed via sub-systems and sub-processes to elementary parts and process steps.

The *technical computer-based systems* are systems that include hardware and software components. *Socio-technical systems* include one or more technical systems but also include knowledge of how the system should be used to achieve some broader objective. This means that these systems have defined operational processes, include people who operate the system, are governed by organisational policies and rules and may be affected by external constraints such as national laws and regulatory policies.

Software engineering is critical for the successful development of complex, computer based socio-technical systems. Software engineer should not simply be concerned with the software itself but they should also have a broader awareness of how that software interacts with other hardware and software systems and how it is supposed to be used by end users.

The complexity of relationships between the components in a system means that the system is more than simply the sum of its parts. It has properties that are properties of the system as a whole. There are two types of emergent properties of system:

1. *Functional properties*. These properties appear when all the parts of a system work together to achieve some objective. For example, when an alarm system turns on the siren it is due to the co-operation between its components.
2. *Non-functional properties*. These properties are related to the behaviour of the system in its operational environment. Examples of non-functional properties are reliability, performance, safety and security for computer-based systems.

2.1 System engineering

Systems engineering focuses on how to design and manage complex engineering projects over their life cycles. It includes the activity of specifying, designing, implementing, validating, deploying and maintaining socio-technical systems. Systems engineers are not just concerned with software but also with hardware and the interactions of a system with users and its environment. Important aspects of system development that should be addressed the services that the system provides, the constraints under which the system must be built and operated and the way in which the system is used to fulfil its purpose. The stages of the systems engineering process are the following:

- System requirements definition
- System architecture design
- Sub-system development
- System integration
- System evolution

2.1.1 System requirements definition

System requirement definitions specify what the system should do, its functionality and its essential and desirable system properties. The techniques applied to elicit and collect information in order to create system specifications and requirement definitions involve consultations, interviews, requirements workshop with customers and end users. The objective of the requirements definition phase is to derive the two types of requirement:

1. *Functional requirements.* They define the basic functions that the system must provide and focus on the needs and goals of the end users.
2. *System requirements (non-functional requirements).* These are non-functional system properties such as availability, performance and safety etc. They define functions of a system, services and operational constraints in detail.

2.1.2 System architecture design

System architecture design is concerned with how the system functionality is to be provided by the components of the system. The activities involved in this process are:

1. *Partition of requirements.* After analysing the system requirements they are organized into related groups using several partitioning options.
2. *Identification of sub-systems.* The objective of this activity is to identify the sub-systems that can individually or collectively meet the system requirements. The relationships between sub-systems should be also identified at this time.
3. *Assignment of requirements to sub-systems.* In this activity the requirements are assigned to the identified sub-systems. If the sub-system identification is based on the results of requirements partitioning it provides an unambiguous assignment.
4. *Specification of sub-system functionality.* It is the specification of functionality provided by each sub-system. There shall be no overlapping or similar functions, the architecture shall be followed.
5. *Definition of sub-system interfaces.* Interfaces provide the communication between the sub-systems. Once the interfaces have been defined it becomes possible to develop sub-systems in parallel.

There may be a lot of feedback and iteration from one stage to another in this design process. As problems, questions, new requirements arise the revision of earlier stages may be required. The processes of requirements engineering and design in practice closely linked. Constraints determined by existing systems may limit design choices, and these choices may be specified in the requirements. During the design process problems with existing requirements may arise and new requirements may emerge. These have effect on design decisions again and vice versa. Therefore the linked processes can be represented by a spiral, as shown in Figure 2.1.

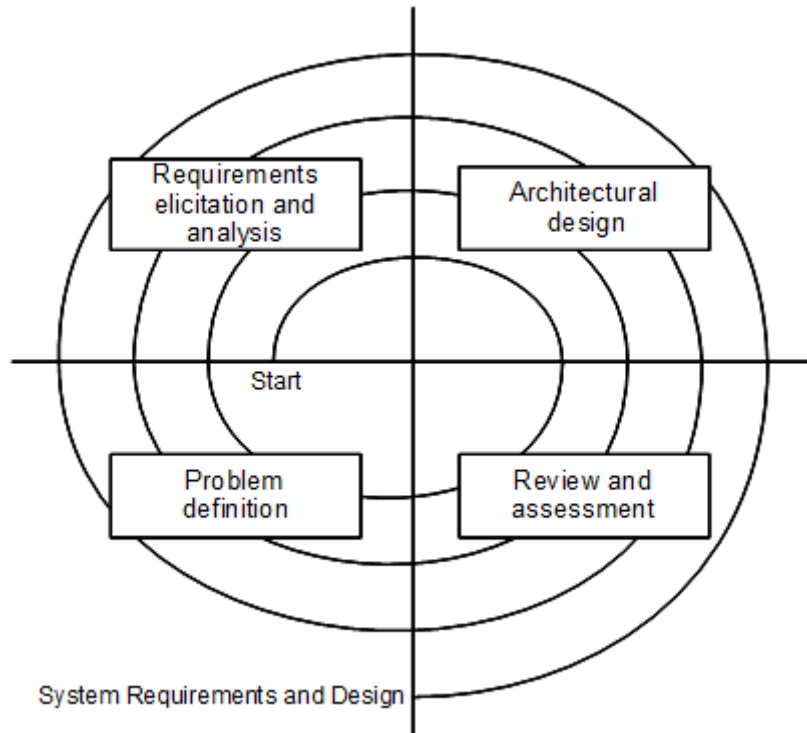


Figure 2.1. Spiral model of requirements and design.

Starting in the centre, each round of the spiral may add more detail to the requirements and the design. Some rounds may focus on requirements, some on design.

During the phase of requirements and design, systems may be modelled as a set of components and relationships between these components. These are usually illustrated graphically in a system architecture model that gives an overview of the system architecture. The system architecture may be simpler presented as a block diagram showing the major sub-systems and the interconnections between these sub-systems. The relationships indicated may include data flow or some other type of dependency relationship. For an example, Figure 2.2. shows the decomposition of an alarm system into its principal components.

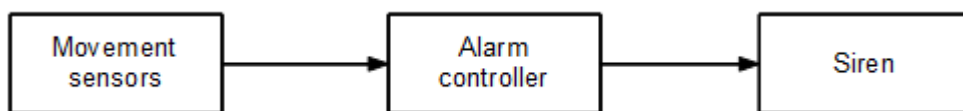


Figure 2.2. Block diagram of a simple alarm system.

2.1.3 Sub-system development

During sub-system development, the sub-systems identified are implemented. This may involve starting another system engineering process for individual sub-systems or, if the sub-system is software, a software process involving requirements, design, implement and testing.

Usually all sub-systems of the system are designed and developed during the development process. However, some of the sub-systems may be bought as commercial system for the reason of integration into the system. Sometime it is cheaper to buy existing products than to develop specific components. However, commercial systems may not meet all the requirements exactly. In these cases, it is necessary to change the requirements and repeat the design phase of development to correctly accommodate the purchased component. Another possibility is to request or make changes on the purchased component.

2.1.4 Systems integration

During the system integration process, the independently developed subsystems are put together to make up a complete system. Integration can be done using such an approach, where all the sub-systems are integrated at the same time. However, for technical and managerial purposes, an incremental integration process where sub-systems are integrated one at a time looks a better approach, for two reasons:

1. It is usually impossible to schedule the development of all the sub-systems so that they are all finished at the same time.
2. Incremental integration reduces the cost of error location. If many sub-systems are simultaneously integrated, an error that arises during testing may be in any of these sub-systems. When a single sub-system is integrated with an already working system, errors that occur are probably in the newly integrated sub-system or in the interactions between the existing sub-systems and the new sub-system.
3. The resources can be shared, their usage can be scheduled on a better way. The costs may be optimized. For example the team which is responsible to test the sub-systems, can consist less engineers and the can test the sub-systems one after another, or the usage of an expensive machine can be shared, etc.

Once the components have been integrated, an extensive system testing takes place. This testing should be aimed at testing the interfaces between components and the behaviour of the system as a whole.

2.1.5 System evolution

Large and complex systems may have a very long lifetime. During their life, they are changed to correct errors in the original system requirements and to implement new requirements that have emerged. Other reasons for changes: the computers in the system are likely to be replaced with new high-performance machines, the organization that uses the system may reorganize itself and hence use the system in a different way, the external environment of the system may change, forcing changes to the system, etc.

System evolution is inherently costly for several reasons:

1. Proposed changes have to be analyzed very carefully from a business and a technical viewpoint. Changes have to contribute to the goals of the system and should not simply be technically motivated.
2. Because sub-systems are never completely independent, changes to one subsystem may affect the performance or behaviour of other sub-systems. Consequent changes to these sub-systems may therefore be needed.
3. The reasons for original design decisions are often unrecorded. Those responsible for the system evolution have to work out why particular design decisions were made.
4. As systems age, their structure typically becomes corrupted by change so the costs of making further changes increases.

2.2 Exercises

1. Give definition of system!
2. What are the socio-technical systems?
3. What are the functional requirements?
4. What is the different between functional and non-functional requirements?
5. What are the phases of system engineering process?
6. List the iterative processes of spiral model!
7. What are the types of system integration?

8. Explain the process of system integration!
9. What is the similarity between incremental system integration and spiral model of design?
10. What is the meaning of system evolution?

3 Software processes

A software development process, also known as a software development lifecycle, is a structure imposed on the development of a software product. A software process is represented as a set of work phases that is applied to design and build a software product. There is no ideal software process, and many organisations have developed their own approach to software development. Software development processes should make a maximum use of the capabilities of the people in an organisation and the specific characteristics of the systems that are being developed [1,14,15].

There are some fundamental activities that are common to all software processes:

1. *Software specification.* In this activity the functionality of the software and constraints on its operation must be defined.
2. *Software design and implementation.* The software that meets the specification is produced.
3. *Software validation.* The software must be validated to ensure that it has all the functionalities what the customer needs.
4. *Software evolution.* The software must evolve to meet changing customer needs.

3.1 Software process models

A software process model is an abstract representation of a software process. In this section a number of general process models are introduced and they are presented from an architectural viewpoint. These models can be used to explain different approaches to software development. They can be considered as process frameworks that may be extended and adapted to create more specific software engineering processes. In this chapter the following process models will be introduced:

1. *The waterfall model.* In this model of software process the fundamental process activities of specification, development, validation and evolution are represented as sequential process phases such as requirements specification, software design, implementation, testing and so on.
2. *Evolutionary development.* This approach interleaves the activities of specification, development and validation. An initial system is rapidly developed from abstract specifications. Then the initial system is refined by customer inputs to produce a system that satisfies the customer's needs.
3. *Component-based software engineering.* The process models that use this approach are based on the existence of a significant number of reusable components. The system development process focuses on integrating these components into a system rather than developing them.

These three generic process models are widely used in current software engineering practice. They are not mutually exclusive and are often used together, especially for large systems development. Sub-systems within a larger system may be developed using different approaches. Therefore, although it is convenient to discuss these models separately, in practice, they are often combined.

3.1.1 The waterfall model

The waterfall model was the first software process model to be introduced (Figure 3.1.). It is also referred to as a linear-sequential life cycle model. The principal stages of the model represent the fundamental development activities:

1. *Requirements analysis and definition.* Software requirements specification establishes the basis for agreement between customers and contractors or suppliers on what the

software product is to do. Software requirements specification permits a rigorous assessment of requirements before design can begin. It should also provide basis for estimating product costs, risks, and schedules.

2. *System and software design.* Design activity results in the overall software architecture. Software design involves identifying and describing the fundamental software system components and their relationships. The systems design process partitions the requirements to either hardware or software components.
3. *Implementation and unit testing.* During this phase, the software design is realised as a set of software components. Components are tested ensuring each component meets its specification.
4. *Integration and system testing.* The program units or components are integrated and tested as a complete system to ensure that the software requirements have been met. After successful testing, the software system is delivered to the customer.
5. *Operation and maintenance.* The system is delivered and deployed and put into practical use. Maintenance involves correcting errors which were not discovered in earlier stages of the life cycle, improving the implementation of system units and providing new functionalities as new requirements emerge.

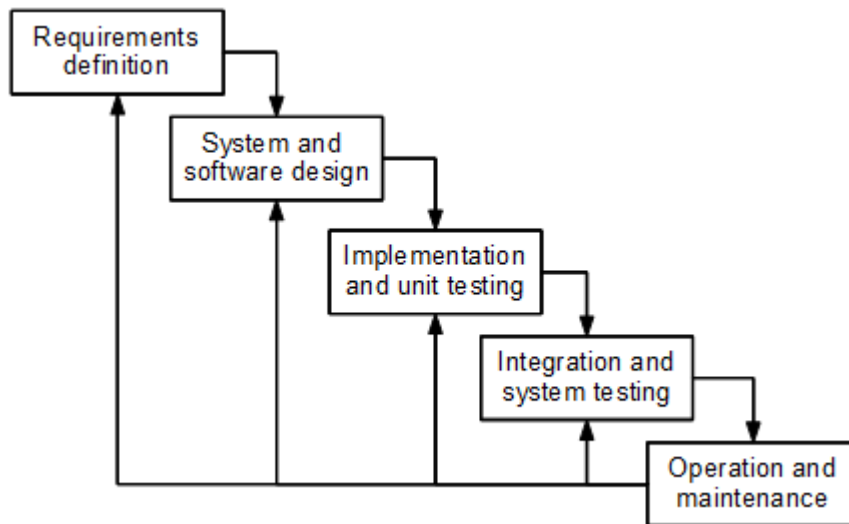


Figure 3.1. The software life cycle.

In principle, the result of each phase is one or more documents. The following phase shall not start until the previous phase has finished. In practice, these stages overlap and feed information to each other. During design, problems with requirements are identified; during coding design problems are found and so on. The software process is not a simple linear model but involves a sequence of iterations of the development activities. Because of the costs of producing and approving documents, iterations are costly and involve significant rework. Therefore, the waterfall model should only be used when the requirements are well understood and unlikely to change significantly during system development.

3.1.1.1 V-model of software process

The V-model represents a software process model that may be considered an extension of the waterfall model. Instead of moving down in a linear way, the process steps are bent upwards after the implementation phase, to form the typical V shape. The V-model demonstrates the relationships between each phase of the development life cycle and its associated phase of testing. The horizontal and vertical axes represent time and level of abstraction respectively. Similarly to waterfall model the process steps follow each other in a sequential order but V-model allows the parallel execution of activities.

In the requirements definition phase the requirements of the system are collected by analyzing the needs of the user, and in parallel the user acceptance or functional test cases are also designed. At the level of architectural design the software architecture, its components with their interface are designed at high-level to provide functional requirements of software. The design of integration testing is also carried out in this phase. In the component design phase the units and modules are designed at low-level. The low level design document or program specifications will contain a detailed functional logic of the units and modules. The unit test cases are also developed in this phase. After the implementation phase the development process continues upward by unit, integration and system testing.

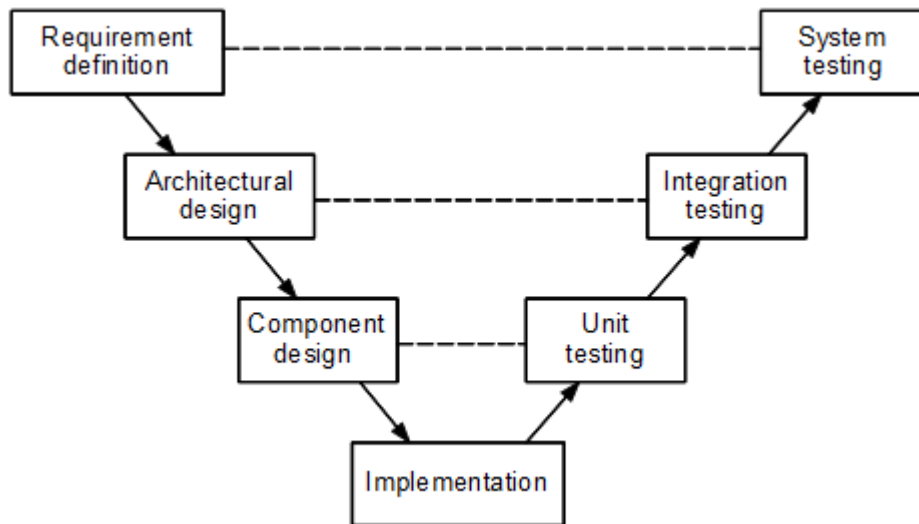


Figure 3.2. Representation of V-model.

3.1.2 Evolutionary development

Evolutionary development is based on the idea of developing an initial implementation, exposing this to user comment and refining it through many versions until an adequate system has been developed (Figure 3.3). Specification, development and validation activities are interleaved with rapid feedback across activities.

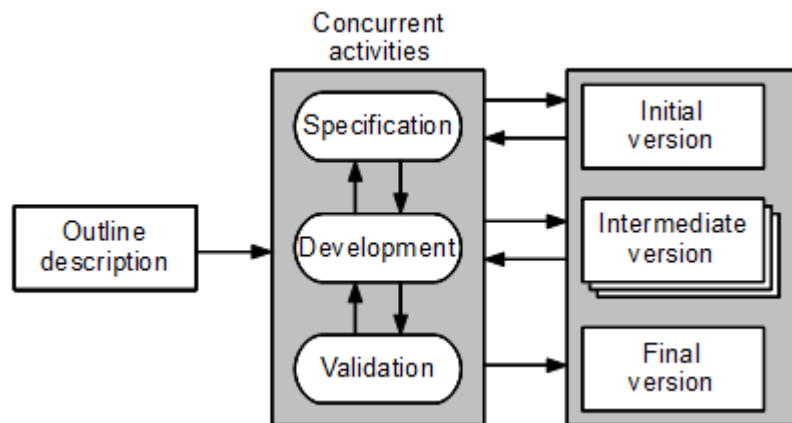


Figure 3.3. Evolutionary development.

There are two fundamental types of evolutionary development:

1. *Exploratory development.* The objective of the process is to work with the customer in order to explore their requirements and deliver a final system. The development starts with the parts of the system that are well understood. The system evolves by adding new features proposed by the customer.

2. *Throwaway prototyping*. In this case the objective of the evolutionary development process is to understand the customer's unclear requirements, namely to validate and derive the requirements definition for the system. The prototype concentrates on experimenting with the customer requirements that are poorly understood.

An evolutionary approach to software development is often more effective than the waterfall approach in producing systems that meet the immediate needs of customers. The advantage of a software process that is based on an evolutionary approach is that the specification can be developed incrementally. As users develop a better understanding of their problem, this can be reflected in the software system. However the evolutionary approach has also problems. Regular deliverables are need for monitoring progress. If systems are developed quickly, it is not cost-effective to produce documents that reflect every version of the system. Continual change tends to corrupt the software structure.

3.1.3 Component-based software engineering

In the majority of software projects, there is some software to reuse. The reusable components are systems that may provide specific functionality for the system. This reuse-oriented approach relies on a large base of reusable software components and some integrating framework for these components. The stages of component-based software process which are different to other processes are the followings:

1. *Component analysis*. Based on the requirements specification, a search is made for components that can implement the given specification. Usually, there is no exact match, and the components that may be used only provide some of the functionality required.
2. *Requirements modification*. During this stage, the requirements are analysed using information about the new components. Requirements are then modified to reflect the services of available components.
3. *System design with reuse*. During this phase, the framework of the system is designed or an existing framework is reused. The designers take into account the components that are reused. Some new software may have to be designed if reusable components are not available.
4. *Development and integration*. Software that cannot be externally procured is developed, and the components and reusable systems are integrated to create the new system.

Component-based software engineering has the obvious advantage of reducing the amount of software to be developed and so reducing cost and risks. It usually also leads to faster delivery of the software. However, requirements compromises are inevitable and this may lead to a system that does not meet the real (original) needs of users.

3.2 Process iteration

Changes are usually unavoidable in all large software projects. The system requirements change as organization continuously responds to the changing environment and conditions. Management priorities may change. Due to the quick progress in technologies, designs and implementation will change. This means that the process activities are regularly repeated as the system is reworked in response to change requirements. The following two process models have been designed to support process iteration:

1. *Incremental delivery*. The software specification, design and implementation are broken down into a series of increments that are each developed in turn.
2. *Spiral development*. The development of the system spirals outwards from an initial outline through to the final developed system.

3.2.1 Incremental delivery

The waterfall model of development requires defining the requirements for a system before design begins. On contrary, an evolutionary development allows requirements to change but it leads to software that may be poorly structured and difficult to understand and maintain.

Incremental delivery (Figure 3.4) is an approach that combines the advantages of these two models. In an incremental development process, customers identify the services to be provided by the software system. They decide which subset of the services is most important and which are least important to them. A number of delivery increments are then defined, with each increment providing a sub-set of the system functionality. The allocation of services to increments depends on the priority of service. The highest priority services are delivered first.

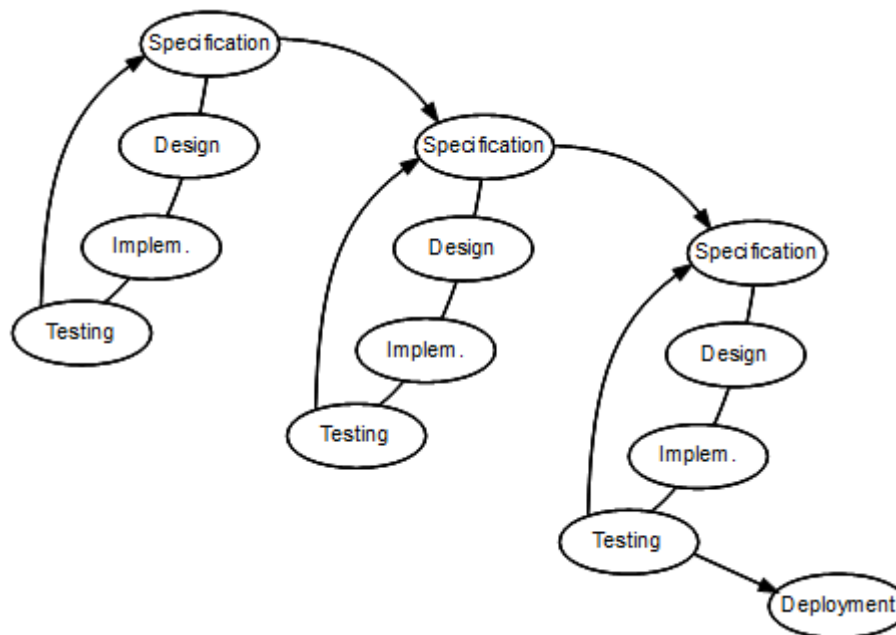


Figure 3.4. Incremental development.

Once the system increments have been identified, the requirements for first increment are defined in detail, and that increment is developed using the best suited software process. In the example given in Figure 3.4. the waterfall model is used to develop the increments in every iteration. As new increments are completed, they are integrated with existing increments and the system functionality improves with each delivered increment. The incremental development process has a number of advantages:

1. Customers do not have to wait until the entire system is delivered before they can gain value from it. The first increment satisfies their most critical requirements so they can use the software immediately.
2. Customers can use the early increments as prototypes and gain experience that informs their requirements for later system increments.
3. There is a lower risk of overall project failure. Although problems may be encountered in some increments, it is likely that these will be solved in later versions.
4. As the highest priority services are delivered first, and later increments are integrated with them, it is unavoidable that the most important system services receive the most testing. This means that software failures are less likely to occur in the most important parts of the system.

3.2.2 Spiral development

The spiral model of the software process (Figure 3.5.) represents the software process as a sequence of activities with some backtracking from one activity to another, the process is represented as a spiral. Each loop in the spiral represents a phase of the software process. Thus, the innermost loop might be concerned with system feasibility, the next loop with requirements definition, the next loop with system design and so on.

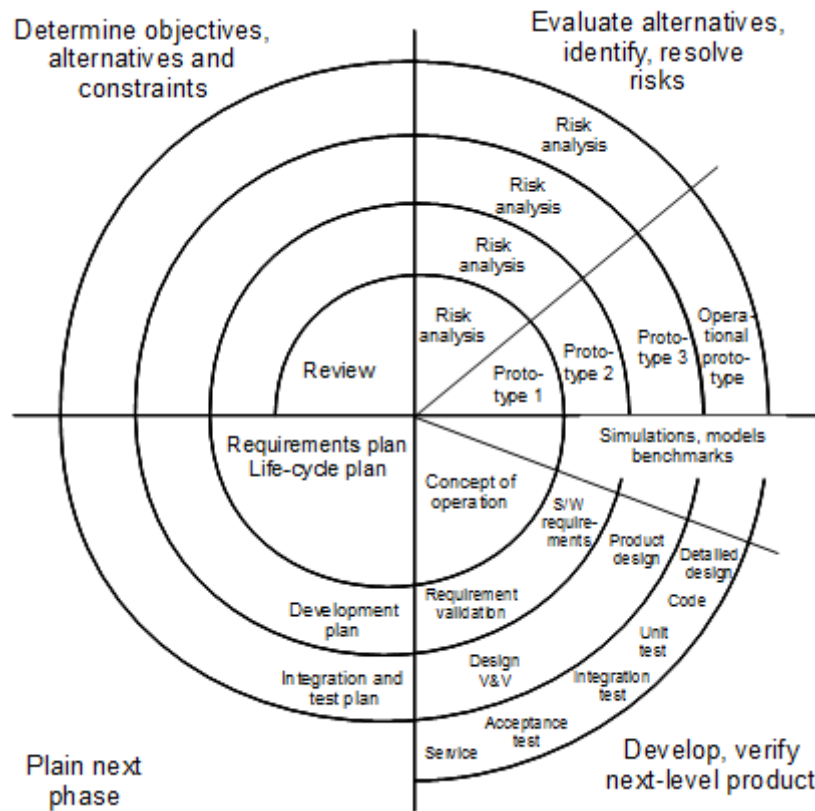


Figure 3.5. Spiral model of software process.

Each loop in the spiral is split into four sectors:

1. *Objective setting.* Specific objectives for that phase of the project are defined. Constraints on the process and the product are identified and a detailed management plan is drawn up. Project risks are identified. Alternative strategies, depending on these risks, may be planned.
2. *Risk assessment and reduction.* For each of the identified project risks, a detailed analysis is carried out. Steps are taken to reduce the risk.
3. *Development and validation.* After risk evaluation, a development model for the system is chosen.
4. *Planning.* The project is reviewed and a decision made whether to continue with a further loop of the spiral. If it is decided to continue, plans are drawn up for the next phase of the project.

The main difference between the spiral model and other software process models is the explicit recognition and handling of risk in the spiral model.

3.3 Process activities

The four basic process activities of specification, development, validation and evolution are organised differently in different development processes. In the waterfall model, they are

organised in sequence, whereas in evolutionary development they are interleaved. How these activities are carried out depends on the type of software, people and organisational structures involved.

3.3.1 Software specification

A software requirement is defined as a condition to which a system must comply. Software specification or requirements management is the process of understanding and defining what functional and non-functional requirements are required for the system and identifying the constraints on the system's operation and development. The requirements engineering process results in the production of a software requirements document that is the specification for the system.

There are four main phases in the requirements engineering process:

1. *Feasibility study*. In this study an estimate is made of whether the identified user needs may be satisfied using current software and hardware technologies. The study considers whether the proposed system will be cost-effective from a business point of view and whether it can be developed within existing budgetary constraints.
2. *Requirements elicitation and analysis*. This is the process of deriving the system requirements through observation of existing systems, discussions with potential users, requirements workshop, storyboarding, etc.
3. *Requirements specification*. This is the activity of translating the information gathered during the analysis activity into a document that defines a set of requirements. Two types of requirements may be included in this document: user (functional) requirements and system (non-functional) requirements.
4. *Requirements validation*. It is determined whether the requirements defined are complete. This activity also checks the requirements for consistency.

3.3.2 Software design and implementation

The implementation phase of software development is the process of converting a system specification into an executable system through the design of system. A software design is a description of the architecture of the software to be implemented, the data which is part of the system, the interfaces between system components and, sometimes, the algorithms used. The design process activities are the followings:

1. *Architectural design*. The sub-systems of system and their relationships are identified based on the main functional requirements of software.
2. *Abstract specification*. For each sub-system, an abstract specification of its services and the constraints under which it must operate is defined.
3. *Interface design*. Interfaces allow the sub-system's services to be used by other sub-systems. The representation of interface should be hidden. In this activity the interface is designed and documented for each sub-system. The specification of interface must be unambiguous.
4. *Component design*. Services are allocated to components and the interfaces of these components are designed.
5. *Data structure design*. The data structures used in the system implementation are designed in detail and specified.
6. *Algorithm design*. In this activity the algorithms used to provide services are designed in detail and specified.

This general model of the design process may be adapted in different ways in the practical uses.

A contrasting approach can be used by *structured methods* for design objectives. A structured method includes a design process model, notations to represent the design, report formats, rules and design guidelines. Most these methods represent the system by graphical models and many cases can automatically generate program code from these models. Various competing methods to support object-oriented design were proposed in the 1990s and these were unified to create the Unified Modeling Language (UML) and the associated unified design process.

3.3.3 Software validation

Software validation or, more generally, verification and validation (V & V) is intended to show that a system conforms to its specification and that the system meets the expectations of the customer buying the system. It involves checking the processes at each stage of the software process. The majority of validation costs are incurred after implementation when the operation of system is tested.

The software is tested in the usual three-stage testing process. The system components, the integrated system and finally the entire system are tested. Component defects are generally discovered early in the process and the interface problems during the system integration. The stages in the testing process are:

1. *Component (or unit) testing.* Individual components are tested to ensure that they operate correctly. Each component is tested independently, without other system components.
2. *System testing.* The components are integrated to make up the system. This testing process is concerned with finding errors that result from interactions between components and component interface problems. It is also concerned with validating that the system meets its functional and non-functional requirements.
3. *Acceptance testing.* It is considered a functional testing of system. The system is tested with data supplied by the system customer.

Usually, component development and testing are interleaved. Programmers make up their own test data and test the code as it is developed. However in many process model, such as in V-model, Test Driven Development, Extreme Programming, etc., the design of the test cases starts before the implementation phase of development. If an incremental approach to development is used, each increment should be tested as it is developed, with these tests based on the requirements for that increment.

3.3.4 Software evolution

Software evolution, specifically software maintenance, is the term used in software engineering to refer to the process of developing software initially, then repeatedly updating it for various reasons.

The aim of software evolution would be to implement the possible major changes to the system. The existing larger system is never complete and continues to evolve. As it evolves, the complexity of the system will grow. The main objectives of software evolution are ensuring the reliability and flexibility of the system. The costs of maintenance are often several times the initial development costs of software.

3.4 The Rational Unified Process

The Rational Unified Process (RUP) methodology is an example of a modern software process model that has been derived from the UML and the associated Unified Software Development Process. The RUP recognises that conventional process models present a single view of the process. In contrast, the RUP is described from three perspectives [3,11,17,22]:

1. A dynamic perspective that shows the phases of the model over time.
2. A static perspective that shows the process activities.
3. A practice perspective that suggests good practices to be used during the process.

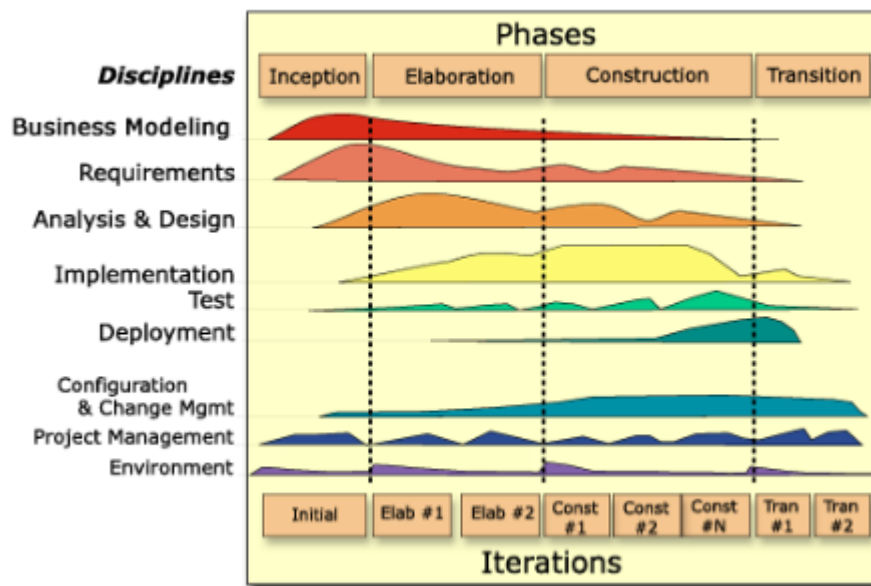


Figure 3.6. Phases of Rational Unified Process.

The RUP (Figure 3.6.) identifies four discrete development phases in the software process that are not equated with process activities. The phases in the RUP are more closely related to business rather than technical concerns. These phases in the RUP are:

1. *Inception*. The goal of the inception phase is to establish a business case for the system, identify all external entities, i.e. people and systems that will interact with the system and define these interactions. Then this information is used to assess the contribution that the system makes to the business.
2. *Elaboration*. The goals of the elaboration phase are to develop an understanding of the problem domain, establish an architectural framework for the system, develop the project plan and identify key project risks.
3. *Construction*. The construction phase is essentially concerned with system design, programming and testing.
4. *Transition*. The final phase of the RUP is concerned with moving the system from the development community to the user community and making it work in a real environment.

Iteration within the RUP is supported in two ways. Each phase may be enacted in an iterative way with the results developed incrementally. In addition, the whole set of phases may also be enacted incrementally.

The static view of the RUP focuses on the activities that take place during the development process. These are called *workflows* in the RUP description. There are six core process workflows identified in the process and three core supporting workflows. The RUP has been designed in conjunction with the UML so the workflow description is oriented around associated UML models. The core engineering and support workflows are the followings:

1. *Business modelling*. The business processes are modelled using business use cases.
2. *Requirements*. Actors who interact with the system are identified and use cases are developed to model the system requirements.

3. *Analysis and design.* A design model is created and documented using architectural models, component models, object models and sequence models.
4. *Implementation.* The components in the system are implemented and structured into implementation sub-systems. Automatic code generation from design models helps accelerate this process.
5. *Testing.* Testing is an iterative process that is carried out in conjunction with implementation. System testing follows the completion of the implementation.
6. *Deployment.* A product release is created, distributed to users and installed in their workplace.
7. *Configuration and change management.* This supporting workflow manages changes to the system.
8. *Project management.* This supporting workflow manages the system development.
9. *Environment.* This workflow is concerned with making appropriate software tools available to the software development team.

The advantage in presenting dynamic and static views is that phases of the development process are not associated with specific workflows. In principle at least, all of the RUP workflows may be active at all stages of the process. Of course, most effort will probably be spent on workflows such as business modelling and requirements at the early phases of the process and in testing and deployment in the later phases.

The practice perspective on the RUP describes good software engineering practices that are recommended for use in systems development. Six fundamental best practices are recommended:

1. *Develop software iteratively.* Plan increments of the system based on customer priorities and develop and deliver the highest priority system features early in the development process.
2. *Manage requirements.* Explicitly document the customer's requirements and keep track of changes to these requirements. Analyze the impact of changes on the system before accepting them.
3. *Use component-based architectures.* Structure the system architecture into components.
4. *Visually model software.* Use graphical UML models to present static and dynamic views of the software.
5. *Verify software quality.* Ensure that the software meets the organisational quality standards.
6. *Control changes to software.* Manage changes to the software using a change management system and configuration management procedures and tools.

The RUP is not a suitable process for all types of development but it does represent a new generation of generic processes. The most important innovations are the separation of phases and workflows, and the recognition that deploying software in a user's environment is part of the process. Phases are dynamic and have goals. Workflows are static and are technical activities that are not associated with a single phase but may be used throughout the development to achieve the goals of each phase.

3.5 Exercises

1. What software development project does waterfall model recommended for?
2. What software development project does waterfall model not recommended for?
3. List application problems related to waterfall development!
4. What is the meaning of requirements validation?
5. What are the main activities of software design phase?

6. What is the different between the system and acceptance testing?
7. Explain the role of prototypes in the evolutionary development!
8. Explain the reason for software developed by an evolutionary development are often more difficult to maintain!
9. What are the support workflows in RUP?
10. How does RUP support the iterative software process?

4 Agile methods

Nowadays, most organizations and companies have to work under rapidly changing conditions. Software is used in almost all business so it is essential that new software is developed quickly. Rapid software development is therefore one of the most critical requirements for software systems [1,23].

The conventional software development processes that are based on completely specifying the requirements then designing, building and testing the system are not applicable to rapid software development. As the requirements change the system design or implementation has to be reworked and retested. As a consequence, the development process is usually late and the final software is delivered to the customer long after it was originally specified. In some cases the original reason for developing software may have changed so radically that the software becomes effectively useless when it is delivered. Therefore, development processes, especially in the case of business systems, have to focus on rapid software development and delivery.

In the 1980s and early 1990s, it was considered that the best way to achieve better software quality was provided by rigorously planned and controlled conventional software development processes that were normally used for development of large-scale systems. These development processes usually involve a significant overhead in planning, designing and documenting the system. However, when this plan-based development approach was applied to small and medium-sized business systems, the overhead involved was so large that it sometimes dominated the software development process. Software engineers often spent more time on how the system should be developed, than on program development and testing.

Dissatisfaction with these approaches led a number of software developers in the 1990s to propose new agile methods. These allowed the development team to focus on the software itself rather than on its design and documentation. Agile methods universally rely on an iterative approach to software specification, development and delivery, and were designed primarily to support business application development where the system requirements usually changed rapidly during the development process. They are intended to deliver working software quickly to customers, who can then propose new and changed requirements to be included in later iterations of the system.

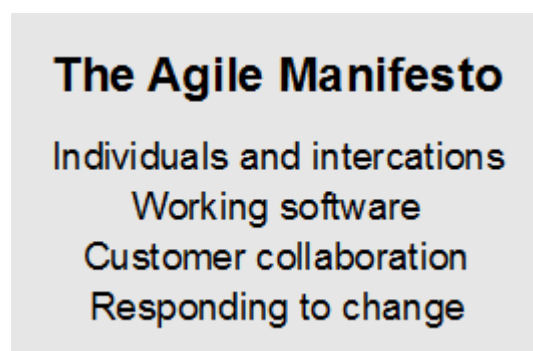


Figure 4.1. The Agile Manifesto.

In February 2001, 17 software developers met at the Snowbird, Utah resort, to discuss lightweight development methods. They published the Manifesto for Agile Software Development to define the approach now known as agile software development. The Agile Manifesto reads, in its entirety, as follows:

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

1. Individuals and interactions over Processes and tools
2. Working software over Comprehensive documentation
3. Customer collaboration over Contract negotiation
4. Responding to change over Following a plan

That is, while there is value in the items on the right, we value the items on the left more. The meanings of the manifesto items on the left within the agile software development context are:

- *Individuals and interactions.* In agile development, self-organization and motivation are important, as are interactions like co-location and pair programming.
- *Working software.* Working software will be more useful and welcome than just presenting documents to clients in meetings.
- *Customer collaboration.* Requirements cannot be fully collected at the beginning of the software development cycle, therefore continuous customer or stakeholder involvement is very important.
- *Responding to change.* Agile development is focused on quick responses to change and continuous development.

The Agile Manifesto is based on twelve principles:

1. Customer satisfaction by rapid delivery of useful software.
2. Welcome changing requirements, even late in development.
3. Working software is delivered frequently (weeks rather than months).
4. Working software is the principal measure of progress.
5. Sustainable development, able to maintain a constant pace.
6. Close, daily cooperation between business people and developers.
7. Face-to-face conversation is the best form of communication (co-location).
8. Projects are built around motivated individuals, who should be trusted.
9. Continuous attention to technical excellence and good design.
10. Simplicity, the art of maximizing the amount of work not done, is essential.
11. Self-organizing teams.
12. Regular adaptation to changing circumstances.

Agile software development processes are designed to produce useful software quickly. Generally, they are iterative processes where specification, design, development and testing are interleaved. The software is not developed and deployed in its entirety but in a series of increments, with each increment including new system functionality. Although there are many approaches to rapid software development, they share some fundamental characteristics:

1. The processes of specification, design and implementation are concurrent. There is no detailed system specification, and design documentation is minimised or generated automatically by the programming environment used to implement the system. The user requirements document defines only the most important characteristics of the system.
2. The system is developed in a series of increments. End-users and other system stakeholders are involved in specifying and evaluating each increment. They may propose changes to the software and new requirements that should be implemented in a later increment of the system.
3. System user interfaces are often developed using an interactive development system that allows the interface design to be quickly created by drawing and placing icons on the interface.

Incremental development involves producing and delivering the software in increments rather than in a single package (Figure 4.2.). Each process iteration produces a new software increment. The two main advantages to adopting an incremental approach to software development are:

1. *Accelerated delivery of customer services.* Early increments of the system can deliver high-priority functionality so that customers can get value from the system early in its development. Customers can see their requirements in practice and specify changes to be incorporated in later releases of the system.
2. *User engagement with the system.* Users of the system have to be involved in the incremental development process because they have to provide feedback to the development team on delivered increments.

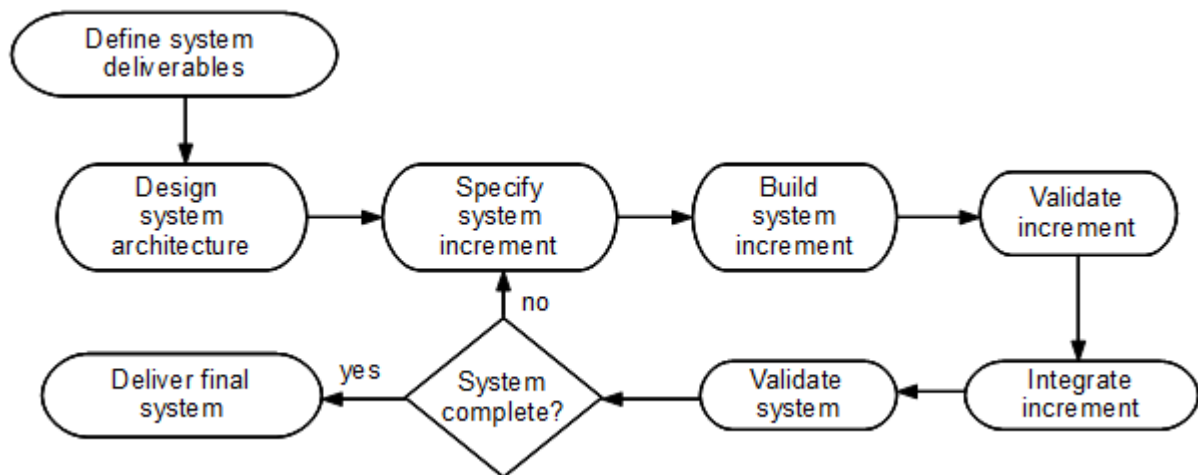


Figure 4.2. Incremental development process.

However, there can be real difficulties with incremental development, particularly in large companies with fairly rigid procedures and in organisations where software development is usually outsourced to an external contractor. The major difficulties with iterative development and incremental delivery are:

1. *Management problems.* Software processes that are generally applied for large system development regularly generate documents that can be used to assess the progress of development. However, in an iterative development process a lots of system documentation is produced that is not cost effective. Furthermore, managers may have difficulties relating to the applied technologies and skill of staff.
2. *Contractual problems.* The contract between a customer and a software developer is normally based on the system specification. In the case of iterative development the complete specification is only available at the end of development, so it may be difficult to design a contract for the system development.
3. *Validation problems.* An independent V & V team can start work as soon as the specification is available and can prepare tests in parallel with the system implementation. However, iterative development processes interleave specification and development. Hence, independent validation of incrementally developed systems is difficult.
4. *Maintenance problems.* Continual change may corrupt the structure of any software system and makes it difficult to understand. This problem can be reduced by continual refactoring program codes.

Of course, there are some types of systems where incremental development and delivery is not the best approach. These are very large systems where development may involve teams

working in different locations, some embedded systems where the software depends on hardware development and some critical systems where all the requirements must be analysed to check for interactions that may compromise the safety or security of the system.

Probably the best-known agile method is extreme programming. Other agile approaches include Scrum, Crystal, Adaptive Software Development, DSDM and Feature Driven Development. Although these agile methods are all based around the notion of incremental development and delivery, they propose different processes to achieve this. However, they share a set of principles and therefore have much in common. In this chapter the Extreme Programming, Scrum and Feature Driven Development are overviewed.

4.1 Extreme programming

Extreme Programming (XP) is a software development methodology which is intended to improve software quality and responsiveness to changing customer requirements. As a type of agile software development, it advocates frequent releases in short development cycles, which is intended to improve productivity and introduce checkpoints at which new customer requirements can be adopted [1,20].

Other elements of Extreme Programming include: programming in pairs or doing extensive code review, unit testing of all code, avoiding programming of features until they are actually needed, a flat management structure, simplicity and clarity in code, expecting changes in the customer's requirements as time passes and the problem is better understood, and frequent communication with the customer and among programmers. The methodology takes its name from the idea that the beneficial elements of traditional software engineering practices are taken to "extreme" levels.

Extreme Programming was created by Kent Beck in 1999 [16]. Although extreme programming itself is relatively new, many of its practices have been around for some time; the methodology, after all, takes best practices to extreme levels. For example, to shorten the total development time, some formal test documents have been developed in parallel the software is ready for testing. In XP, this concept is taken to the extreme level by writing automated tests which validate the operation of even small sections of software coding, rather than only testing the larger features.

XP attempts to reduce the cost of changes in requirements by having multiple short development cycles, rather than a long one. On this basis, changes are a natural, unavoidable and desirable aspect of software-development projects, and should be planned for, instead of attempting to define a non-variable set of requirements.

Extreme programming also introduces a number of basic values, principles and practices on top of the agile programming framework.

Values

Extreme Programming initially recognizes five values:

- Communication
- Simplicity
- Feedback
- Courage
- Respect

These five values are described below.

Communication

Building software systems requires communicating system requirements to the developers of the system. In formal software development methodologies, this task is accomplished through documentation. Extreme programming techniques can be considered as methods for rapid developments and sharing best practices and knowledge among members of a development team. The goal is to give all developers a shared view of the system which matches the user's view of the system. For this reason extreme programming favors simple designs, collaboration of users and programmers, frequent verbal communication, and feedback.

Simplicity

Extreme programming encourages starting with the simplest solution. Extra functionality can then be added later. The difference between this approach and more conventional system development methods is the focus on designing and coding for the needs of today instead of those of tomorrow, next week, or next month. This is sometimes summed up as the "You aren't gonna need it" (YAGNI) approach. Proponents of XP acknowledge the disadvantage that this can sometimes entail more effort tomorrow to change the system; their claim is that this is more than compensated for by the advantage of not investing in possible future requirements that might change before they become relevant. Coding and designing for uncertain future requirements implies the risk of spending resources on something that might not be needed, while perhaps delaying crucial features. Related to the "communication" value, simplicity in design and coding should improve the quality of communication. A simple design with very simple code could be easily understood by most programmers in the team.

Feedback

Within extreme programming, feedback relates to different dimensions of the system development:

- *Feedback from the system:* by writing unit tests, or running periodic integration tests, the programmers have direct feedback from the state of the system after implementing changes.
- *Feedback from the customer:* The functional tests are written by the customer and the testers. They will get concrete feedback about the current state of their system. This review is planned once in every two or three weeks so the customer can easily steer the development.
- *Feedback from the team:* When customers come up with new requirements in the planning game the team directly gives an estimation of the time that it will take to implement.

Feedback is closely related to communication and simplicity. Flaws in the system are easily communicated by writing a unit test that proves a certain piece of code will break. The direct feedback from the system tells programmers to recode this part. A customer is able to test the system periodically according to the functional requirements, known as *user stories*.

Courage

Several practices embody courage. One is the commandment to always design and code for today and not for tomorrow. This is an effort to avoid getting intense in design and requiring a lot of effort to implement anything else. Courage enables developers to feel comfortable with refactoring their code when necessary. This means reviewing the existing system and modifying it so that future changes can be implemented more easily. Another example of courage is knowing when to throw code away: courage to remove source code that is out of date, no matter how much effort was used to create that source code. Also, courage means

persistence: A programmer might be stuck on a complex problem for an entire day, then solve the problem quickly the next day, but only if they are persistent.

Respect

The respect value includes respect for others as well as self-respect. Programmers should never commit changes that break compilation, that make existing unit-tests fail, or that otherwise delay the work of their peers. Members respect their own work by always striving for high quality and seeking for the best design for the solution at hand through refactoring.

Adopting the four earlier values leads to respect gained from others in the team. Nobody on the team should feel unappreciated or ignored. This ensures a high level of motivation and encourages loyalty toward the team and toward the goal of the project. This value is very dependent upon the other values, and is very much oriented toward people in a team.

Requirements are recorded on Story Cards and the Stories to be included in a release are determined by the time available and their relative priority. The developers break these Stories into development “Tasks”. Developers work in pairs, checking each other’s work and providing the support to always do a good job. An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented. The Figure 4.3. shows the release cycle of extreme programming.

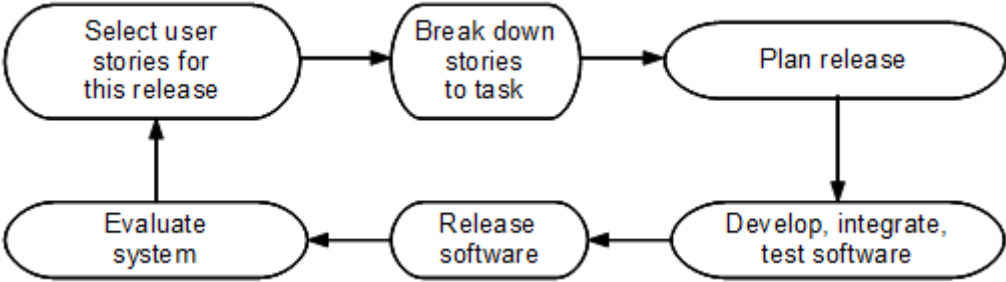


Figure 4.3. The extreme programming release cycle.

Extreme programming involves a number of practices, summarised in Table 4.1.

Table 4.1. Extreme programming practices.

Principle or practice	Description
Incremental planning	Requirements are recorded on Story Cards and the Stories to be included in a release are determined by the time available and their relative priority. The developers break these Stories into development “Tasks”.
Small releases	The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the first release.
Simple design	Enough design is carried out to meet the current requirements and no more.
Test-first development	An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.
Refactoring	All developers are expected to refactor the code continuously as soon as possible code improvements are found. This keeps the

	code simple and maintainable.
Pair programming	Developers work in pairs, checking each other's work and providing the support to always do a good job.
Collective ownership	The pairs of developers work on all areas of the system and all the developers own all the code. Anyone can change anything.
Continuous integration	As soon as work on a task is complete it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.
Sustainable pace	Large amounts of overtime are not considered acceptable as the net effect is often to reduce code quality and medium-term productivity
On-site customer	A representative of the end-user of the system should be available full time for the use of the XP team.

In an XP process, customers are intimately involved in specifying and prioritising system requirements. The customer is part of the development team and discusses scenarios with other team members. Together, they develop a 'story card' that encapsulates the customer needs. The development team then aims to implement that scenario in a future release of the software.

Once the story cards have been developed, the development team breaks these down into tasks and estimates the effort and resources required for implementation. The customer then prioritises the stories for implementation, choosing those stories that can be used immediately to deliver useful business support.

Extreme programming takes an extreme approach to iterative development. New versions of the software may be built several times per day and increments are delivered to customers roughly every two weeks. When a programmer builds the system to create a new version, he or she must run all existing automated tests as well as the tests for the new functionality. The new build of the software is accepted only if all tests execute successfully.

4.1.1 Testing in XP

In the iterative development processes there is no system specification that can be used by an external testing team to develop system tests. As a consequence, some approaches to iterative development have only a very informal testing process. For this reason XP places more emphasis than other agile methods on the testing process. The key features of testing in XP are:

1. Test-first development.
2. Incremental test development from scenarios.
3. User involvement in the test development and validation.
4. The use of automated test harnesses.

Test-first development is one of the most important characteristic in XP. Writing tests first implicitly defines both an interface and a specification of behaviour for the functionality being developed. User requirements in XP are expressed as scenarios or stories and the customer prioritises these for development. The role of the customer in the testing process is to help develop acceptance tests for the stories that are to be implemented in the next release of the

system. Acceptance testing is the process where the system is tested using customer data to check that it meets the customer's needs.

In the test-first development the test is written before the code. More precisely, the test is written as an executable component before the task is implemented. Once the software has been implemented, the test can be executed immediately. The testing component simulates the submission of input to be tested and check that the result meets the output specification.

The automated test harness is a system that submits these automated tests for execution.

Using test-first development, there is always a set of tests that can be quickly and easily executed. This means that whenever any functionality is added to the system, the tests can be run and problems that the new code has introduced can be recognised immediately.

4.1.2 Pair programming

Another innovative practice that has been introduced is that programmers work in pairs to develop the software. They actually sit together at the same workstation to develop the software. Development does not always involve the same pair of people working together. Rather, the idea is that pairs are created dynamically so that all team members may work with other members in a programming pair during the development process.

The use of pair programming has a number of advantages:

- *Collective code ownership.* When everyone on a project is Pair Programming, and pairs rotate frequently, everybody gains a working knowledge of the entire codebase.
- *Better code.* Pair programming acts as an informal review process because each line of code is looked at by at least two people. Code inspections and reviews are very successful in discovering a high percentage of software errors.
- *Increased discipline.* Pairing partners are more likely to "do the right thing" and are less likely to take long breaks.
- *Resilient flow.* Pairing leads to a different kind of flow than programming alone, but it does lead to flow. Pairing flow is more resilient to interruptions: one programmer deals with the interruption while the other keeps working.
- *Improved morale.* Pair programming, done well, is much more enjoyable than programming alone, done well.
- *Mentoring.* Everyone, even junior programmers, has knowledge that others don't. Pair programming is a painless way of spreading that knowledge.
- *Team cohesion.* People get to know each other more quickly when pair programming.
- *Fewer interruptions.* People are more reluctant to interrupt a pair than they are to interrupt someone working alone.
- *One less workstation.*

4.2 Scrum

Scrum is an iterative and incremental agile software development framework for managing software projects and product or application development. In rugby football, a scrum refers to the manner of restarting the game after a minor infraction [19].

In 1986 by Hirotaka Takeuchi and Ikujiro Nonaka described a new approach to commercial product development that would increase speed and flexibility, based on case studies from manufacturing firms in the automotive, photocopier and printer industries. They called this *rugby approach*, as the whole process is performed by one cross-functional team across multiple overlapping phases, where the team tries to go the distance as an unit, passing the ball back and forth.

Scrum enables teams to self-organize by encouraging physical co-location of all team members and daily face to face communication among all team members and disciplines in the project. A key principle of Scrum is its recognition that during a project the customers can change their minds about what they want and need, and that unpredicted challenges cannot be easily addressed in a traditional predictive or planned manner. Scrum focuses on maximizing the team's ability to deliver quickly and respond to emerging requirements.

The Scrum framework consists of Scrum Teams and their associated roles, events, artifacts, and rules. Each component within the framework serves a specific purpose and is essential to Scrum's success and usage.

4.2.1 Scrum roles

The Scrum Team consists of a Product Owner, the Development Team, and a Scrum Master. Scrum Teams are self-organizing and cross-functional. Self-organizing teams choose how best to accomplish their work, rather than being directed by others outside the team. Cross-functional teams have all competencies needed to accomplish the work without depending on others not part of the team.

Scrum Teams deliver products iteratively and incrementally, maximizing opportunities for feedback. Incremental deliveries of "Done" product ensure a potentially useful version of working product is always available.

Product Owner

The Product Owner is responsible for maximizing the value of the product and the work of the Development Team and for managing the Product Backlog. Product Backlog management includes: ordering the items in the Product Backlog to best achieve goals and missions; ensuring that the Product Backlog is visible, transparent, and clear to all; ensuring the Development Team understands items in the Product Backlog to the level needed.

The Product Owner may represent the desires of a committee in the Product Backlog, but those wanting to change a backlog item's priority must convince the Product Owner. For the Product Owner to succeed, the entire organization must respect his or her decisions.

Development Team

The Development Team consists of 3-9 professionals who do the work of delivering a potentially releasable Increment of "Done" product at the end of each Sprint. Only members of the Development Team create the Increment. Development Teams have the following main characteristics:

- They are self-organizing. No one (not even the Scrum Master) tells the Development Team how to turn Product Backlog into Increments of potentially releasable functionality;
- Development Teams are cross-functional, with all of the skills as a team necessary to create a product Increment;
- Individual Development Team members may have specialized skills and areas of focus, but accountability belongs to the Development Team as a whole;

Scrum Master

The Scrum Master is responsible for ensuring Scrum is understood and enacted. Scrum Masters do this by ensuring that the Scrum Team adheres to Scrum theory, practices, and rules. The Scrum Master is a servant-leader for the Scrum Team.

The Scrum Master helps those outside the Scrum Team understand which of their interactions with the Scrum Team are helpful and which aren't. The Scrum Master helps everyone change these interactions to maximize the value created by the Scrum Team.

4.2.2 Scrum artifacts

Scrum's artifacts represent work or value in various ways that are useful in providing transparency and opportunities for inspection and adaptation. Artifacts defined by Scrum are specifically designed to maximize transparency of key information needed to ensure Scrum Teams are successful in delivering a "Done" Increment.

Product backlog

The Product Backlog is an ordered list of everything that might be needed in the product and is the single source of requirements for any changes to be made to the product. The Product Owner is responsible for the Product Backlog, including its content, availability, and ordering.

A Product Backlog is never complete. The earliest development of it only lays out the initially known and best-understood requirements. The Product Backlog evolves as the product and the environment in which it will be used evolves.

The Product Backlog lists all features, functions, requirements, enhancements, and fixes that constitute the changes to be made to the product in future releases. Product Backlog items have the attributes of a description, order, and estimate.

As a product is used and gains value, and the marketplace provides feedback, the Product Backlog becomes a larger and more exhaustive list. Requirements never stop changing, so a Product Backlog is a living artifact. Changes in business requirements, market conditions, or technology may cause changes in the Product Backlog.

Sprint backlog

The Sprint Backlog is the set of Product Backlog items selected for the Sprint plus a plan for delivering the product Increment and realizing the Sprint Goal. The Sprint Backlog is a forecast by the Development Team about what functionality will be in the next Increment and the work needed to deliver that functionality.

The Sprint Backlog defines the work the Development Team will perform to turn Product Backlog items into a "Done" Increment. The Sprint Backlog makes visible all of the work that the Development Team identifies as necessary to meet the Sprint Goal.

As new work is required, the Development Team adds it to the Sprint Backlog. As work is performed or completed, the estimated remaining work is updated. When elements of the plan are deemed unnecessary, they are removed. Only the Development Team can change its Sprint Backlog during a Sprint. The Sprint Backlog is a highly visible, real-time picture of the work that the Development Team plans to accomplish during the Sprint, and it belongs solely to the Development Team.

Increment

The Increment is the sum of all the Product Backlog items completed during a Sprint and all previous Sprints. At the end of a Sprint, the new Increment must be "Done," which means it must be in useable condition and meet the Scrum Team's Definition of "Done." It must be in useable condition regardless of whether the Product Owner decides to actually release it.

4.2.3 Scrum events

Prescribed events are used in Scrum to create regularity and to minimize the need for meetings not defined in Scrum. Scrum uses time-boxed events, such that every event has a maximum duration. This ensures an appropriate amount of time is spent planning without allowing waste in the planning process.

Other than the Sprint itself, which is a container for all other events, each event in Scrum is a formal opportunity to inspect and adapt something. These events are specifically designed to enable critical transparency and inspection. Failure to include any of these events results in reduced transparency and is a lost opportunity to inspect and adapt.

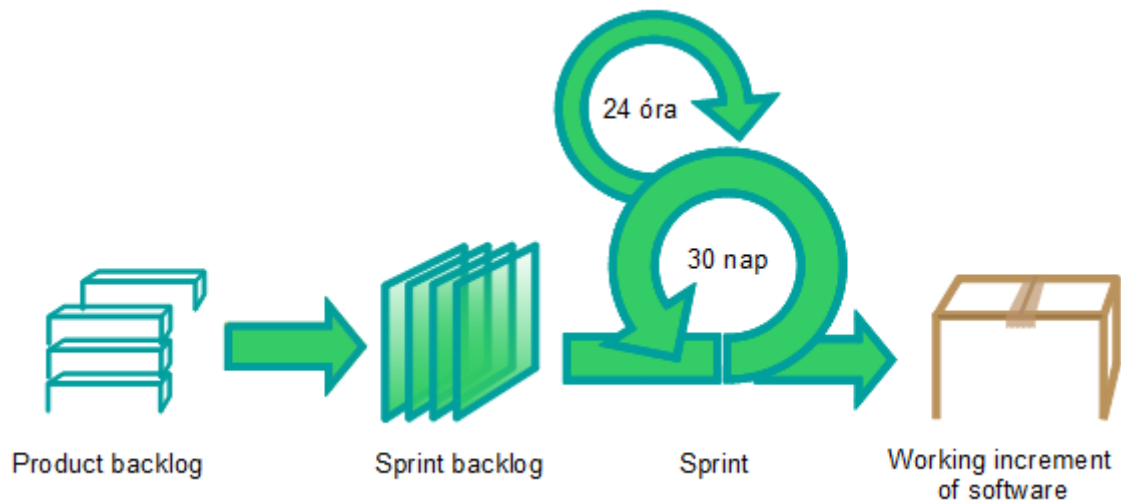


Figure 4.4. A scrum sprint.

The Sprint

The Sprint (Figure 4.4.) is a time-box of one month or less during which a “Done”, useable, and potentially releasable product Increment is created. The duration of Sprint is fixed in advance for each sprint and is normally between one week and one month, although two weeks is typical.

Each Sprint has a definition of what is to be built, a design and flexible plan that will guide building it, the work, and the resultant product.

Each sprint is started by a planning meeting, where the tasks for the sprint are identified and an estimated commitment for the sprint goal is made, and ended by a sprint review.

Scrum emphasizes working product at the end of the Sprint that is really "done"; in the case of software, this means a system that is integrated, fully tested, end-user documented, and potentially shippable.

Sprint planning meeting

The work to be performed in the Sprint is planned at the Sprint Planning Meeting. This plan is created by the collaborative work of the entire Scrum Team.

In this part, the Development Team works to forecast the functionality that will be developed during the Sprint. The Product Owner presents ordered Product Backlog items to the Development Team and the entire Scrum Team collaborates on understanding the work of the Sprint.

The number of items selected from the Product Backlog for the Sprint is solely up to the Development Team. By the end of the Sprint Planning Meeting, the Development Team should be able to explain to the Product Owner and Scrum Master how it intends to work as a self-organizing team to accomplish the Sprint Goal and create the anticipated Increment.

Daily Scrum meeting

The Daily Scrum is a 15-minute time-boxed event for the Development Team to synchronize activities and create a plan for the next 24 hours. The Daily Scrum is held at the same time and place each day to reduce complexity. During the meeting, each Development Team member explains:

- What has been accomplished since the last meeting?
- What will be done before the next meeting?

- What obstacles are in the way?

The Development Team uses the Daily Scrum to assess progress toward the Sprint Goal and to assess how progress is trending toward completing the work in the Sprint Backlog. Daily Scrums improve communications, eliminate other meetings, identify and remove impediments to development, highlight and promote quick decision-making, and improve the Development Team’s level of project knowledge.

Sprint review

This is an informal meeting, and the presentation of the Increment. A Sprint Review is held at the end of the Sprint to inspect the Increment and adapt the Product Backlog if needed. During the Sprint Review, the Scrum Team and stakeholders collaborate about what was done in the Sprint. Based on that and any changes to the Product Backlog during the Sprint, attendees collaborate on the next things that could be done.

The result of the Sprint Review is a revised Product Backlog that defines the probable Product Backlog items for the next Sprint. The Product Backlog may also be adjusted overall to meet new opportunities.

4.3 Feature driven development (FDD)

Feature driven development is an iterative and incremental software development process. It is one of a number of agile methods. FDD blends a number of industry recognized best practices into a cohesive whole. The description of FDD was first introduced by Peter Coad, Eric Lefebvre and Jeff De Luca in 1999 [21].

FDD is a model-driven short-iteration process that consists of five basic activities. For accurate state reporting and keeping track of the software development project, milestones that mark the progress made on each feature are defined. Figure 4.5. shows the whole development process. During the first two sequential activities, an overall model shape is established. The final two activities are iterated for each feature.

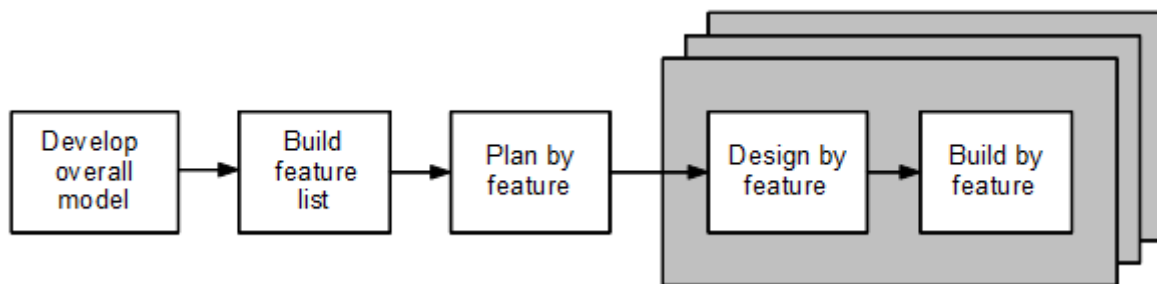


Figure 4.5. Process of feature driven development.

The detailed activities of development process are the followings:

Develop overall model

The project started with a high-level walkthrough of the scope of the system and its context. Next, detailed domain walkthroughs were held for each modelling area. In support of each domain, walkthrough models were then composed by small groups, which were presented for peer review and discussion. One of the proposed models, or a merge of them, was selected which became the model for that particular domain area. Domain area models were merged into an overall model, and the overall model shape was adjusted along the way.

Build feature list

The knowledge that was gathered during the initial modelling was used to identify a list of features. This was done by functionally decomposing the domain into subject areas. Subject areas each contain business activities, the steps within each business activity formed the categorized feature list. Features in this respect were small pieces of client-valued functions expressed in the form "<action> <result> <object>", for example: "Calculate the total of a

sale” or “Validate the password of a user”. Features should not take more than two weeks to complete, else they should be broken down into smaller pieces.

Plan by feature

After the feature list had been completed, the next step was to produce the development plan. Class ownership has been done by ordering and assigning features (or feature sets) as classes to chief programmers.

Design by feature

A design package was produced for each feature. A chief programmer selected a small group of features that are to be developed within two weeks. Together with the corresponding class owners, the chief programmer worked out detailed sequence diagrams for each feature and refines the overall model. Next, the class and method prologues are written and finally a design inspection is held.

Build by feature

After a successful design inspection a per feature activity to produce a completed client-valued function (feature) is being produced. The class owners develop the actual code for their classes. After a unit test and a successful code inspection, the completed feature is promoted to the main build.

4.3.1 Milestones

Since features are small, completing a feature is a relatively small task. For accurate state reporting and keeping track of the software development project it is however important to mark the progress made on each feature. FDD therefore defines six milestones per feature that are to be completed sequentially. The first three milestones are completed during the Design By Feature activity, the last three are completed during the Build By Feature activity. To help with tracking progress, a percentage complete is assigned to each milestone. In the table below the milestones (and their completion percentage) are shown. A feature that is still being coded is 44% complete (Domain Walkthrough 1%, Design 40% and Design Inspection 3% = 44%).

Table 4.2. Milestones of FDD.

Domain Walkthrough	Design	Design Inspection	Code	Code Inspection	Promote To Build
1%	40%	3%	45%	10%	1%

Feature-Driven Development is built around a core set of industry-recognized best practices, derived from software engineering. These practices are all driven from a client valued feature perspective. It is the combination of these practices and techniques that makes FDD so compelling. The best practices that make up FDD are shortly described below. For each best practice a short description is given:

- *Domain Object Modelling.* Domain Object Modelling consists of exploring and explaining the domain of the problem to be solved. The resulting domain object model provides an overall framework in which to add features.
- *Developing by Feature.* Any function that is too complex to be implemented within two weeks is further decomposed into smaller functions until each sub-problem is small enough to be called a feature. This makes it easier to deliver correct functions and to extend or modify the system.

- *Individual Class (Code) Ownership.* Individual class ownership means that distinct pieces or grouping of code are assigned to a single owner. The owner is responsible for the consistency, performance, and conceptual integrity of the class.
- *Feature Teams.* A feature team is a small, dynamically formed team that develops a small activity. By doing so, multiple minds are always applied to each design decision and also multiple design options are always evaluated before one is chosen.
- *Inspections.* Inspections are carried out to ensure good quality design and code, primarily by detection of defects.
- *Configuration Management.* Configuration management helps with identifying the source code for all features that have been completed to date and to maintain a history of changes to classes as feature teams enhance them.
- *Regular Builds.* Regular builds ensure there is always an up to date system that can be demonstrated to the client and helps highlighting integration errors of source code for the features early.
- *Visibility of progress and results.* By frequent, appropriate, and accurate progress reporting at all levels inside and outside the project, based on completed work, managers are helped at steering a project correctly

4.4 Exercises

1. List the main values of Agile Manifesto!
2. Explain the process of iterative software development!
3. What are the disadvantages of iterative developments?
4. What is the refactoring?
5. What are the main values of Extreme Programming:
6. What are the benefits of Pair Programming?
7. Explain the meaning of test-first development!
8. Explain the meaning of Scrum sprint!
9. List the basic activities of FDD process!
10. What are the milestones of FDD process?

5 Object-oriented development

The terms object and object-oriented are applied to different types of entity, design methods, systems and programming languages. An object-oriented system is made up of interacting objects that maintain their own local state and provide operations on that state. There is a general acceptance that an object is an encapsulation of information. The representation of the state is private and cannot be accessed directly from outside the object [1,2,4].

An object is an *entity* that has a state and a defined set of operations that operate on that state. The state is represented as a set of object *attributes*. The *operations* associated with the object provide services to other objects that request these services when some computation is required. Objects communicate by exchanging the information required for service provision. The copies of information needed to execute the service and the results of service execution are passed as parameters.

Objects are created according to an object *class* definition. An object class definition is both a type specification and a template for creating objects. It includes declarations of all the attributes and operations that should be associated with an object of that class.

Object-oriented design processes involve designing object classes and the relationships between these classes. These classes define the objects in the system and their interactions. When the design is realised as an executing program, the objects are created dynamically from these class definitions.

Object-oriented design is part of object-oriented development where an object-oriented strategy is used throughout the development process:

- *Object-oriented analysis*. It is concerned with developing an object-oriented model of the application domain. The objects in that model reflect the entities and operations associated with the problem to be solved.
- *Object-oriented design*. It is concerned with developing an object-oriented model of a software system to implement the identified requirements.
- *Object-oriented programming*. It is concerned with implementing a software design using an object-oriented programming language, such as Java.

The transition between these stages of development should, ideally, be seamless, with compatible notations used at each stage. Moving to the next stage involves refining the previous stage by adding detail to existing object classes and devising new classes to provide additional functionality.

The object-oriented systems might have some advantages in comparison to systems, which have been developed with other methods, for example realizing changes with an object-oriented system can be easier because the objects are independent. They may be understood and modified as standalone entities. Changing the implementation of an object or adding services should not affect other system objects.

Objects are, potentially, reusable components because they are independent encapsulations of state and operations. Designs can be developed using objects that have been created in previous designs. This reduces design, programming and validation costs. It may also lead to the use of standard objects and reduce the risks involved in software development.

In 1990s several object-oriented design methods have been proposed and the Unified Modelling Language (UML) became a unification of the notations used in these methods. In year 2000 the UML was accepted by the International Organization for Standardization (ISO) as a standard for modelling software-intensive systems [3,4,5,6,7,9,18].

In the UML, an object class is represented as a named rectangle with two sections. The object attributes are listed in the top section. The operations that are associated with the object are set out in the bottom section. Figure 5.1. illustrates this notation using an object class that models an employee in an organisation. The UML uses the term *operation* to mean the specification of an action; the term *method* is used to refer to the implementation of an operation.

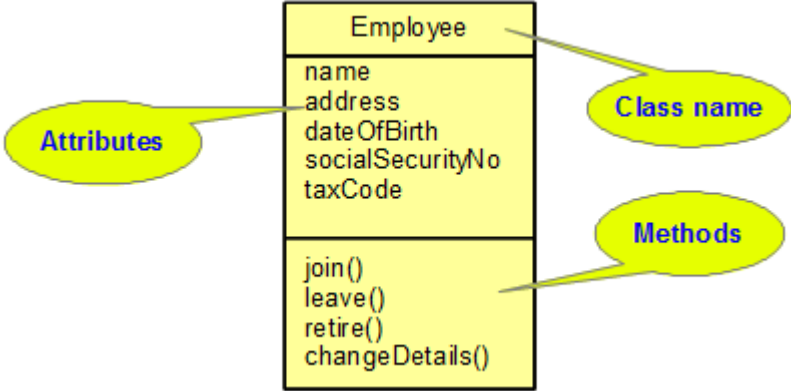


Figure 5.1. Representation of class Employee.

The class Employee defines a number of attributes that hold information about employees including their name and address, social security number, tax code, and so on. Operations associated with the object are join (called when an employee joins the organisation), leave (called when an employee leaves the organisation), retire (called when the employee becomes a pensioner of the organisation) and changeDetails (called when some employee information needs to be modified). For examples, in Figure 5.2. two objects of class Employee are shown.

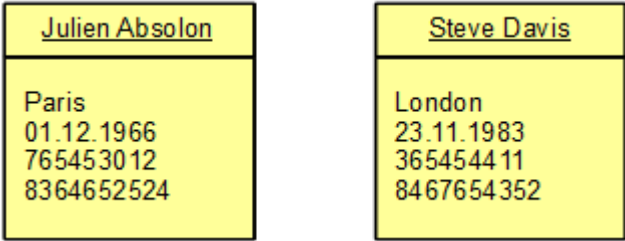


Figure 5.2. Instances of class Employee.

There are a lot of object-oriented systems where classes may have methods with similar name. In this case, the services of this method are determined by the implementation of methods in the classes. This is called *polymorphism* in the object-oriented design. In practice, polymorphism appears at the design of the class hierarchy by inheritance. Figure 5.3. shows an example of polymorphism. Classes Circle, Triangle and Square have the same method called area(). These methods are implemented differently in all classes by application of different expression for calculating the area.

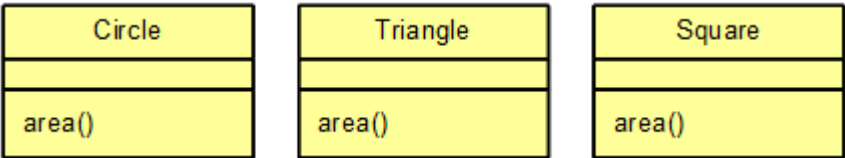


Figure 5.3. Example of polymorphism.

Object-oriented systems are usually modelled by interacting objects. An *object model* represents the static and most stable phenomena in a modelled domain. Main concepts are

classes and associations with attributes and operations. Aggregation and generalization with multiple inheritances are predefined relationships. In UML the objects and their static relationships are presented by *class diagrams*.

UML supports *stereotypes*, which are an inbuilt mechanism for logically extending or altering the meaning, display, characteristics or syntax of a basic UML model element such as class and object. Extending classes in UML the next stereotypes are often used:

- <<boundary>> stereotype. A boundary class represents a user interface.
- <<entity>> stereotype. Entity classes represent manipulated units of information.
- <<control>> stereotype. Control objects encapsulate logic that is not particularly related to user interface issues.

UML representation of stereotypes applied to classes Car, Login and Bank account are shown in Figure 5.4. together with an icon for their default representation.

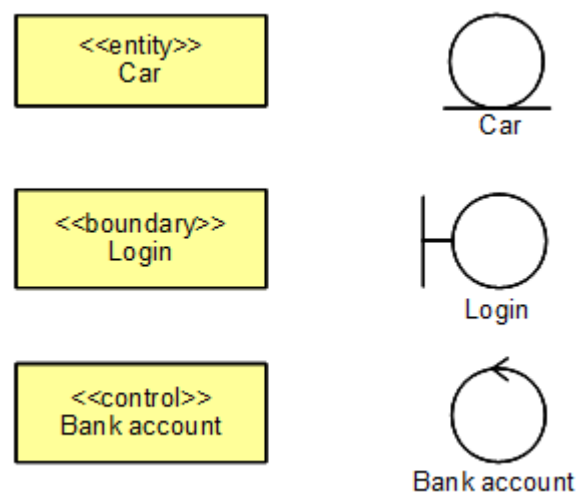


Figure 5.4. Representation of stereotypes.

Dynamic models describe the dynamic structure of the system and show the interactions between the system objects. Interactions that may be documented include the sequence of service requests made by objects and the way in which the state of the system is related to these object interactions. UML uses interaction diagrams such as sequence diagram to model dynamic behaviour.

The interactions of a system with its environment can be modelled using an abstract approach that does not include too much detail. The approach that is proposed in the RUP is to develop an *use-case* model where each use-case represents an interaction with the system. Each of these use-cases can be described in structured natural language. This helps designers identify objects in the system and gives them an understanding of what the system is intended to do

5.1 Generalisation

Object classes can be arranged in a generalisation or inheritance hierarchy that shows the relationship between general and more specific object classes. The more specific object class is completely consistent with its parent class but includes further information. In the UML, an arrow that points from a class entity to its parent class indicates generalisation. In object-oriented programming languages, generalisation is implemented using inheritance. The child class inherits attributes and operations from the parent class.

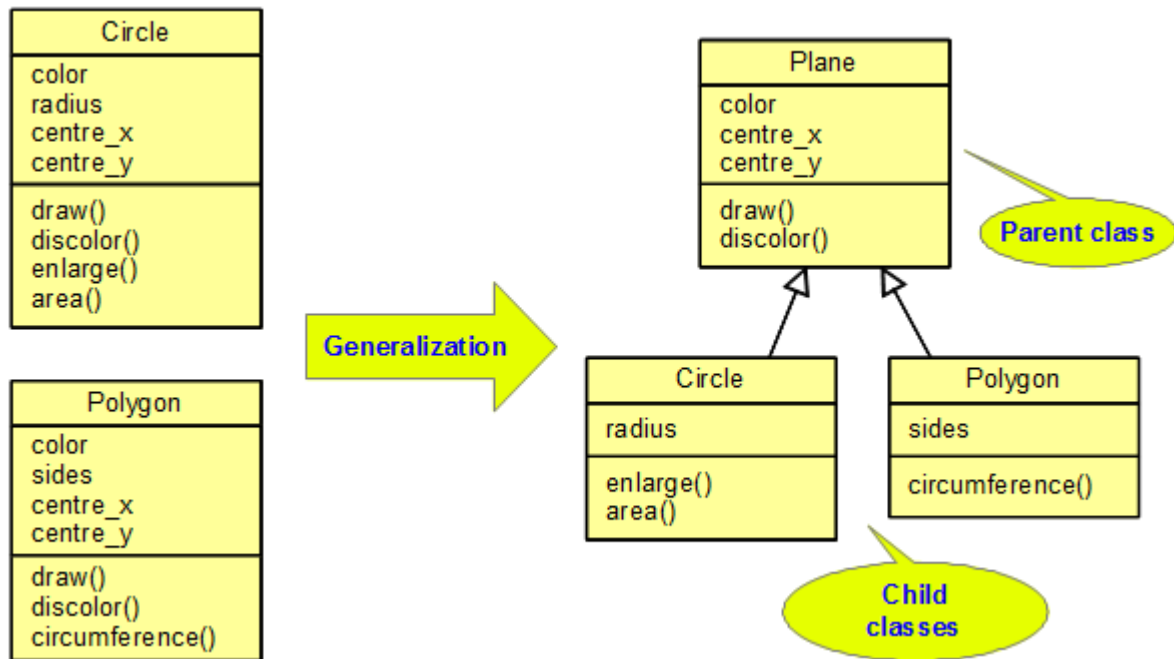


Figure 5.5. A generalisation hierarchy.

Figure 5.5. shows an example of an object class hierarchy where different classes of plane are shown. Classes lower down the hierarchy have the same attributes and operations as their parent classes but may add new attributes and operations or modify some of those from their parent classes. Class Circle and Polygon contain same attributes and methods. Generalizing these classes the same attributes and methods are added the class Plain. In this hierarchy the class Plain is considered as parent class and classes Circle and Polygon are child classes. The relationship of generalization is denoted by a hollow triangle shape on the parent class end of the line that connects it to one or more child classes.

The generalisation relationship is often used to create base classes in order to derive subclasses. These classes are called *abstract* classes and generally cannot be instantiated, but they can be subclasses.

5.2 Association

Objects that are members of an object class participate in relationships with other objects. These relationships may be modelled by describing the associations between the object classes. In the UML, associations are denoted by a line between the object classes that may optionally be annotated with information about the association. Association is a very general relationship and is often used in the UML to indicate that either an attribute of an object is an associated object or the implementation of an object method relies on the associated object. This association relationship indicates that at least one of the two related classes make reference to the other. The association relationship may have and an optional notation at each end indicating the *multiplicity* of instances of that entity (the number of objects that participate in the association). Multiplicity is given by an integer or an interval of integers. Examples multiplicities are given in the following:

- 1, exactly one instance participates in the association
- 0..1, no instances, or one instance participates in the association
- 0..*, zero or more instances participates in the association.
- 9, exactly 9 instances participates in the association.
- 3..9, the number of instances in the association between 3 and 9.

An example of association relationship is illustrated in Figure 5.6, which shows the association between objects of class Firm and objects of class Manager and Worker. This association represents relationships of such firms, managers and workers where 10 or more workers and 1, 2 or 3 managers work. A worker can be employed only at one firm but managers can be employed at two firms at the same time.

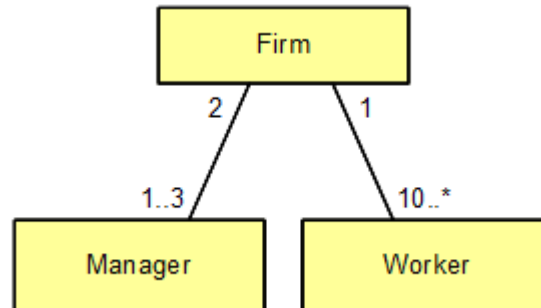


Figure 5.6. Representation of an association model.

One of the most common associations is aggregation that represents a part-whole or part-of relationship and illustrates how objects may be composed of other objects. Composition is a stronger variant of aggregation. Composition usually has a strong life cycle dependency between instances of the container class and instances of the contained class(es). If the container is destroyed, normally every instance that it contains is destroyed as well.

5.3 Object-oriented design process

A software development methodology is an organized process of fundamental activities of specification, development, validation and evolution and represents them as separate process phases such as requirements specification, software analysis and design, implementation, testing and so on.

The purpose of requirements specification phase is to define the complete functional and non-functional software requirements for the system. Domain requirements are a subcategory of requirements. These are requirements that come from the application domain of the system and that reflect characteristics and constraints of that domain.

The overall purpose of the analysis and design phases is to understand and model the application domain and to translate the requirements into a specification of how to implement the system. Analysis and design phases result in analysis and design model. Analysis is mainly involved with transforming the requirements into an architecture and collection of components that could fully support an implementation of the proposed system. The design model serves as an abstraction of the source code, it consists of design classes structured into design packages and design subsystems with well-defined interfaces, representing what will become components in the implementation.

The general process for an object-oriented design has a number of stages:

1. Understand and define the context and the modes of use of the system.
2. Design the system architecture.
3. Identify the principal objects in the system.
4. Develop design models.
5. Specify object interfaces.

Although these activities look separate stages the object-oriented design is not a simple, well-structured process. All of the above activities are interleaved and so influence each other. Objects are identified and the interfaces fully or partially specified as the architecture of the

system is defined. As object models are produced, these individual object definitions may be refined, which leads to changes to the system architecture.

5.3.1 System context and models of use

The first stage in any software design process is to develop an understanding of the relationships between the software and its external environment. This helps to provide the required system functionality and structure the system to communicate with its environment. The system context and the model of system use represent two complementary models of the relationships between a system and its environment:

1. The system context is a static model that describes the other systems in that environment.
2. The model of the system use is a dynamic model that describes how the system actually interacts with its environment.

The context model of a system may be represented using associations where a simple block diagram of the overall system architecture is produced. The interactions of a system with its environment can be modelled by use-case models where each use-case represents an interaction with the system. Use-cases can be described in structured natural language. This helps designers identify objects in the system and gives them an understanding of what the system is intended to do.

5.3.2 Architectural design

The system architecture can be represented by a number of architectural views. These views capture the major structural design decisions. Architectural views are the abstractions or simplifications of the entire design, in which important characteristics are made more visible by leaving details aside.

Design activities are centred around the notion of architecture. Once the interactions between the software system and its environment have been defined by use case models, this information can be used as a basis for designing the system architecture. During architectural design the architecturally significant use cases are selected by performing use case analysis on each one and the system is decomposed to a collection of interacting components.

In UML the architecture of system can be represented by a component diagram. It shows a collection of model elements, such as components, and implementation subsystems, and their relationships, connected as a graph to each other. Component diagrams can be organized into, and owned by implementation sub-systems, which show only what is relevant within a particular implementation subsystem.

5.3.3 Object and class identification

The analysis process starts with the identification of a set of conceptual classes that are the categories of things which are of significance in the system domain. When identifying appropriate classes, a good understanding of the system domain is important.

Possible classes for inclusion may emerge during the course of the requirements specification process. There are a number of other techniques can be used to help identify appropriate classes from a requirements document. Because a class is conceptually a set of objects of the same kind and objects represent things in the system domain, nouns and noun phrases in the requirements document can be used to identifying possible classes. There have been various proposals to identify object classes:

1. Grammatical analysis of a natural language description of a system. Objects and attributes are nouns; operations or services are verbs.

2. Using tangible entities, things, in the application domain such as car, roles such as worker, events such as request, interactions such as meetings, locations such as offices and so on.
3. Using a behavioural approach where the designer first understands the overall behaviour of the system. The various behaviours are assigned to different parts of the system and an understanding is derived of who initiates and participates in these behaviours. Participants who play significant roles are recognised as objects.
4. Using a scenario-based analysis where various scenarios of system use are identified and analysed in turn. As each scenario is analysed, the team responsible for the analysis must identify the required objects, attributes and operations.

These approaches help to get started with object identification. Further information from application domain knowledge or scenario analysis may then be used to refine and extend the initial objects. This information may be collected from requirements documents, from discussions with users and from an analysis of existing systems.

5.3.4 Design models

Design models show the objects or object classes in a system and the relationships between these entities. The design of system essentially consists of design models. Design models can be considered as a transformation of system requirements to a specification of how to implement the system. Design models have to be abstract enough to hide the unnecessary details. However, they also have to include enough detail for programmers to make implementation decisions.

In general, models are developed at different levels of detail. An important step in the design process, therefore, is to decide which design models has to be developed and the level of detail of these models. This usually depends on the type of system that is being developed. There are two types of models that should be produced to describe an object-oriented design:

1. *Static models.* They describe the static structure of the system using object classes and their relationships. Important relationships are the association, generalisation, dependency, aggregation and composition relationships.
2. *Dynamic models.* They describe the dynamic structure of the system and show the interactions between the system objects. Interactions that may be documented include the sequence of service requests made by objects and the way in which the state of the system is related to these object interactions.

The UML provides for 14 different static and dynamic models that may be produced to document a design. For examples, some models:

1. *Sub-system models.* This model shows logical groupings of objects into coherent sub-systems. These are represented using a form of class diagram where each sub-system is shown as a package. Sub-system models are static models.
2. *Sequence models.* This model shows the sequence of object interactions. These are represented using an UML sequence or a collaboration diagram. Sequence models are dynamic models.
3. *State machine models.* This model shows how individual objects change their state in response to events. These are represented in the UML using state-chart diagrams. State machine models are dynamic models.

Other models will be discussed in the next chapters: use-case models show interactions with the system; object models describe the object classes; generalisation or inheritance show how classes may be generalisations of other classes; and aggregation models show how collections of objects are related.

5.3.5 Object interface specification

Software components communicate each other using their interfaces. Object interface design is concerned with specifying the detail of the interface to an object or to a group of objects.

The same object may have several interfaces, each of which is a viewpoint on the methods that it provides. Equally, a group of objects may all be accessed through a single interface.

The interface representation should be hidden and object operations are provided to access and update the data. If the representation is hidden, it can be changed without affecting the objects that use these attributes. This leads to a design that is inherently more maintainable.

Interfaces can be specified in the UML using the same notation as in a class diagram. However, there is no attribute section, and the UML stereotype <<interface>> should be included in the name part.

5.4 Exercises

1. What is the different objects and object classes?
2. Explain the information encapsulation of objects!
3. Can abstract classes be instantiated?
4. What is the stereotyping?
5. What is the meaning of polymorphism?
6. Explain the importance of generalisation of classes!
7. What types of association exist between classes?
8. What are the general stages of object-oriented design process?
9. How objects and classes can be identified?
10. What is the functionality of interface classes?

6 Unified Modelling Language

Due to the rapid development in computer hardware the complexity of computer-based systems continuously increases. New technologies resulting from developments place new demands on software and software engineers. Producing and maintaining high-quality software systems cost-effectively is essential for software industry.

Activities involved in software development such as software specification, design and implementation have already well understood to produce large and complex software systems.

But application of unified notations, languages and modelling tools can reduce the development costs effectively. The Unified Modelling Language (UML) is such a language. In the last decades, the most significant developments in software engineering are related the emergence of the UML as a standard for object-oriented system description [3,4,5,6,7,9,18].

In the years after 1980 many object-oriented modelling techniques were being developed to model software development processes. These modelling techniques used different visual modelling techniques and notations.

Three methodologies among them have become popular: the Object Modelling Technique (Jim Rumbaugh), the Object-Oriented Software Engineering method (Ivar Jacobson) and Booch method (Grady Booch). In the mid of 1990s, Rational Software integrated the methods of Object Modelling Technique and Booch method into version 0.8 of UML what was then called the Unified Method. In 1995, Jacobson, who developed Object-Oriented Software Engineering method, joined Rational Software and together with Rumbaugh and Booch developed version 0.9 of the Unified Method in 1996. Later other IT companies joined the UML Consortium developing UML. In 1997 they submitted version 1.0 of the Unified Method, renamed as the Unified Modelling Language to the independent standards body Object management Group. As an independent standards body, the OMG took over the UML development and released subsequent versions of the UML. The latest version, UML 2.5, is released by OMG in 2013.

Although the UML is a standard OMG the UML is not just for modelling object-oriented software applications. The UML is a graphical language that was designed to be very flexible and customizable.

The UML features an underlying meta-model that enables the UML to be flexible enough so that you can do what you need to do with it.

A meta-model is a model of a model. The UML meta-model expresses the proper semantics and structure for UML models. A UML model is made up of many different elements. The meta-model defines the characteristics of these elements, the ways in which these elements can be related, and what such relationships mean. In other words, the UML meta-model sets the rules for how it can be used to model.

The UML meta-model is also the foundation for UML's extensibility. Using the definitions of UML elements in the meta-model new UML modelling elements can be created. We can add additional properties to the new elements. This allows to give additional characteristics and behaviours to the new element for specific needs, while it still remains compliant with the structure and semantics of the original element that it was based upon. In this way, users can customize the UML to their specific needs.

The ability of extensibility makes UML to accommodate present as well as future needs. It makes possible to model different systems such as software and business system in the same language that helps the communications of developers and helps easier to understand developments of different areas.

6.1 Model, modelling

In a general sense, a *model* is anything used in any way to represent anything else. A model is considered a simplified, abstract view of a complex reality. It can represent objects, phenomena, and processes in a logical way. Modelling helps to better understand the processes and to develop systems that have to meet prescribed requirements. Modelling helps to identify how changes will affect the system. It can help to identify strengths and weaknesses, identify areas that need to be changed or optimized and simulate different process options. It also helps to communicate designs, to clarify complex problems, and ensure that designs come closer to reality prior to implementation. This can save a lot of time and money for the organization, and it enables teams of people to work together more effectively. In software industry modelling is primarily used to visually model the static and dynamic views of software.

UML provides a common modelling language to bring together business analysts, software developers, architects, testers, database designers, and the many other professionals who are involved in software design and development so that they can understand the business, its requirements, and how the software and architectures will be created. A business analyst who practiced in using UML can understand what a software developer is creating using the UML because the UML is a common language. With the ongoing need to think globally when building software, the ability provided by the UML to communicate globally becomes very important.

The UML can be applied to model business processes, database design, application architectures, hardware designs, and much more. Designing software and systems is a complicated task requiring the coordinated efforts of different groups performing various activities: define of requirements for business and systems, designing software components, constructing databases, assembling hardware to support the systems, and so on.

In UML different types of diagrams can be used to create various types of models. The UML models consist of different diagram types, model elements, and linkages between model elements. These models are to describe different information from different viewpoints.

6.2 Diagrams in UML

UML diagrams represent two different views of a system model:

1. Static (or *structural*) view: emphasizes the static structure of the system using objects, attributes, operations and relationships. The structural view includes class diagrams and composite structure diagrams.
2. Dynamic (or *behavioural*) view: emphasizes the dynamic behaviour of the system by showing collaborations among objects and changes to the internal states of objects. This view includes sequence diagrams, activity diagrams and state machine diagrams.

UML has 14 types of diagrams divided into two categories. Seven diagram types represent *structural* information, and the other seven represent general types of *behaviour*, including four that represent different aspects of *interactions*.

UML does not restrict UML element types to a certain diagram type. In general, every UML element may appear on almost all types of diagrams. UML profiles may define additional diagram types or extend existing diagrams with additional notations.

Structure diagrams emphasize the things that must be present in the system being modelled. Since structure diagrams represent the structure, they are used extensively in documenting the software architecture of software systems. The UML structure diagrams are the followings:

1. *Class diagram* describes the structure of a system by showing the system's classes, their attributes, and the relationships among the classes.

2. *Component diagram* describes how a software system is split up into components and shows the dependencies among these components.
3. *Composite structure diagram* describes the internal structure of a class and the collaborations that this structure makes possible.
4. *Deployment diagram* describes the hardware used in system implementations and the execution environments and artifacts deployed on the hardware.
5. *Object diagram* shows a complete or partial view of the structure of an example modelled system at a specific time.
6. *Package diagram* describes how a system is split up into logical groupings by showing the dependencies among these groupings.
7. *Profile diagram* operates at the meta-model level to show stereotypes as classes with the <<stereotype>> stereotype, and profiles as packages with the <<profile>> stereotype. The extension relation (solid line with closed, filled arrowhead) indicates what meta-model element a given stereotype is extending.

Behaviour diagrams emphasize what must happen in the system being modelled. Since behaviour diagrams illustrate the behaviour of a system, they are used extensively to describe the functionality of software systems:

1. *Activity diagram* describes the business and operational step-by-step workflows of components in a system. An activity diagram shows the overall flow of control.
2. *State machine diagram* describes the states and state transitions of the system.
3. *Use Case diagram* describes the functionality provided by a system in terms of actors, their goals represented as use cases, and any dependencies among those use cases.

Interaction diagrams, a subset of behaviour diagrams, emphasize the flow of control and data among the things in the system being modelled:

1. *Communication diagram* shows the interactions between objects or parts in terms of sequenced messages. They represent a combination of information taken from Class, Sequence, and Use Case Diagrams describing both the static structure and dynamic behaviour of a system.
2. *Interaction overview diagram* provides an overview in which the nodes represent communication diagrams.
3. *Sequence diagram* shows how objects communicate with each other in terms of a sequence of messages. Also indicates the lifespans of objects relative to those message.
4. *Timing diagram* a specific type of interaction diagram where the focus is on timing constraints.

6.3 Business models

UML is frequently used in the area of business modelling. Examples of simple business models can effectively present the practical use of UML and provide a good starting point to solve more complex developments. The purpose of this section is to introduce the use of UML diagrams, model elements, and linkages through a simple business modelling example. More examples provided in the next chapters to understand UML in more detail.

Business model is an abstract representation of a business that provides a simplified look at various aspects of the business. A business isn't represented with just one type of business model. Different models will emphasize certain business characteristics or concepts while hiding other aspects at the same time. The simplest business models are the organization charts, which are models of a business's overall organizational structure. Other models such as business process models show specific business functions by sequence of activities.

Each model provides a different view of the business. In the case of an overall business model it is requested to model the structure and interactions between the business's organizations, stakeholders, customers, workers, business functions, business assets, etc.

The business model should also reflect the architecture, the static structures in the company, the flows of business activities, that is, the dynamic behaviour of the elements in the architecture.

When modelling a business using the UML first the next questions is generally considered:

1. Who will be related to business?
2. What do they want your business to do for them?
3. How does your business meet its needs?

For an example, let us consider an retail shop. People, like customer, seller, shipping worker, etc. or systems, like shipping companies, suppliers, etc. can relate this business. All these people and systems play a *role* related to this business and they are called *business actors*. Figure 6.1. shows UML representation of business actors.

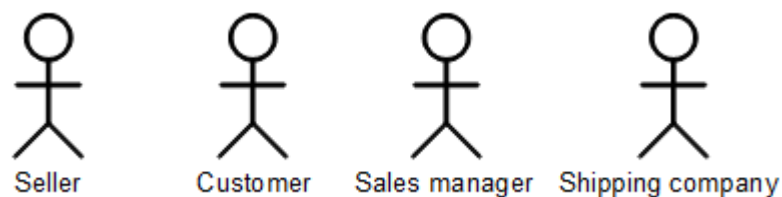


Figure 6.1. Business actors related to retail shop.

The reasons or goals for business actors interacts retail shop may be the followings:

- customers purchase products,
- sellers billing,
- sales manager orders goods,
- shipping companies deliver products to retail shop,
- bank manages account,
- and more.

For these needs retail shop provides services or business. Some typical business functions in retail might be the following:

- Goods order,
- Ship order,
- Shipping,
- Billing,
- Storage,
- And more.

These actions are called *business use cases* when business actors will use the business. Figure 6.2. shows the UML representation of business use cases.

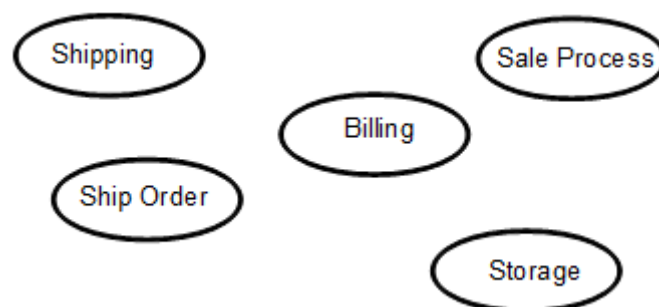


Figure 6.2. UML business use cases for retail shop.

6.3.1 Business use case diagrams

The *business use case diagram* represents the functionality of business system in term of business actors. Their goals are represented by business use cases. Use case diagram also shows what is outside of business (the business actors), what is inside the business (the business use cases), and the relationships between the two (Figure 6.3.).

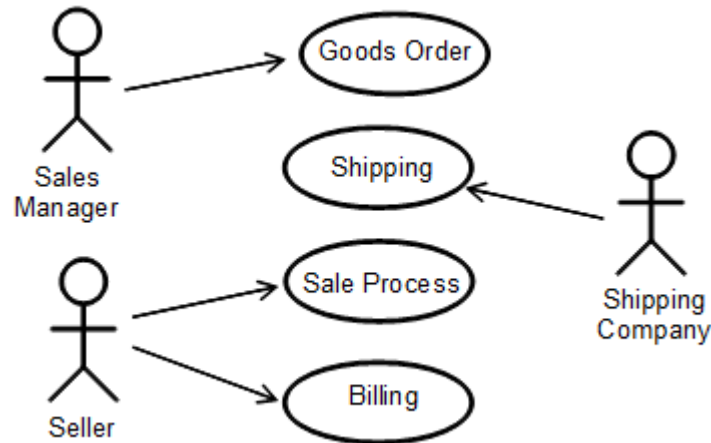


Figure 6.3. Business use case diagram.

The lines with arrowheads between a business actor and business use case are association. Associations define a relationship between the two model elements they connect. The direction of the arrow shows which element initiates the activity. In the example in Figure 6.3., the transport worker uses or initiates the Shipping business use case. An association can exist without an arrowhead representing a bi-directional communication path.

Besides association UML contains various linkages defining relationships between elements. They are listed in Table 6.1.

Table 6.1. Relationships used in UML.

Name	Notation	Description
Association	—————>	Logical connection or relationship between classes
Bi-directional association	—————	Bi directional logical connection or relationship between classes
Generalization	—————>▷	The generalization relationship indicates that one of the two related classes (the <i>subclass</i>) is considered to be a specialized form of the other (the <i>super type</i>) and superclass is considered as generalisation of subclass.
Dependency	- - - - ->	Dependency indicates that one class depends on another because it uses it at some point in time.
Aggregation	—————◊	Aggregation is an association that represents a part-whole relationship.
Composition	—————◆	Composition is a stronger variant of association relationship. Composition usually has a strong

		life cycle dependency between instances of the container class and instances of the contained class(es): If the container is destroyed, normally every instance that it contains is destroyed as well.
Realization	----->	A realization relationship is a relationship between two model elements, in which one model element (the client) realizes (implements or executes) the behaviour that the other model element (the supplier) specifies.

6.3.1.1 Use case relationships

Use cases can have relationships with other use cases. UML defines two directed relationships between use cases, the include and extend relationships.

For an example of these relationships let's again consider the retail shop business example. In business process models the specific business functions (business use cases) can be modelled by a flow of different activities and could have many different alternate scenarios. Both use case Goods Order and Ship Order certainly includes the activity Send Order. The *include* relationship in this respect means that the *included use case* (Send Order) behaviour is inserted in the flow of the *base use case(s)* (Goods Order, Ship Order).

This common behaviour can be represented by a dashed arrow with an open arrowhead from the including (base) use case to the included (common part) use case. The arrow is labelled with the keyword «include» (Figure 6.4.).

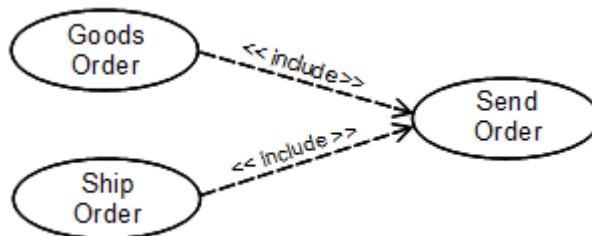


Figure 6.4. UML representation of include relationship.

Scenarios of business use cases in retail business system can be extended by many different optional behaviours. For example the use case Billing can be extended by the optional extending use case Payment by Card. This behaviour can be represented by the *extend* relationship. It means that the extending use case (Payment by Card) optionally may change the flow (scenario) of the base use case (Billing). Extend relationship is shown as a dashed line with an open arrowhead directed from the extending use case to the extended (base) use case. The arrow is labelled with the keyword «extend» (Figure 6.5.).

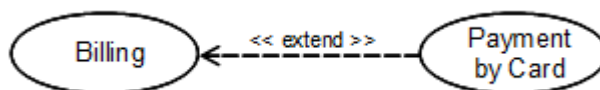


Figure 6.5. UML representation of extend relationship.

Includes and extends relationships are useful tools to structure use case models, identify common use cases and simplify complex scenarios.

6.3.1.2 Generalisation

Business actors are the people or any system that interact with business system. They are not a specific people or system they represent a *role* that the actor plays with respect to

business system. An actor can also represent a set of roles. In this case the actor can be represented by an actor who inherits several actors and each inherited actor represents one of the roles. In the other case several actors can play the same role in a particular use case. In this case the shared role can be modelled as an actor inherited by the original actors.







In the generalization of use cases a parent use case may be specialized into one or more child use cases that represent more specific forms of the parent. A child inherits all structure, behaviour, and relationships of the parent. Generalization is used when two or more use cases have common behaviour, structure, and purpose. In this case the shared parts are captured in a new abstract use case that can be specialized by child use cases.

6.3.1.3 Activity diagrams

Considering again the retail shop example, now we have known the people, businesses, and systems interacting the business and we know what services are provided to meet their needs. What we have to understand now is how business actors interacts the system to provide the business use cases. What are the steps taken and by whom? Business use cases are described by a flow of separate activities. Scenarios define these activities in a textual form and they are called actions in UML. For the graphical representation of activities UML uses activity diagrams.

Activity diagram makes the flow of activities visible and show how a use case is realised. The elements of UML activity diagram are shown in Table 6.2.

Table 6.2. Elements of activity diagram.

Notation	Description
	This element represents the start (initial state) of the workflow.
	The encircled black circle represents the end (final state) of the workflow.
	Rounded rectangles represent actions.
	Diamonds represent decision points.
	Synchronization point. Black bar represents the start (split) or end (join) of concurrent activities.
	Arrows run from the start towards the end and represent the order in which activities happen.

During design phase business use cases are usually described by alternate scenarios. For an example the use case Sale Process can be realised by the following scenario:

1. Customer enters retail shop.
2. Customer chooses products.
3. Customer presents products to seller.
4. Seller scans all products.
5. Seller provides total cost.
6. Seller inquires the payment method.
7. If customer provides payment by cash customer pass payment and seller accepts it, otherwise payment is by credit card and customer authorizes payment by giving PIN code.
8. Customer receives receipt and products.
9. Customer leaves shop.

This scenario gives the activity diagram shown in Figure 6.6. The activity diagram shows how Customer and Seller interacts the business system in the realization of use case Sale Process. The two business actors are shown at the top of the columns in the diagram. These columns are called *swimlanes*. Any *activity* in a given column is performed by the actor listed at the top of the swimlane. The flow starts at the *start state* and flows as indicated by the arrows.

At the *decision point* the flow of activities is controlled by the payment method required by Seller.

The diagram show two activities, Receive Receipt and Receive Products, can happen in parallel. This is shown in activity diagram by using a *synchronization point*. The two flows that come *out* of the bar indicate that they can happen independently. When two or more flows come *into* a synchronization point, it indicates that the workflow cannot continue until all the inflowing activities are complete. The last activity, Customer Leaves, is the *terminating activity* to the flow.

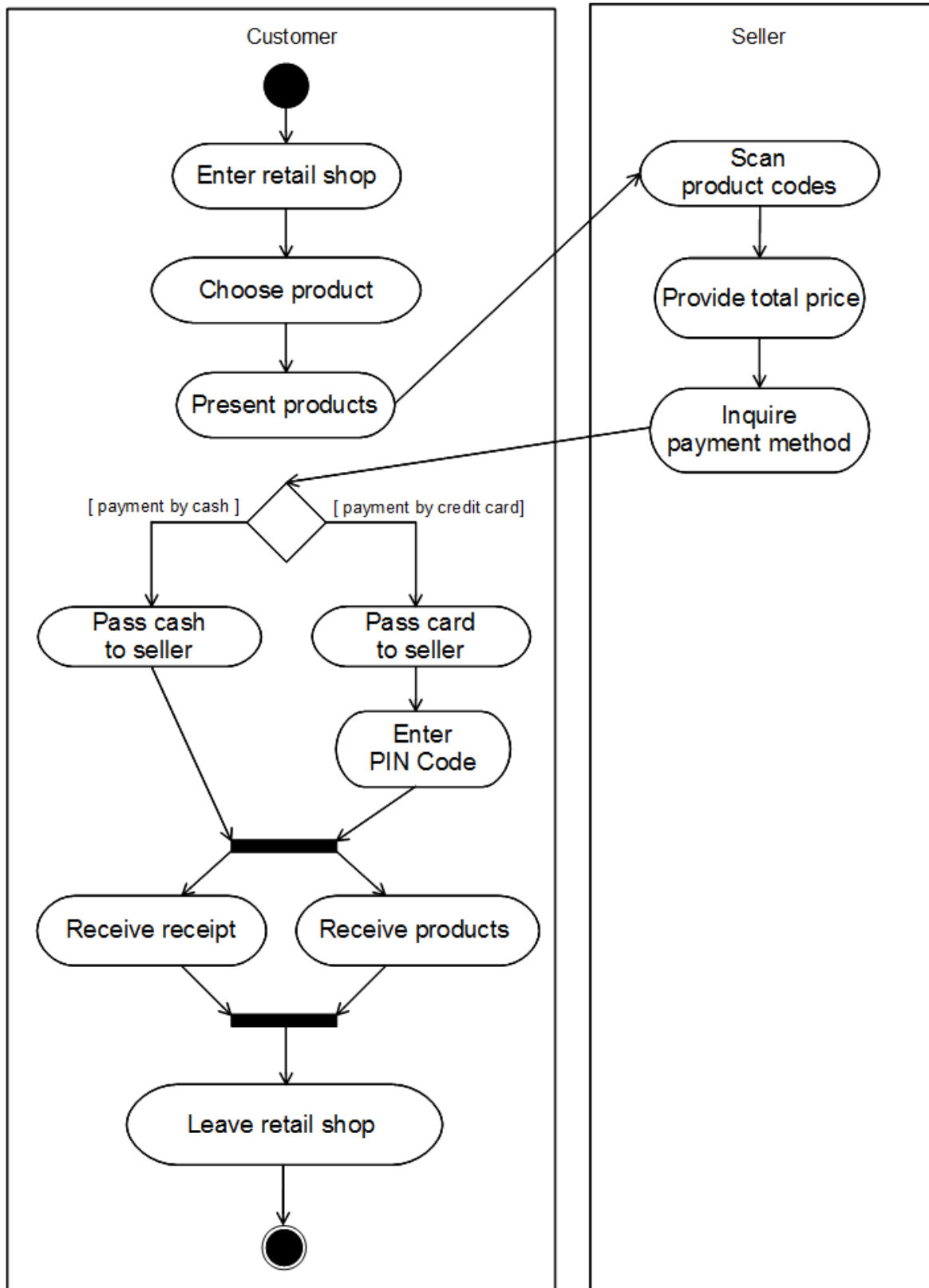


Figure 6.6. The activity diagram of use case Sale Process.

6.3.2 Business analysis model

After development of business use case models and realizing the use cases by activity diagrams we have the knowledge of how business actors outside of business want to use the

system, how they interact the system and what services the system provides for them. Now the purpose is to find what people, assets, etc. will be used to provide the services of business. Review of business use case diagrams and activity diagrams helps to determine business workers (internal people) and business entities (assets, information, etc.) that are involved in these activities. Figure 6.7. shows an example of business workers and business entities for the retail shop.

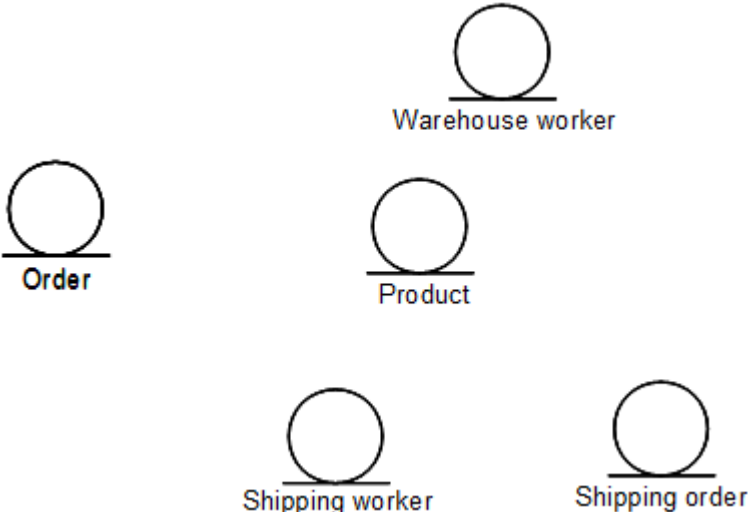


Figure 6.7. Business workers and entities for retail shop.

The relations between business workers and entities are shown in the *business analysis model*. This provides an inside look at how people interact with other business workers, business actors, and business entities to achieve the business processes defined in the business use case model.

A partial *business objects diagram* is shown in Figure 6.8. including business workers, business entities and their static relationships.

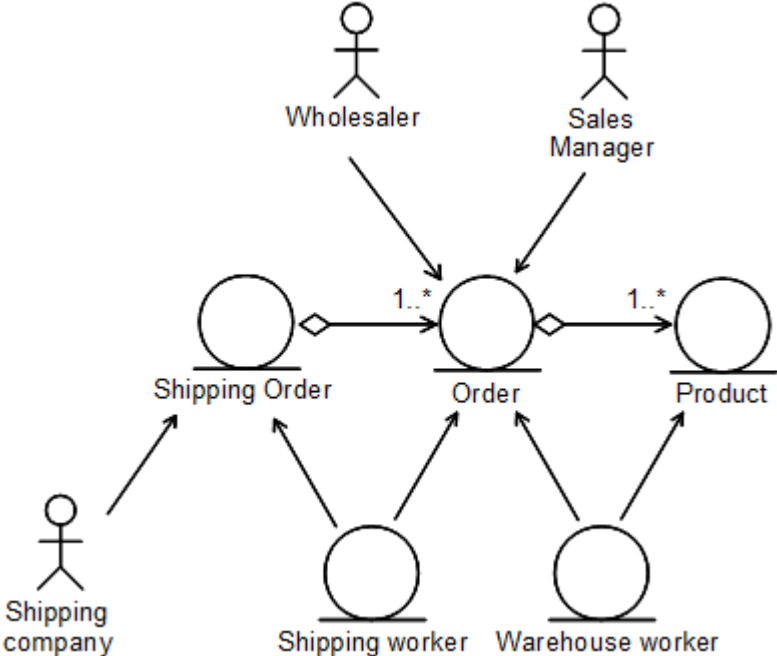


Figure 6.8. Business object diagram for retail shop.

6.3.2.1 Sequence diagrams

The business objects diagram presented in the previous section captured the static relationships between the business actors, workers and entities. The view of dynamic interactions between these elements over time is also an important aspect in system development. The dynamic views of the system can be represented by using a type of UML *interaction diagram* called *sequence diagram*. A sequence diagram shows all the interactions between the model elements for a given scenario arranged in a time sequence. It represents graphically how object interacts each other by messages [18].

For an example let's consider the sequence diagram shown in Figure 6.9. In the diagram a scenario of an online order process is captured. The purpose of using sequence diagram is the graphical representation of interactions between objects that realise the functionality of the given scenario. Objects that live simultaneously are shown as parallel vertical lifelines and horizontal arrows represent the messages exchanged between them.

Diagram shows how the online order process is realised in a time sequence:

1. An Order Entry Window object (instantiated by an business actor) send a message „prepare” to Order object
2. An Order object send a message „prepare” to OrderLine for all order line in the order.
3. Every Order Line objects send a „check” message to the related Stock Item object to check the amount of item in the stock. If the required amount of item is available the Stock Item reduces the amount of available item in the stock by the amount of ordered item. If the amount of item in stock is not sufficient Stock Item object creates a Reorder Item object and then Order Line object creates a new Delivery Item object.
4. It is shown in the diagram that an object may send message to itself. This event is indicated by a line returning back to itself and called self-delegation.
5. The diagram also shows an example of the use of iteration marker of *. Application of iteration marker provides a multiple send of message prepare() to Order Line object, i.e. Order object creates as many Order Line objects as included in order. In the diagram three conditional events are captured.
6. Stock Item object creates a Reorder Item object only if needsToReorder==true, Order Line object creates a new Delivery Item object if hasStock==false and Stock Item object reduces the amount of item if hasStock=true.
7. For an example the diagram contains a return arrow indicating that the control is given back to Order Line object after reorder Item objects is created.

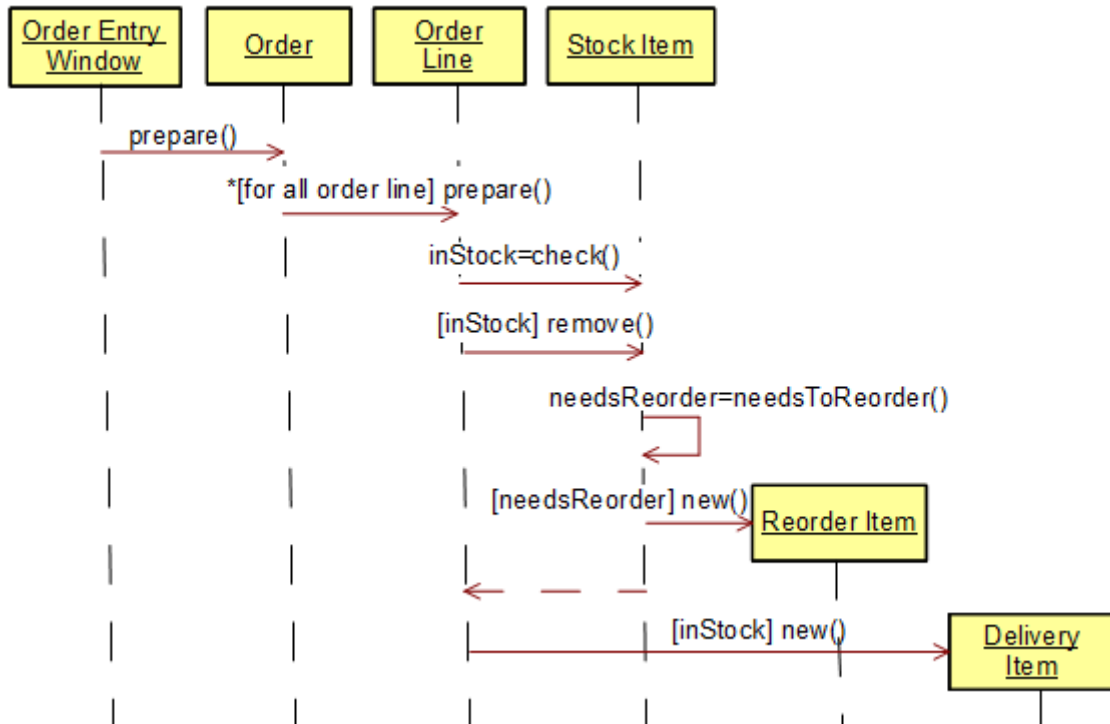


Figure 6.9. Sequence diagram of online order.

6.3.2.2 Concurrent processes

Sequence diagrams are also capable to represent dynamic interaction of concurrent objects. The example shown in Figure 6.10. depicts objects controlling transactions in a bank.

After the Transaction object is instantiated it creates (by new method) a Transaction Coordinator object. Transaction Coordinator has a function to check the transaction. It creates as many Transaction Checker object as several control task has to be done in accordance with bank rules. In this example two of them (first and second) are created. Both have the own independent control function. The two control tasks are carried out in parallel and there is no need for any timing or synchronization between them. They operate asynchronously. As one of Transaction Checker finishes its control task notifies the Transaction Coordinator object. At this time Transaction Coordinator object checks whether other Transaction Checker has finished its task. If both control task are finished Transaction Coordinator object send a message to Transaction object that transaction is OK.

The rectangles located along lifelines indicate a time interval during which object is activated. These rectangles are called activation. Some of the arrowhead on Figure 6.10. are incomplete. These half-arrowheads denote asynchronous messages. If an object sends an asynchronous message, it can continue processing and doesn't have to wait for a response. Asynchronous messages sent to an object start a new thread or communicate an existing thread.

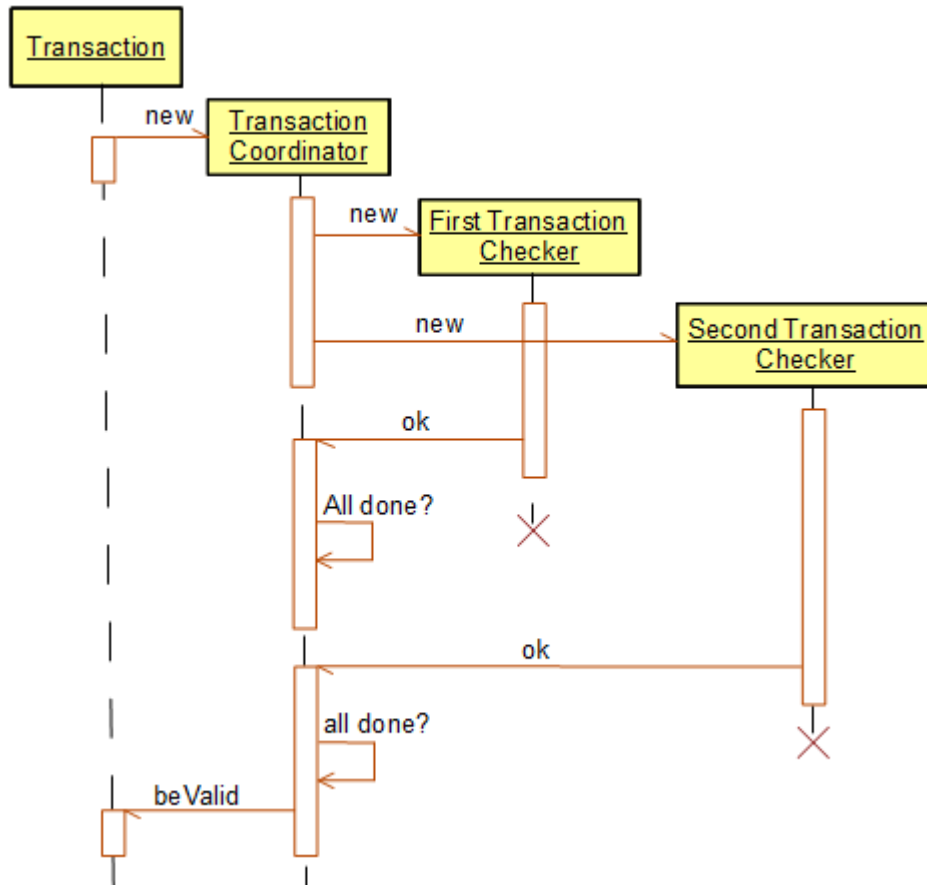


Figure 6.10. Representation of concurrent processes in sequence diagram.

The examples of other interaction diagrams will be presented in Chapter 8.

6.4 Exercises

1. What is the meaning of model?
2. What can behaviour diagrams be used for?
3. What can structure diagrams be used for?
4. Explain the concept of actor!
5. What relationship can be defined between actors?
6. What is different between use case model and use case diagram?
7. What can use case diagrams be used for?
8. List the graphical elements of activity diagram!
9. What can business analysis model used for and what elements it may has?
10. List the graphical elements of sequence diagram!
11. What does the „extend” relationship mean between two use cases?
12. What does the „include” relationship mean between two use cases?

7 Requirements analysis

The phases of requirements analysis and design are discussed in the frame of the Rational Unified Process (RUP) software development process in the next two chapters. RUP is based on the Unified Modelling Language (UML) as an object-oriented graphical modelling language, so the practical examples presented should not be considered general in the practice of other development methodologies [3,10,11,12,17,22].

The purpose of requirements discipline in RUP is to definition of the requirement specification capturing the complete functional and non-functional software requirements for the system.

It is important, first of all, to understand the definition and scope of the problem which we are trying to solve with this system. The models developed during Business Modelling can serve as valuable input to this effort. Business modelling helps to understand the structure and the dynamics of the organization in which a system is to be deployed and derive the system requirements needed to support the target organization.

A requirement is defined as a condition or capability to which a system must conform. Requirements management is a systematic approach to finding, documenting, organizing and tracking the changing requirements of a system.

There are many different kinds of requirements. Requirements are usually looked upon as statements of text fitting into one of the following categories:

1. *User requirements.* User requirements focus on the needs and goals of the end users. They are statements of what services the system is expected to provide and the constraints under which it must operate.
2. *System requirements.* They set out the system's functions, services and operational constraints in detail.

Software system requirements are often classified as functional and non-functional requirements:

1. *Functional requirements.* They specify actions that a system must be able to perform, without taking physical constraints into consideration. Functional requirements thus specify the input and output behaviour of a system.
2. *Non-functional requirements.* Many requirements are non-functional, and describe only attributes of the system or attributes of the system environment. They are often categorized as usability, reliability, performance, and substitutability requirements. They are often requirements that specify need of compliance with any legal and regulatory requirements. They can also be design constraints due to the operating system used, the platform environment, compatibility issues, or any application standards that apply.

Domain requirements are a subcategory of requirements. These are requirements that come from the application domain of the system and that reflect characteristics and constraints of that domain. They are formulated by domain-specific terminology. They may be functional or non-functional requirements.

The user requirements are provided by functional requirements. The functional requirements are often best described in a UML use-case model and in use cases. Every functionality of system is modelled by one or more use cases.

The general objectives of the requirements discipline is:

- To establish and maintain agreement with the customers and other stakeholders on what the system should do.
- To provide system developers with a better understanding of the system requirements.
- To define the boundaries of the system.
- To provide a basis for planning the technical contents of iterations.
- To provide a basis for estimating cost and time to develop the system.
- To define a user-interface for the system, focusing on the needs and goals of the users.

In the frame of RUP methodology the activities and artifacts are organized into workflow details as follows:

- Analyze the Problem
- Understand Stakeholder Needs
- Define the System
- Manage the Scope of the System
- Refine the System Definition
- Manage Changing Requirements

Each workflow detail represents a key skill that need to be applied to perform effective requirements management. Analyze the Problem and Understand Stakeholder Needs are focused on during the Inception phase of a project, whereas the emphasis is on Define the System and Refine the System Definition during the Elaboration phase. Manage the Scope of the System and Manage Changing Requirements are done continuously throughout the development project.

The workflow details are listed above in the sequence that is most likely applied to the first iteration of a new project. They are applied continuously in varied order as needed throughout the project.

7.1 Analyse the problem

Problem analysis is done to understand problems, initial stakeholder needs, and propose high-level solutions. It is an act of reasoning and analysis to find "the problem behind the problem". During problem analysis, agreement is gained on the real problems and stakeholders are identified. Also, initial boundaries of the solution and constraints are defined from both technical and business perspectives.

The purpose of this workflow detail is to:

- Gain agreement on the problem being solved,
- Identify stakeholders,
- Define the system boundaries, and
- Identify constraints imposed on the system.

7.1.1 Definition of glossary

The first step in any problem analysis is to make sure that all parties involved agree on what is the problem that we are trying to solve with our system. In order to avoid misunderstandings, it is important to agree on common terminology which will be used throughout the project and provide a consistent set of definitions for the system. Early on, we should begin defining our project terms in a glossary which will be maintained throughout the project lifecycle.

7.1.2 Find actors

In order to fully understand the problem, it is very important to identify the stakeholders. Some of these stakeholders, the users of the system, will be represented by actors in our use-case model.

A use-case model is a model of the system's intended functions and its surroundings, and serves as a contract between the customer and the developers. It describes a system's requirements in terms of use cases. The most important purpose of a use-case model is to communicate the system's behaviour to the customer or end user.

The functionality of a system is defined by different use cases, each of which represents a specific flow of events. The description of a use case defines what happens in the system when the use case is performed. Each use case has a task of its own to perform. The collected use cases constitute all the possible ways of using the system.

To fully understand the system's purpose it must be known who the system is for, that is, who will be using the system. Different user types are represented as actors. An *actor* defines a role that users of the system can play when interacting with it. An actor instance can be also played by either an individual or an external system. An actor may be anything that exchanges data with the system. The difference between an actor and an individual system user is that an actor represents a particular class of user rather than an actual user. Several users can play the same role, which means they can be one and the same actor. In that case, each user constitutes an instance of the actor.

The principal symbol representing an actor in UML is shown in Figure 7.1.



Figure 7.1. UML representation of an actor.

Finding the actors means the establishing the boundaries of the system, which helps in understanding the purpose and extent of the system. Only those who directly communicate with the system need to be considered as actors. The following set of questions is useful to have in mind when you are identifying actors:

- Who will supply, use, or remove information?
- Who will use this functionality?
- Who is interested in a certain requirement?
- Where in the organization is the system used?
- Who will support and maintain the system?
- What are the system's external resources?
- What other systems will need to interact with this one?

After identifying an actor a brief description of it should be given including information about:

- What or who the actor represents.
- Why the actor is needed.
- What interests the actor has in the system.

7.1.3 Develop requirements management plan

Since requirements are things to which the system being built must conform, it makes sense to find out what the requirements are, write them down, organize them, and track them

in the event they change. The requirements management emphasizes the importance of tracking changes to maintain agreements between stakeholders and the project team.

The effective requirements management include maintaining a clear statement of the requirements, along with applicable attributes for each requirement type and traceability to other requirements and other project artifacts. It provides guidance on the requirements artifacts that should be developed, the types of requirements that should be managed for the project, the requirements attributes that should be collected and the requirements traceability that will be used in managing the product requirements.

A Requirements Management Plan specifies the control mechanisms which will be used for measuring, reporting, and controlling changes to the product requirements.

7.1.4 Develop Project Vision document

The primary artifact in which we document the problem analysis information is the Vision document, which identifies the high-level user or customer view of the system to be built.

The Vision document provides a complete vision for the software system under development and supports the contract between the funding authority and the development organization. Every project needs a source for capturing the expectations among stakeholders. The vision document is written from the customer's perspective, focusing on the essential features of the system and acceptable levels of quality. The Vision should include a description of what features will be included. It should also specify operational capacities (volumes, response times, accuracies), user profiles (who will be using the system), and inter-operational interfaces with entities outside the system boundary, where applicable. The Vision document provides the contractual basis for the requirements visible to the stakeholders.

7.2 Understand stakeholder needs

The purpose of this workflow detail is to elicit and collect information from the stakeholders of the project in order to understand what their needs are. The collected stakeholder requests can be regarded as a "wish list" that will be used as primary input to defining the high-level features of our system, as described in the Vision. This activity is mainly performed during iterations in the inception and elaboration phases.

The key activity of this workflow is to elicit stakeholder requests using such input as business rules, enhancement requests, interviews and requirements workshops. The primary outputs are collections of prioritized features and their critical attributes.

This information results in a refinement of the Vision document, as well as a better understanding of the requirements attributes. Another important output is an updated Glossary of terms to facilitate common vocabulary among team members.

Also, during this workflow detail discussion of the functional requirements of the system in terms of its use cases and actors is started. Those non-functional requirements, which do not fit easily into the use-case model, should be documented in the Supplementary Specifications.

7.2.1 Elicit stakeholder requests

In this step, the purpose is to elicit the stakeholder's needs relating to the project. Examples of stakeholders are investor, shareholder, buyer, designer, documentation writer and so on. The following techniques can be applied to collect and elicit relevant information from the stakeholders:

- Interviews.
- Requirements Workshop.
- Brain-storming and idea reduction.

- Use-Case Workshop.
- Storyboarding.
- Role playing.
- Review of existing requirements.

7.2.2 Find use cases

After finding actors and collecting stakeholder's needs we have to identify use cases defining the functionality of the system. Use cases are a technique for capturing the functional requirements of a system. A use case describes what happens in the system when an actor interacts with the system to execute the use case, it specifies the behaviour of a system or some subset of a system.

The collected use cases constitute all the possible ways of using the system. Each of use cases represents a specific flow of events. Then Flow of Events of a use case contains the most important information derived from use-case modelling work. It should describe the use case's flow of events clearly enough for an outsider to easily understand it. The flow of events should present what the system does, not how the system is designed to perform the required behaviour.

Use cases are graphically represented as an oval with the name of its functionality written below it (Figure 7.2.).

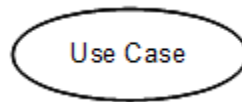


Figure 7.2. UML representation of an use case.

Following questions can be useful when identifying use cases:

- For each actor you have identified, what are the tasks in which the system would be involved?
- Does the actor need to be informed about certain occurrences in the system?
- Will the actor need to inform the system about sudden, external changes?
- Does the system supply the business with the correct behaviour?
- Can all features be performed by the use cases you have identified?
- What use cases will support and maintain the system?
- What information must be modified or created in the system?

Use cases that are often overlooked, since they do not represent what typically are the primary functions of the system, can be of the following kind:

- System start and stop.
- Maintenance of the system. For example, adding new users and setting up user profiles.
- Maintenance of data stored in the system. For example, the system is constructed to work in parallel with a legacy system, and data needs to be synchronized between the two.
- Functionality needed to modify behaviour in the system. An example would be functionality for creating new reports.

Similarly to actors the brief description of the use case should be given reflecting its purpose, the actors involved in the use case and the related glossary items.

Diagrams with actors, use cases, and relationships among them are called use-case diagrams and illustrate relationships in the use-case model (Figure 7.3.).

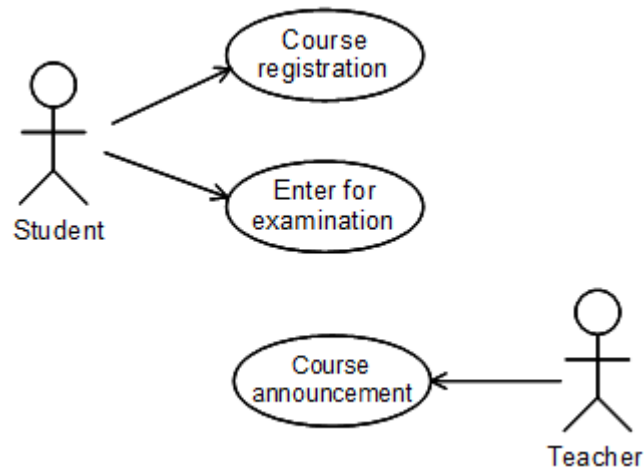


Figure 7.3. Use case diagram in UML.

A use-case model is a model of the system's intended functions and its surroundings, and serves as a contract between the customer and the developers. It describes a system's requirements in terms of use cases.

7.2.3 Manage dependencies

The introduction of notion of requirements types helps separate the different levels of abstraction and purposes of the requirements. Changes to requirements naturally impact the models produced.

Keys to effective requirements management include maintaining a clear statement of the requirements, along with applicable attributes for each requirement type and traceability to other requirements and other project artifacts. A Requirements Management Plan should be developed to specify the information and control mechanisms that will be collected and used for measuring, reporting, and controlling changes to the product requirements. Its purpose is to describe how the project will set up requirements documents and requirement types, and their respective requirement attributes and traceability. Using attributes requirements can be specified more:

- It helps the allocation of resources
- It helps the effective project monitoring
- We can define metrics
- We can manage risks
- It helps to estimate costs
- It helps to manage the Vision

Setting dependencies between requirements provides relations between them and we can keep track how the changes in requirements influence other requirements. Regular reviews, along with updates to the attributes and dependencies, should be done whenever the requirements specifications are updated.

7.3 Define the system

The purpose of this workflow detail is to:

- Align the project team in their understanding of the system.
- Perform a high-level analysis on the results of collecting stakeholder requests.
- Refine the Vision to include the features to include in the system, along with appropriate attributes.
- Refine the use-case model, to include outlined use cases.
- More formally document the results in models and documents.

Problem Analysis and activities for Understanding Stakeholder Needs create early iterations of key system definitions, including the features defined in the Vision document, a first outline to the use-case model and the Requirements Attributes. In Defining the System you will focus on identifying actors and use cases more completely, and expand the global non-functional requirements as defined in the Supplementary Specifications. Refining and structuring the use-case model has three main reasons for:

- To make the use cases easier to understand.
- To partition out common behaviour described within many use cases
- To make the use-case model easier to maintain.

An aspect of organizing the use-case model for easier understanding is to group the use cases into packages. A model structured into smaller units is easier to understand. It is easier to show relationships among the model's main parts if they are expressed in terms of packages. The use-case model can be organized as a hierarchy of use-case packages, with "leaves" that are actors or use cases.

A use-case package contains a number of actors, use cases, their relationships, and other packages; thus, there may be a multiple levels of use-case packages. Use-case packages can reflect order, configuration, or delivery units in the finished system. The packages can be organised according the principles:

- A package of use cases can be related to a subsystem.
- A package of use cases can be related to an actor.
- Packages can be formed by any logical grouping.

7.4 Manage the scope of the system

The objectives of this workflow detail are to:

- Prioritize and refine input to the selection of features and requirements that are to be included in the current iteration.
- Define the set of use cases (or scenarios) that represent some significant, central functionality.
- Define which requirement attributes and traceabilities to maintain.

The scope of a project is defined by the set of requirements allocated to it. Managing project scope, to fit the available resources such as time, people, and money is key to managing successful projects. Managing scope is a continuous activity that requires iterative or incremental development, which breaks project scope into smaller more manageable pieces.

Using requirement attributes, such as priority, effort, and risk, as the basis for negotiating the inclusion of a requirement is a particularly useful technique for managing scope.

Project scope should be managed continuously throughout the project. A better understanding of system functionality is obtained from identifying most actors and use cases. Non-functional requirements, which do not fit in the use-case model, should be documented in the Supplementary Specifications. The values of requirements attributes: priority, effort, cost, risk values etc., are determined from the appropriate stakeholders. These will be used in planning the iterations and identifying the architecturally significant use cases.

7.5 Refine the system definition

The purpose of this workflow detail is to further refine the requirements in order to:

- Describe the use case's flow of events in detail.
- Detail Supplementary Specifications.

- Develop a Software Requirements Specification, if more detail is needed, and
- Model and prototype the user interface.

The output of this workflow detail is a more in-depth understanding of system functionality expressed in detailed use cases, revised and detailed Supplementary Specifications, as well as user-interface elements. A formal Software Requirements Specification may be developed, if needed, in addition to the detailed use cases and Supplementary Specifications.

It is important to work closely with users and potential users of the system when modelling and prototyping the user-interface. This may be used to address usability of the system, to help uncover any previously undiscovered requirements and to further refine the requirements definition.

The Requirements Management Plan defines the attributes to be tracked for each type of requirement. The most important attributes are the benefit, the effort to implement, the risk to the development effort, the stability, and architectural impact of each requirement.

Regular reviews, along with updates to the attributes and dependencies, should be done, as shown in the Manage Changing Requirements workflow detail, whenever the requirements specifications are updated.

7.5.1 Details of use cases

The functionality of a system is defined by the use cases, each of which represents a specific flow of events. A use case describes what happens in the system when an actor interacts with the system to execute the use case. The use cases describe all the possible ways of using the system. The use case does not define how the system internally performs its tasks in terms of collaborating objects. This is left for the use-case realizations to show.

In an executing system, an instance of a use case corresponds to a specific flow of events that is invoked by an actor and executed as a sequence of events among a set of objects. We call this the realization of the use case. Often, the same objects participate in realizations of more than one use case.

The Flow of Events of a use case contains the most important information derived from use-case modelling work. It should describe the use case's flow of events clearly enough for an outsider to easily understand it. Guidelines for the contents of the flow of events are:

- Describe how the use case starts and ends
- Describe what data is exchanged between the actor and the use case
- Describe the flow of events, not only the functionality. To enforce this, start every action with "When the actor ... "
- Describe only the events that belong to the use case, and not what happens in other use cases or outside of the system
- Detail the flow of events, all "what" should be answered. Remember that test designers are to use this text to identify test cases.

The two main parts of the flow of events are basic flow of events and alternative flows of events. The basic flow of events should cover what "normally" happens when the use case is performed. The alternative flows of events cover behaviour of optional or exceptional character in relation to the normal behaviour, and also variations of the normal behaviour.

Both the basic flow of events and the alternative flows events should be further structured into steps or sub-flows. A sub-flow should be a segment of behaviour within the use case that has a clear purpose, and is "atomic" in the sense that either all or none of the actions is done.

7.5.2 Modelling and prototyping user-interface

The purpose of modelling user-interface is to build a model of the user interface that supports the reasoning about, and the enhancement of, its usability. For each use case prioritized to be considered from a usability perspective in the current iteration, we have to take the following steps:

- Describing the characteristics of actors related to use case
- Creating a use-case storyboard. A use-case storyboard is a logical and conceptual description of how a use case is provided by the user interface, including the interaction required between the actors and the system. The use-case storyboard has various properties, such as text fields and diagrams that describe the usability aspects of the use case.
- Describing the flow of events - storyboard. The step-by-step description of each use case that is input to this activity need to be refined and focused on usability issues; this refinement is captured in the Flow of Events - Storyboard property of the use-case storyboard.
- Capturing usability requirements on the use-case storyboard
- Identification of boundary classes needed by the use-case storyboard. In this step we identify the boundary classes needed to provide the user interface of the use case.
- Describe Interactions Between Boundary Objects and Actors. Illustration of the interactions between the participating boundary objects by creating one or more collaboration diagrams. The collaboration diagram should also show interactions between the system and its actors. The flow of events usually begins when one of the actors requests something from the system, since an actor always invokes the use case.

For the identified boundary classes, we have to describe:

- Responsibilities of boundary classes
- Attributes of boundary classes
- Relationships between boundary classes
- Usability requirements on boundary classes

7.6 Manage changing requirements

The purpose of this workflow detail is to:

- Evaluate formally submitted change requests and determine their impact on the existing requirement set.
- Structure the use-case model.
- Set up appropriate requirements attributes and traceabilities.
- Formally verify that the results of the Requirements discipline conform to the customer's view of the system.

Changes to requirements naturally impact the developed models. Identifying the relationships between requirements and other artifacts are the key to understanding the impact of requirements change.

Regular reviews, along with updates to the attributes and dependencies, should be done as shown in this workflow detail whenever the requirements specifications are updated.

7.6.1 Structure the use case model

There are three main reasons for structuring the use-case model:

- To make the use cases easier to understand.

- To partition out common behaviour described within many use cases
- To make the use-case model easier to maintain.

In order to structure the use cases models, we have three kinds of relationships: the extend and include relationships between use cases and generalization.

Extend relationship between use cases

If there is a part of a base use case that is optional, or not necessary to understand the primary purpose of the use case, it can be factor out to an addition use case in order to simplify the structure of the base use case. The addition is implicitly inserted in the base use case, using the extend-relationship

An extend-relationship is a relationship from an extension use case to a base use case, specifying how the behaviour defined for the extension use case can be inserted into the behaviour defined for the base use case. It is implicitly inserted in the sense that the extension is not shown in the base use case. The extensions can be used for several purposes:

- To show that a part of a use case is optional, or potentially optional, system behaviour.
- To show that a sub-flow is executed only under certain conditions, such as triggering an alarm.
- To show that there may be a set of behaviour segments of which one or several may be inserted at an extension point in a base use case. The behaviour segments that are inserted will depend on the interaction with the actors during the execution of the base use case.

The extension is conditional, which means its execution is dependent on what has happened while executing the base use case. The base use case does not control the conditions for the execution of the extension, the conditions are described within the extend relationship. The extension use case may access and modify attributes of the base use case. The base use case, however, cannot see the extensions and may not access their attributes.

The base use case is implicitly modified by the extensions. We can say that the base use case defines a modular framework into which extensions can be added, but the base does not have any visibility of the specific extensions.

The base use case should be complete in and of itself, meaning that it should be understandable and meaningful without any references to the extensions. However, the base use case is not independent of the extensions, since it cannot be executed without the possibility of following the extensions.

Figure 7.4. shows an example for application of extend relationship.

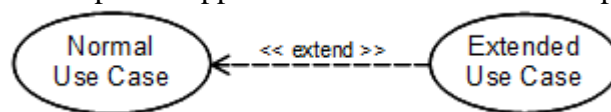


Figure 7.4. Representation of extend relationship.

Include relationship between use cases

If there is a part of a base use case that represents a function of which the use case only depends on the result, not the method used to produce the result it can be factored out to an addition use case. The addition is explicitly inserted in the base use case, using the include-relationship.

An include-relationship is a relationship from a base use case to an inclusion use case, specifying how the behaviour defined for the inclusion use case is explicitly inserted into the behaviour defined for the base use case.

The include-relationship connects a base use case to an inclusion use case. The inclusion use case is always abstract. It describes a behaviour segment that is inserted into a use-case

instance that is executing the base use case. The base use case has control of the relationship to the inclusion and can depend on the result of performing the inclusion, but neither the base nor the inclusion may access each other's attributes. The inclusion is in this sense encapsulated, and represents behaviour that can be reused in different base use cases.

Reasons for using the include-relationship are as follows:

- Factor out behaviour from the base use case that is not necessary for the understanding of the primary purpose of the use case, only the result of it is important.
- Factor out behaviour that is in common for two or more use cases.

The include relationship is shown in Figure 7.5.

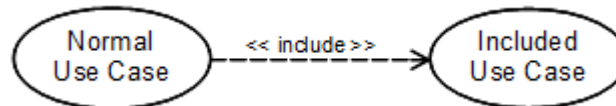


Figure 7.5. representation of include relationship.

Generalization

If there are use cases that have commonalities in behaviour and structure and similarities in purpose, their common parts can be factored out to a base use case (parent) that is inherited by addition use cases (children). The child use cases can insert new behaviour and modify existing behaviour in the structure they inherit from the parent use case. A generalization is a taxonomic relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element, and contains additional information.

Many things in real life have common properties. Objects can have common properties as well, which can be clarified using a generalization between their classes. By extracting common properties into classes of their own, it is possible to change and maintain the system more easily in the future.

A generalization shows that one class inherits from another. The inheriting class is called a descendant. The class inherited from is called the ancestor. Inheritance means that the definition of the ancestor including any properties such as attributes, relationships, or operations on its objects is also valid for objects of the descendant. The generalization is drawn from the descendant class to its ancestor class.

A use-case-generalization is a relationship from a child use case to a parent use case, specifying how a child can specialise all behaviour and characteristics described for the parent.

A parent use case may be specialized into one or more child use cases that represent more specific forms of the parent. A child inherits all structure, behaviour, and relationships of the parent. Children of the same parent are all specializations of the parent. This is generalization as applicable to use cases.

Generalization is used when two or more use cases have commonalities in behaviour, structure, and purpose. When this happens, the shared parts can be described in a new, often abstract, use case that is then specialized by child use cases. The representation of use case generalization is shown in Figure 7.6.

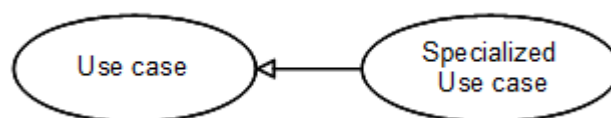


Figure 7.6. Use case generalization.

An actor-generalization from an actor type (descendant) to another actor type (ancestor) indicates that the descendant inherits the role the ancestor can play in a use case.

A user can play several roles in relation to the system, which means that the user may, in fact, correspond to several actors. To make the model clearer, the user can be represented by one actor who inherits several actors. Each inherited actor represents one of the user's roles relative to the system.

Several actors can play the same role in a particular use case. The shared role is modelled as an actor inherited by the two original actors. This relationship is shown with actor-generalizations. The representation of use case generalization is shown in Figure 7.7.

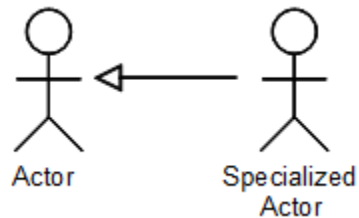


Figure 7.7. Use case generalization.

7.6.2 Review requirements

The purpose is to formally verify that the results of Requirements conform to the customer's view of the system. The following guidelines are helpful when reviewing the results of Requirements:

- Always conduct reviews in a meeting format, although the meeting participants might prepare some reviews on their own.
- Continuously check what is produced to make sure the product quality is as high as possible. Checkpoints are provided for this purpose; refer to the checkpoints for each analysis activity. They can be used for informal review meetings or in daily work.
- Normally, the review should be divided into the following meetings:
 - a) A review of change requests which impact the existing requirements set.
 - b) A review of the entire use-case model.
 - c) A review of the use cases, along with their diagrams. If the system is large, break this review into several meetings, possibly one per Use-Case Package or Software Requirements Specification.

7.7 Exercises

1. What are the requirements?
2. What are the workflow details of Requirement discipline?
3. What are the types of requirements?
4. List some non-functional requirements!
5. List some stakeholders in a software development project!
6. Why requirement management is so important?
7. How use cases can be structured?
8. What is a scenario?
9. Explain the main points of workflow detail Manage the Scope of the System!
10. How generalization and specialisation can be realised?

8 Analysis and design

The overall purpose of the Analysis and design workflow is to translate the requirements into a specification of how to implement the system. The activity spans a range of abstraction, looking at fairly abstract architectural issues in the early iterations but becoming increasingly detailed with later iterations. The main objectives in this workflow are the followings [3,10,11,12,17,22]:

- To transform the requirements into a specification of how to implement the system.
- To evolve a robust architecture for the system. This means that system can be easily changed when its functional requirements change.
- To adapt the design to match the implementation environment, designing it for performance.

Analysis and design results in a design model and optionally in an analysis model. The design model serves as an abstraction of the source code; that is, the design model acts as a 'blueprint' of how the source code is structured and written. The design model consists of design classes structured into design packages and design subsystems with well-defined interfaces, representing what will become components in the implementation. It also contains descriptions of how objects of these design classes collaborate to perform use cases.

Design activities are centred around the notion of architecture. The production and validation of this architecture is the main focus of early design iterations. Architecture is represented by a number of architectural views. These views capture the major structural design decisions. In essence, architectural views are abstractions or simplifications of the entire design, in which important characteristics are made more visible by leaving details aside.

Analysis is mainly involved with transforming the requirements into an architecture and collection of components that could fully support an implementation of the proposed system. The focus is mainly on functional requirements and the creation of a number of design components. Analysis creates an idealised view of the design of the system that will be likely modified during detailed design.

The analysis model is usually an abstraction of the design model it is the primary artifact of the workflow detail called Analyze Behaviour. It is a platform independent model, which means that it does not contain technology-based decisions. It omits much of the detail of the design model.

The major content of the analysis model includes UML collaborations, which group class and sequence diagrams. The collaborations are traced back to the use cases from which they are realized, using an UML realization relationship. The analysis model also contains analysis classes, which are organized according to the logical architectural pattern.

Design begins where analysis leaves off. Taking the idealised design as the starting point the design process attempts to create the framework for an implementation that is able to meet all the non-functional requirements that are largely ignored in the analysis phase. Design should be detailed enough that it determines the structure of the implementation sufficiently that we can be sure any implementation meeting the design will satisfy the requirements. A consequence of this is that the level at which design stops will vary depending on the experience of the implementation team, the nature of the development environment, and the precision to which the requirements are specified.

The design model is the primary artifact of the analysis and design workflow. It comprises class definitions and how they collaborate to provide requirements specified by use cases.

These may be further aggregated into packages and subsystems that group related classes to allow us to hide unnecessary detail where necessary.

Design ensures that non-functional requirements are met. The design model has a mixture of behaviour and technology, and is considered a platform-specific model.

In the frame of RUP methodology the activities and artifacts are organized into workflow details as follows:

- Define a candidate architecture.
- Refine the architecture.
- Analyze behaviour.
- Design components.

8.1 Define a candidate architecture

In this workflow detail some candidate architectures are proposed and tested against the relevant use cases. The purpose of this workflow detail is to:

- Create the candidate architecture of the system
 - a. Define an initial set of architecturally significant elements to be used as the basis for analysis.
 - b. Define an initial set of analysis mechanisms.
 - c. Define the initial layering and organization of the system.
 - d. Define the use-case realizations to be addressed in the current iteration.
- Identify analysis classes from the architecturally significant use cases.
- Update the use-case realizations with analysis class interactions.

We perform an initial pass at the architecture, then we choose architecturally significant use cases, performing an Use-Case Analysis on each one. After each use case is analyzed, we update the architecture as needed to reflect adaptations required to accommodate new behaviour of the system and to address potential architectural problems which are identified.

8.1.1 Architectural analysis

In this activity the purpose is:

- To define a candidate architecture for the system, based on experience gained from similar systems or in similar problem domains.
- To define the architectural patterns, key mechanisms and modelling conventions for the system.
- To define the reuse strategy.
- To provide input to the planning process.

Where the architecture already exists, either from a prior project or iteration, change requests may need to be created to change the architecture to account for the new behaviour the system must support. These changes may be to any artifact in the process, depending on the scope of the change.

Analysis pattern

An analysis mechanism represents a pattern that constitutes a common solution to a common problem. They may show patterns of structure, patterns of behaviour, or both. They are used during analysis to reduce the complexity of analysis, and to improve its consistency by providing designers with a short-hand representation for complex behaviour. Mechanisms allow the analysis effort to focus on translating the functional requirements into software.

Analysis mechanisms often result from the instantiation of one or more architectural or analysis patterns. They provide specific behaviours to a domain-related class or component, or correspond to the implementation of cooperation between classes and/or components [8].

Reuse strategy

Developing reuse strategy the reusable elements have to be identified and the possibilities for reuse have to be investigated.

Layers

The design model is normally organized in layers that are a common architectural pattern for moderate to large sized systems.

Subsystems should be organized into layers with application-specific sub-systems located in the upper layers of the architecture, hardware and operating-specific subsystems located in the lower layers of the architecture, and general-purpose services occupying the middleware layers.

Layering represents an ordered grouping of functionality, with the application-specific located in the upper layers, functionality that spans application domains in the middle layers, and functionality specific to the deployment environment at the lower layers.

The number and composition of layers is dependent upon the complexity of both the problem domain and the solution space.

The following sample architecture has four layers:

- The top layer, *application layer*, contains the application specific services.
- The next layer, *business-specific layer*, contains business specific components, used in several applications.
- The *middleware layer* contains components such as GUI-builders, interfaces to database management systems, platform-independent operating system services, and OLE-components such as spreadsheets and diagram editors.
- The bottom layer, *system software layer*, contains components such as operating systems, databases, interfaces to specific hardware and so on.

Layering provides a logical partitioning of sub-systems into a number of sets, with certain rules as to how relationships can be formed between layers. The layering provides a way to restrict inter-subsystem dependencies, with the result that the system is more loosely coupled and therefore more easily maintained.

Subsystems and packages within a particular layer should only depend upon subsystems within the same layer, and at the next lower layer.

Packages

UML uses packages to structure models into smaller elements. A model structured into smaller units is easier to understand. It is easier to show relationships among the model's main parts if they are expressed in terms of packages. A package is either the top-level package of the model, or stereotyped as an use-case package.

By grouping model elements into packages and subsystems, then showing how those groupings relate to one another, it is easier to understand the overall structure of the model. A package should be identified for each group of classes that are functionally related. Packages can be specialized and stereotyped (Figure 8.1.).

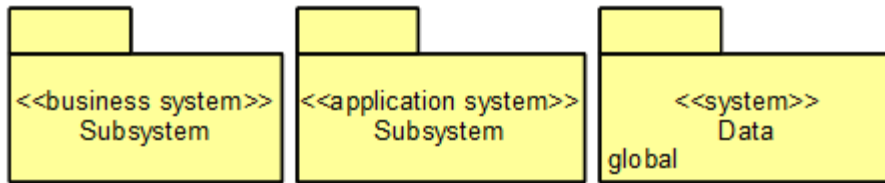


Figure 8.1. Representation of UML packages.

If a class in one package has an association to a class in a different package, then these packages depend on each other. Package dependencies are modelled using a dependency relationship between the packages. Figure 8.2. shows dependency between two packages.



Figure 8.2. Representing dependency of packages.

Use cases describe interacts between actors and systems, they specify the functional behaviour of a system or subsystem.

A use-case realization describes how a particular use case is realized within the model, in terms of collaborating objects. For each use-case realization there may be one or more class diagrams depicting its participating classes. A class and its objects often participate in several use-case realizations. It is important during design to coordinate all the requirements on a class and its objects that different use-case realizations may have.

A use-case realization represents the design perspective of a use case. It is an organization model element used to group a number of artifacts related to the design of a use case, such as class diagrams of participating classes and subsystems, and sequence diagrams which illustrate the flow of events of a use case performed by a set of class and subsystem instances.

For each use case in the use-case model, there is a use-case realization in the design model with a realization relationship to the use case. In the UML this is shown as a dashed arrow, with an arrowhead like a generalization relationship, indicating that a realization is a kind of inheritance, as well as a dependency. Figure 8.3. shows a realization relationship.

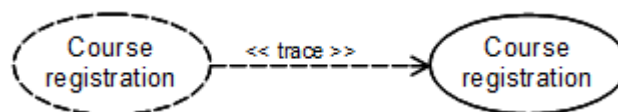


Figure 8.3. Representation of use-case realization in UML.

For each use-case realization there is one or more interaction diagrams depicting its participating objects and their interactions. There are two types of interaction diagrams: sequence diagrams and collaboration diagrams. They express similar information, but show it in different ways. Sequence diagrams show the explicit sequence of messages and are better when it is important to visualize the time ordering of messages, whereas collaboration diagrams show the communication links between objects and are better for understanding all of the effects on a given object.

8.1.2 Use case analysis

After performing Architectural Analysis we choose architecturally significant use cases, performing Use-Case Analysis on each one. Purpose of use case analysis is:

- To identify the classes which perform a use case's flow of events.
- To distribute the use case behaviour to those classes, using use-case realizations.
- To identify the responsibilities, attributes and associations of the classes.
- To note the usage of architectural mechanisms

8.1.2.1 Find analysis classes

The purpose of this activity is to identify a candidate set of analysis classes which will be capable of performing the behaviour described in use cases.

Finding a candidate set of analysis classes is the first step in the transformation of the system from a statement of required behaviour to a description of how the system will work. In this effort, analysis classes are used to represent the roles of model elements which provide the necessary behaviour to fulfil the functional requirements specified by use cases and the non-functional requirements specified by the supplemental requirements. As the project focus shifts to design, these roles evolve a set of design elements which realize the use cases.

The roles identified in Use-Case Analysis primarily express behaviour of the upper-most layers of the system i.e. application-specific behaviour and domain specific behaviour. Boundary classes and control classes typically evolve into application-layer design elements, while entity classes evolve into domain-specific design elements. Lower layer design elements typically evolve from the analysis mechanisms which are used by the analysis classes identified here.

Three different perspectives of the system are used in the identification of candidate classes. The three perspectives are that of the boundary between the system and its actors, the information used by system, and the control logic of the system. The corresponding class stereotypes are boundary, entity and control.

Identification of classes means just that they should be identified, named, and described briefly in a few sentences.

In UML models, a stereotype is a model element that is an extensibility mechanism, which can be used to identify the purpose of the model element to which we apply it. A stereotype can be used to refine the meaning of a model element. Figure 8.4. shows an example of stereotyping a class as interface.

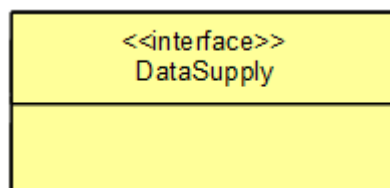


Figure 8.4. Representation of stereotyping.

Analysis classes may be stereotyped as one of the following:

- Stereotype <<boundary>> - Boundary classes.
- Stereotype <<control>> - Control classes
- Stereotype <<entity>> - Entity classes.

This stereotyping results in a robust object model because changes to the model tend to affect only a specific area.

A *boundary class* is a class used to model interaction between the system's surroundings and its inner workings. They are modelled according to what kind of boundary they represent. Communication with another system and communication with a human actor (through a user interface) have very different objectives.

A system may have several types of boundary classes:

- *User interface classes.* Classes which intermediate communication with human users of the system.
- *System interface classes.* Classes which intermediate communication with other system.
- *Device interface classes.* Classes which provide the interface to devices, which detect external events.

There is at least one boundary object for each use-case actor-pair. This object can be viewed as having responsibility for coordinating the interaction with the actor. Sketches, use screen shots from a user-interface prototype can illustrate the behaviour and appearance of the boundary objects.

For analysis goals only key abstractions of the system necessary to model. Examples of boundary classes are shown in Figure 8.5.

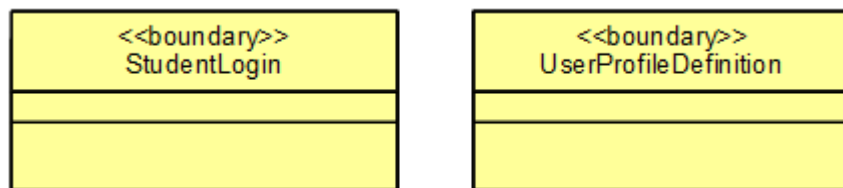


Figure 8.5. Boundary classes in UML.

An *entity class* is a class used to model information and associated behaviour that must be stored. Entity objects represent the key concepts of the system being developed, such as an event, a person, or some real-life object. They are usually persistent, having attributes and relationships needed for a long period, sometimes for the life of the system. A frequent source of inspiration for entity classes are the Glossary developed during requirements. Figure 8.6. shows examples for entity classes.

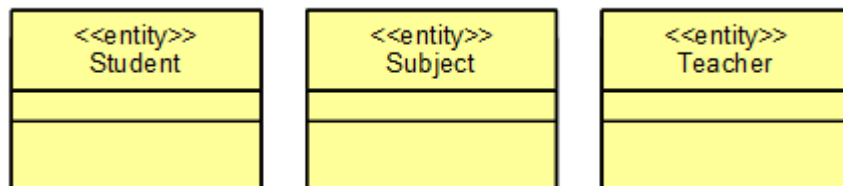


Figure 8.6. Representation of entity classes in UML.

A *control class* is a class used to model control behaviour specific to one or a few use cases. Control objects often control other objects, so their behaviour is of the coordinating type. Control classes encapsulate use-case specific behaviour.

The system can perform some use cases without control objects (just using entity and boundary objects). They are particularly use cases that involve only the simple manipulation of stored information.

More complex use cases generally require one or more control classes to coordinate the behaviour of other objects in the system. Examples of control objects include programs such as transaction managers, resource coordinators, and error handlers.

Control classes effectively de-couple boundary and entity objects from one another, making the system more tolerant of changes in the system boundary. Control classes are represented as shown in Figure 8.7.

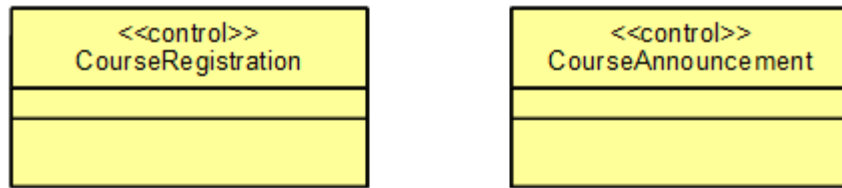


Figure 8.7. Representation of control classes in UML.

8.1.2.2 Distribute behaviour to analysis classes

The purpose of this activity is to express the use-case behaviour in terms of collaborating analysis classes and to determine the responsibilities of analysis classes.

For each use-case realization there is one or more interaction diagrams depicting its participating objects and their interactions. There are two types of interaction diagrams: sequence diagram and collaboration diagram. They express similar information, but show it in different ways. Sequence diagrams show the explicit sequence of messages and are better when it is important to visualize the time ordering of messages, whereas collaboration diagrams show the communication links between objects and are better for understanding all of the effects on a given object.

Sequence diagrams

We use a sequence diagram to illustrate use-case realizations and show how objects interact to perform the behaviour of all or part of a use case. A sequence diagram describes a pattern of interaction among objects, arranged in a chronological order; it shows the objects participating in the interaction by their "lifelines" and the messages that they send to each other.

Sequence diagrams are particularly important to designers because they clarify the roles of objects in a flow and thus provide basic input for determining class responsibilities and interfaces.

We can have objects and actor instances in sequence diagrams, together with messages describing how they interact. The diagram describes what takes place in the participating objects, in terms of activations, and how the objects communicate by sending messages to one another.

An object is shown as a vertical dashed line called the "lifeline". The lifeline represents the existence of the object at a particular time. In sequence diagrams, a message is shown as a horizontal solid arrow from the lifeline of one object to the lifeline of another object.

For an example, the sequence diagram shown in Figure 8.8. illustrates the use case realization of a course registration. In this use case student successfully registers a course coded by UML-01. Students can communicate the system through two user interfaces realized by boundary classes Login and CourseRegistration. The use case realization is controlled by the control class named Control. There are four entity classes: Student, Subject, Course, and Registration.

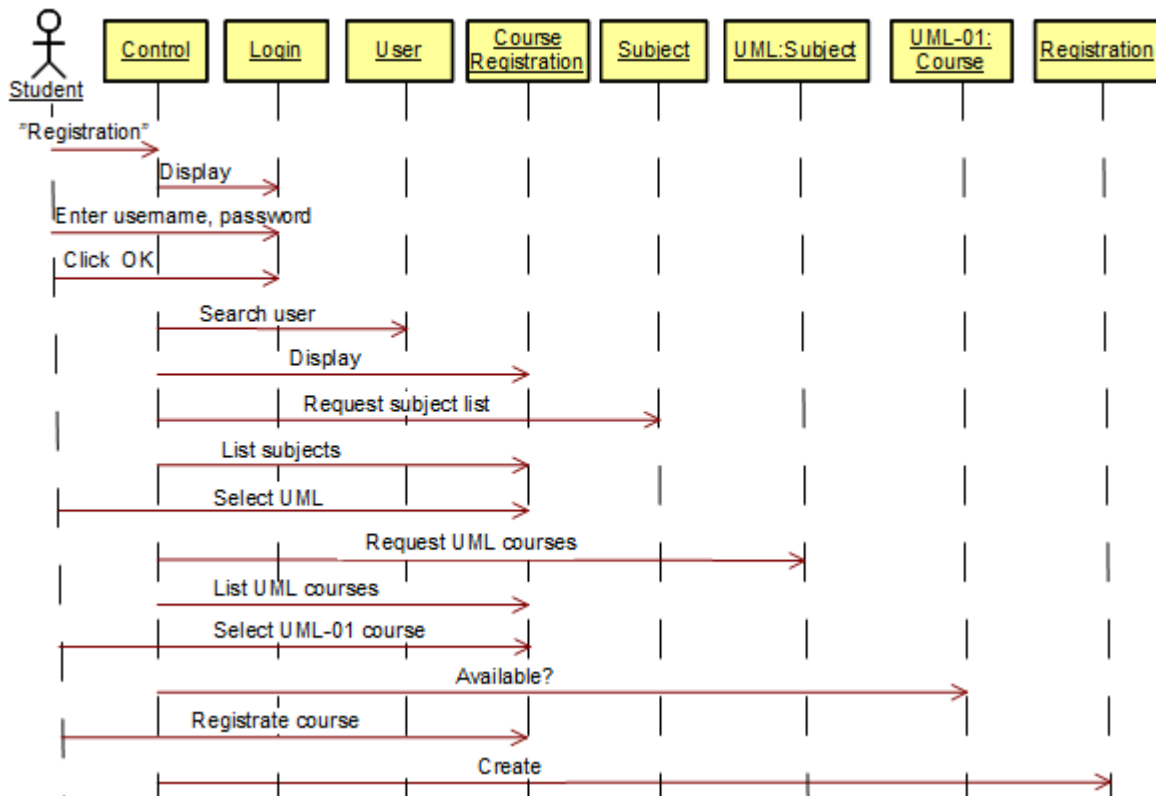


Figure 8.8. The sequence diagram of use case UML-01 course registration.

collaboration diagrams

Collaboration diagrams are used to show how objects interact to perform the behaviour of a particular use case, or a part of a use case.

A collaboration diagram describes a pattern of interaction among objects; it shows the objects participating in the interaction by their links to each other and the messages that they send to each other.

Along with sequence diagrams, collaborations are used by designers to define and clarify the roles of the objects that perform a particular flow of events of a use case. They are the primary source of information used to determining class responsibilities and interfaces.

We can have objects and actor instances in collaboration diagrams, together with links and messages describing how they are related and how they interact. The diagram describes what takes place in the participating objects, in terms of how the objects communicate by sending messages to one another. A link is a relationship among objects across which messages can be sent. In collaboration diagrams, a link is shown as a solid line between two objects. In collaboration diagrams, a message is shown as a labelled arrow placed near a link. This means that the link is used to transport, or otherwise implement the delivery of the message to the target object. The arrow points along the link in the direction of the target object (the one that receives the message). The arrow is labelled with the name of the message, and its parameters. The arrow may also be labelled with a sequence number to show the sequence of the message in the overall interaction.

For an example, the use case realization illustrated in Figure 8.8. is shown in a collaboration diagram in Figure 8.9.

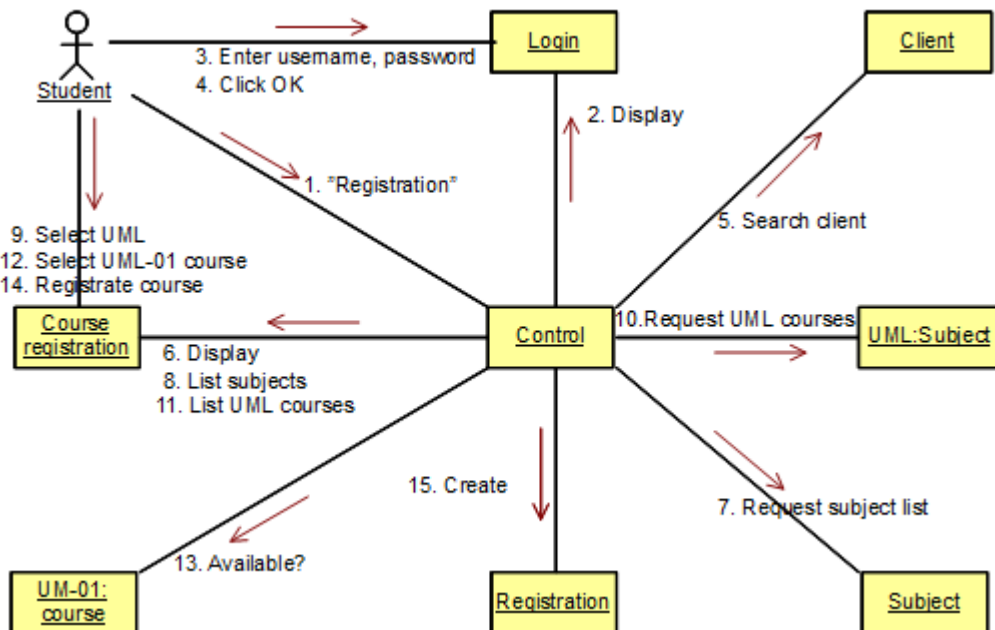


Figure 8.9. Collaboration diagram of use case UML-01 course registration.

In order to distribute behaviour to analysis classes we follow the next steps for each independent scenario:

- *Create one or more collaboration diagrams.* At least one diagram is usually needed for the main flow of events of the use case, plus at least one diagram for each alternate/exceptional flow.
- *Identify the analysis classes* responsible for the required behaviour by stepping through the flow of events of the scenario, ensuring that all behaviour required by the use case is provided by the use-case realization.
- *Illustrate interactions between analysis classes in the collaboration diagram.* The collaboration diagram also shows interactions of the system with its actors.

8.1.2.3 Describe responsibilities

The purpose of this activity is to describe the responsibilities of a class of objects identified from use-case behaviour. A responsibility is a statement of something an object can be asked to provide. Responsibilities evolve into operations on classes in design; they can be characterized as:

- The actions that the object can perform.
- The knowledge that the object maintains and provides to other objects.

Each analysis class should have several responsibilities; a class with only one responsibility is probably too simple, while one with a dozen or more is pushing the limit of reasonability and should potentially be split into several classes.

Responsibilities are derived from messages in collaboration diagrams. For each message, the class of the object is examined to which the message is sent. If the responsibility does not yet exist, we create a new responsibility that provides the requested behaviour.

Responsibilities are documented with a short name for the responsibility, and a short description. The description states what the object does to fulfil the responsibility, and what result is returned when the responsibility is invoked.

Figure 8.10. shows the class Subject with operations identified using diagrams in Figure 8.8. and 8.9.

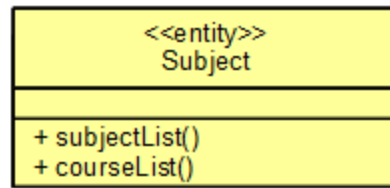


Figure 8.10. Class Subject with identified methods.

The collaboration diagram modified by identified methods shown in Figure 8.11.

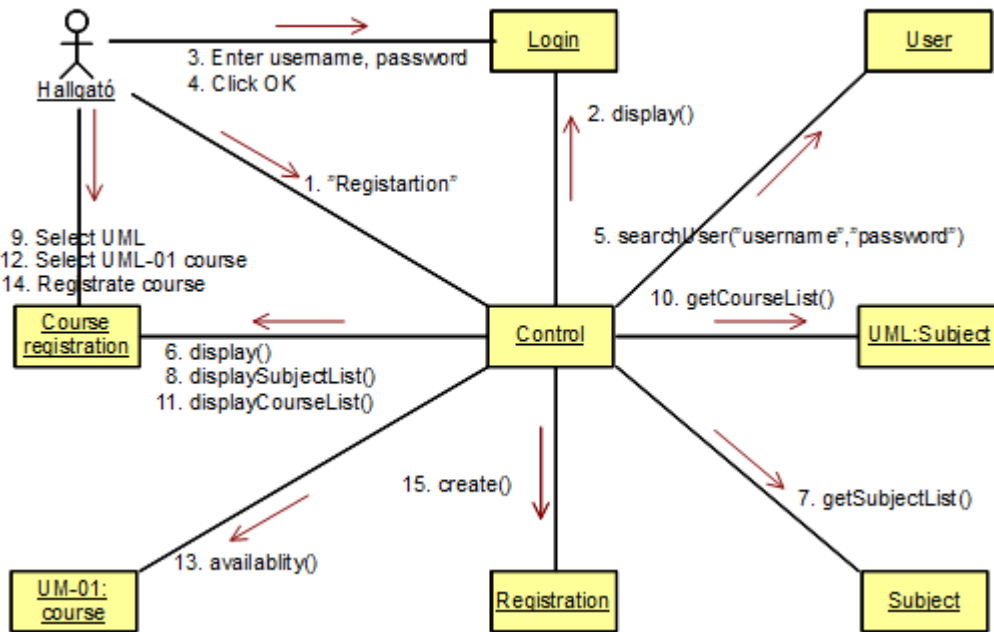


Figure 8.11. The modified collaboration diagram of use case UML-01 course registration.

Attributes

Attributes are used to store information by a class. Specifically, attributes are used where the information is:

- Referred to "by value"; that is, it is only the value of the information, not its location or object identifier which is important.
- Uniquely "owned" by the object to which it belongs; no other objects refer to the information.
- Accessed by operations which only get, set or perform simple transformations on the information; the information has no "real" behaviour other than providing its value.

If, on the other hand, the information has complex behaviour, is shared by two or more objects, or is passed "by reference" between two or more objects, the information should be modelled as a separate class.

The attribute name should be a noun that clearly states what information the attribute holds. The description of the attribute should describe what information is to be stored in the attribute; this can be optional when the information stored is obvious from the attribute name. The attribute type is the simple data type of the attribute. Examples include string, integer, number. In Figure 8.12. the attributes of class Subject are shown.

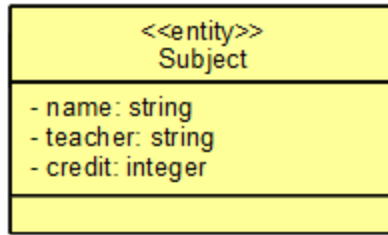


Figure 8.12. Attributes of class Subject.

Figure 8.13. shows all the classes realize use case UML-01 course registration.

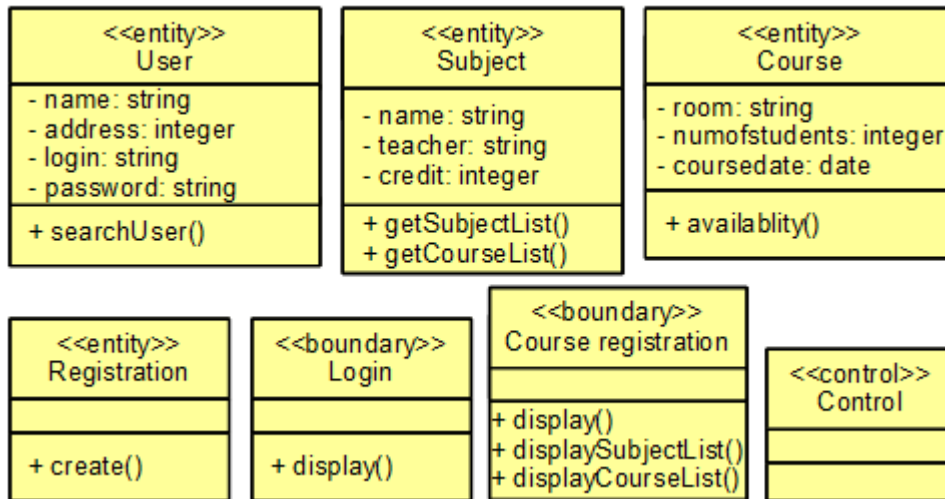


Figure 8.13. Classes realizing use case UML-01 course registration.

Association

In order to carry-out their responsibilities, classes frequently depend on other classes to supply needed behaviour. Associations document the inter-class relationships and help us to understand class coupling.

In the collaboration diagrams links between classes indicate that objects of the two classes need to communicate with one another to perform the Use Case. Once we start designing the system, these links may be realized in several ways:

- The object may have "global" scope, in which case any object in the system can send messages to it
- One object may be passed the second object as a parameter, after which it can send messages to the passed object
- The object may have a permanent association to the object to which messages are sent.

Associations represent structural relationships between objects of different classes; they represent connections between instances of two or more classes that exist for some duration. It can be used to show that objects know about another objects. Sometimes, objects must hold references to each other to be able to interact, for example send messages to each other; thus, in some cases associations may follow from interaction patterns in sequence diagrams or collaboration diagrams.

Most associations are binary, and are drawn as solid paths connecting pairs of class symbols (Figure 8.14.). An association may have either a name or the association roles.

Each end of an association is a *role* specifying the face that a class plays in the association. The role name should be a noun indicating the associated object's role in relation to the

associating object. In the example shown in Figure 8.14. the role employee is assigned to class Firm and role employer is assigned to class Person.

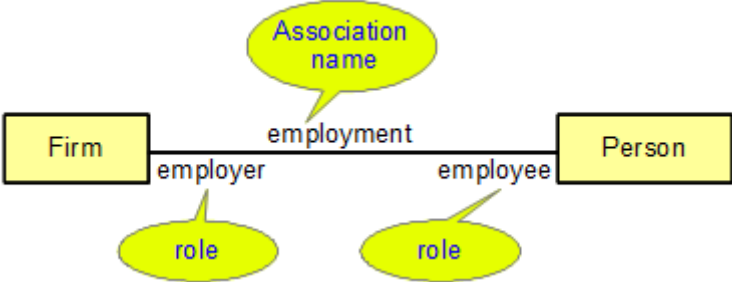


Figure 8.14. Association relationship between classes.

Multiplicity

For each role we can specify the *multiplicity* of its class, how many objects of the class can be associated with one object of the other class. Multiplicity is indicated by a text expression on the role. The expression is a comma-separated list of integer ranges. A range is indicated by an integer for the lower value, two dots, and an integer for the upper value.

Some examples are given in the followings:

- 1 : exactly one object is associated.
- * : any number of object including none.
- 0..1 : 0 or 1 object is associated.
- 1..* : 1 or more object is associated.
- 22..44 : the number of object associated is from the range [22;44].
- 9 : exactly 9 object is associated.

The *navigability* property on a role indicates that it is possible to navigate from an associating class to the target class using the association. Navigability is indicated by an open arrow, which is placed on the target end of the association line next to the target class.

The Figure 8.15. shows the association between class Firm and Person when specifying multiplicity. In the example we represent an association in which every person has an employee and every firm may have number of employer between 1 and 10.

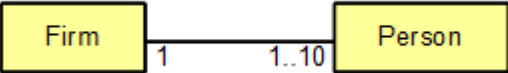


Fig 8.15. Representation of multiplicity.

An *association class* is an association that also has class properties (such as attributes, operations, and associations). It is shown by drawing a dashed line from the association path to a class symbol that holds the attributes, operations, and associations for the association. The attributes, operations, and associations apply to the original association itself. Each link in the association has the indicated properties. Fig 8.16. shows the application of an association class. In this case associations can be regarded as instances of class Employment. Using attribute payment we can order different payment to employers.

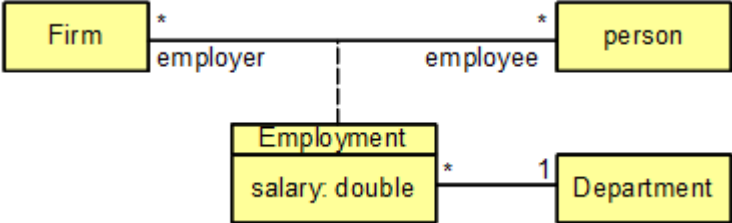


Figure 8.16. Representation of association class.

Aggregation and composition

An *aggregation* is a special form of association that models a whole-part relationship between an aggregate (the whole) and its parts. Aggregation is used to model a compositional relationship between model elements. In UML diagrams a hollow diamond is attached to the end of an association path on the side of the aggregate (the whole) to indicate aggregation.

Composition is a form of aggregation with strong ownership and coincident lifetime of the part with the aggregate. The multiplicity of the aggregate end may not exceed one. Composition represented by a solid filled diamond attached to the end of an association path. Figure 8.17. shows an example of aggregation and composition. A polygon may have a number of vertices and characteristics such as color and surface area. Using the aggregation we define Polygon have vertices at most 5. The composition is used to express that lifetime of characteristics depends on lifetime of polygon.

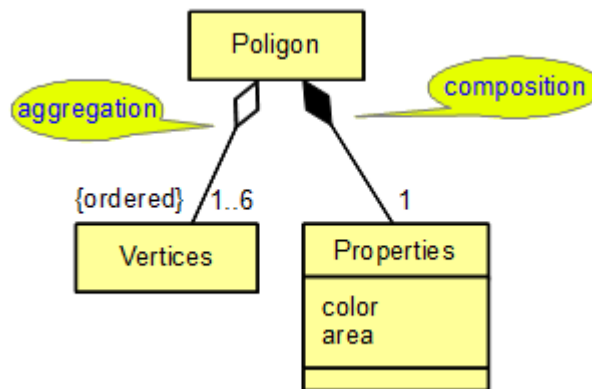


Figure 8.17. Representation of aggregation and composition relationship.

Figure 8.18. show the class diagram of classes and their associations realizing the use case UML-01 course registration.

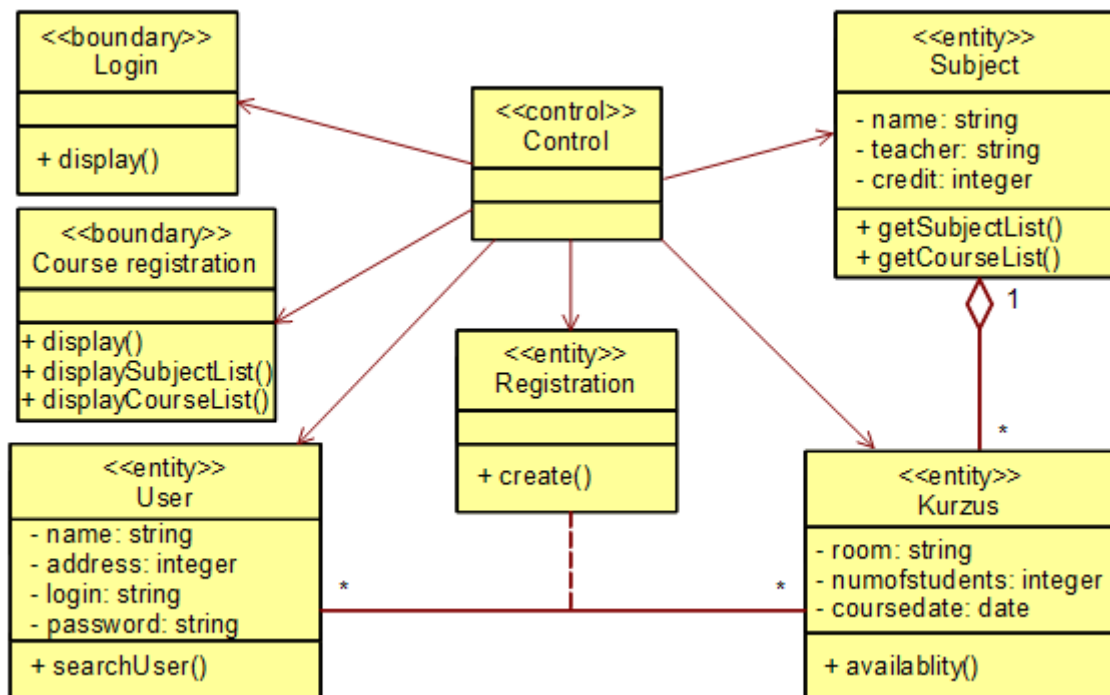


Figure 8.18. Identified classes and associations of use case UML-01 course registration.

Enforcing consistency

- When a new behaviour is identified, it can be checked to see if there is an existing class that has similar responsibilities, reusing classes where possible. The new class should be created only when sure that there is not an existing object that can perform the behaviour.
- As classes are identified, we have to examine them to ensure they have consistent responsibilities. When a classes responsibilities are disjoint, we have to split the object into two or more classes. The collaboration diagrams are updated accordingly.
- If a class is split because disjoint responsibilities are discovered, we have to examine the collaborations in which the class plays a role to see if the collaboration needs to be updated. The collaboration is updated if needed.
- A class with only one responsibility is not a problem, per se, but it should raise questions on why it is needed.

8.2 Refine the architecture

The purpose of this workflow detail is to:

- Provide the natural transition from analysis activities to design activities, identifying:
 - a) appropriate design elements from analysis elements.
 - b) appropriate design mechanisms from related analysis mechanisms.
- Maintain the consistency and integrity of the architecture, ensuring that:
 - a) new design elements identified for the current iteration are integrated with pre-existing design elements.
 - b) maximal re-use of available components and design elements is achieved as early as possible in the design effort.
- Describe the organization of the system's run-time and deployment architecture
- Organize the implementation model so as to make the transition between design and implementation seamless

8.2.1 Identify design mechanisms

The purpose of this activity is to refine the analysis mechanisms into design mechanisms based on the constraints imposed by the implementation environment.

Design mechanism is an architectural mechanism used during the design process. It is a refinement of a corresponding analysis mechanism and it may bind one or more architectural and design patterns. A mechanism at the analysis level and the design level means the same things, but at a different level of refinement. A design mechanism assumes some details of the implementation environment.

A design pattern provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context. The steps for refining the information gathered on the analysis mechanisms are as follows:

- *Identification of the clients of each analysis mechanism.* We scan all clients of a given analysis mechanism, looking at the characteristics they require for that mechanism.
- *Identification of characteristic profiles for each analysis mechanism.* There may be widely varying characteristics profiles, providing varying degrees of performance, footprint, security, economic cost, etc. Each analysis mechanism is different, different characteristics will be applied to each.

- *Grouping clients according to their use of characteristic profiles.* We form groups of clients that need for an analysis mechanism with a similar characteristics profile and we identify a design mechanism based on each such characteristics profile. These groupings provide an initial cut at the design mechanisms. Different characteristic profiles will lead to different design mechanisms which emerge from the same analysis mechanism.

8.2.2 Identify design elements

Purpose of this activity is to analyze interactions of analysis classes to identify design model elements.

The use case analysis results in analysis classes, which represent conceptual things which can perform behaviour. In design, analysis classes evolve into a number of different kinds of design elements:

- classes, to represent a set of rather fine-grained responsibilities;
- subsystems, to represent a set of coarse-grained responsibilities, perhaps composed of a further set of subsystems, but ultimately a set of classes;
- active classes, to represent threads of control in the system;
- interfaces, to represent abstract declarations of responsibilities provided by a class or subsystem.

When the analysis class is simple and already represent a single logical abstraction, it can be directly mapped, 1:1, to a *design class*. Typically, entity classes are converted into design classes. Since entity classes are typically also persistent, determine whether the design class should be persistent and note it accordingly in the class description.

When the analysis class is complex, such that it appears to embody behaviours that cannot be the responsibility of a single class acting alone, the analysis class should be mapped to a design subsystem. The design subsystem is used to encapsulate these collaborations in such a way that clients of the subsystem can be completely unaware of the internal design of the subsystem, even as they use the services provided by the subsystem.

In the design model, interfaces are mainly used to define the interfaces for subsystems. Interfaces are important for subsystems because they allow the separation of the declaration of behaviour from the realization of behaviour.

8.3 Analyze behaviour

After creating an initial architecture or producing architecture in previous iterations the purpose is to refining the architecture and analysing behaviour and creating an initial set of design model elements which provide the appropriate behaviour.

Steps of this workflow item are the followings:

- *Use case analysis.* Identification of the classes which perform a use case's flow of events and distribute the use case behaviour to those classes, using use-case realizations.
- *Identify design elements.* Analyzing interactions of analysis classes to identify new design model elements.
- *Review the design.* We have to verify that the design model fulfils the requirements on the system and serves as a good basis for its implementation. We have to ensure that the design model is consistent with respect to the general design guidelines.

8.4 Design components

After the initial design elements are identified, they are further refined. This workflow produces a set of components which provide the appropriate behaviour to satisfy the requirements on the system. In parallel with these activities, persistence issues are handled in Design the Database. The result is an initial set of components which are further refined in Implementation. The purpose of Design Components workflow detail is to:

- Refine and update the use-case realizations based on new design element identified.
- Refine the definitions of design elements (including capsules and protocols) by working out the details of how the design elements implement the behaviour required of them.
- Reviewing the design as it evolves.
- Implementing the design elements as components.
- Testing the implemented components to verify functionality and satisfaction of requirements at the component/unit level.

The architecture of system can be represented by a component diagram in UML. It shows a collection of model elements, such as components, and implementation subsystems, and their relationships, connected as a graph to each other. Component diagrams can be organized into, and owned by implementation sub-systems, which show only what is relevant within a particular implementation subsystem.

The following structures are suitable for illustration in component diagrams:

- Implementation sub-systems and their import dependencies.
- The implementation sub-systems organized in layers.
- Components (source code files) and their compilation dependencies.
- Components (applications) and their run-time dependencies.
- Important structure of components, for example to illustrate a typical use of a component.

8.4.1 Use case design

The purpose of this workflow item is:

- To refine use-case realizations in terms of interactions.
- To refine requirements on the operations of design classes.
- To refine requirements on the operations of subsystems and/or their interfaces.

8.4.2 Sub-system design

The purpose of this activity is:

- To define the behaviours specified in the subsystem's interfaces in terms of collaborations of contained classes.
- To document the internal structure of the subsystem.
- To define realizations between the subsystem's interfaces and contained classes.
- To determine the dependencies upon other subsystems

The external behaviours of the subsystem are defined by the interfaces it realizes. When a subsystem realizes an interface, it makes a commitment to support each and every operation defined by the interface. The operation may be in turn realized by:

- an operation on a class contained by the subsystem; this operation may require collaboration with other classes or subsystems

- an operation on an interface realized by a contained subsystem

The collaborations of model elements within the subsystem should be documented using sequence diagrams which show how the subsystem behaviour is realized. Each operation on an interface realized by the sub-system should have one or more documenting sequence diagrams. This diagram is owned by the subsystem, and is used to design the internal behaviour of the subsystem.

To document the internal structure of the sub-system, create one or more class diagrams showing the elements contained by the subsystem, and their associations with one another. One class diagram should be sufficient, but more can be used to reduce complexity and improve readability.

When an element contained by a sub-system uses some behaviour of an element contained by another sub-system, a dependency is created between the enclosing sub-systems. In figure 8.19. the UML representation of dependency between packages is shown. It expresses that package Payment schedule is dependent on package Billing.

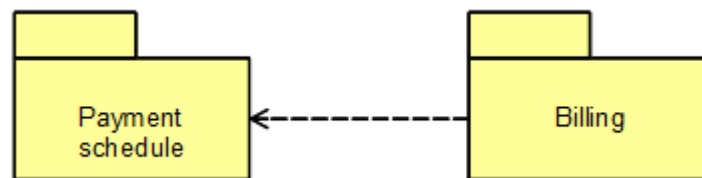


Figure 8.19. Dependency of sub-systems.

8.4.3 Class design

The initial design classes are created for the analysis class given as input to this activity, and assign trace dependencies. The design classes created in this step will be refined, adjusted in the subsequent design steps when various design properties assigned:

1. Create Initial Design Classes.
2. Identify Persistent Classes.
3. Define Class Visibility.
4. Define Operations.
5. Define Methods.
6. Define States.
7. Define Attributes.
8. Define Dependencies.
9. Define Associations.
10. Define Generalizations.
11. Handle Non-Functional Requirements in General.

Depending on the type of the analysis class (boundary, entity, or control) that is to be designed, there are specific strategies that can be used to create initial design classes.

Designing boundary classes

The general rule in analysis is that there will be one boundary class for each window, or one for each form, in the user interface. The consequence of this is that the responsibilities of the boundary classes can be on a fairly high level, and need then be refined and detailed in this step. The Figure 8.20. shows the class “Course registration” which is responsible for the control of the user interface of subject register system.

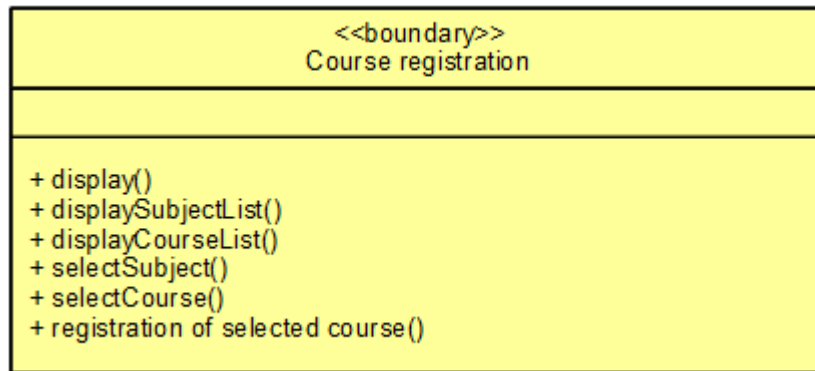


Figure 8.20. The Corse registration boundary class.

The design of boundary classes depends on the user interface (or GUI) development tools available to the project. Using current technology, it is quite common that the user interface is visually constructed directly in the development tool, thereby automatically creating user interface classes that need to be related to the design of control and/or entity classes. If the GUI development environment automatically creates the supporting classes it needs to implement the user interface, there is no need to consider them in design - it is only necessary to design what the development environment does not create.

Boundary classes which represent the interfaces to existing systems are typically modelled as subsystems, since they often have complex internal behaviour. If the interface behaviour is simple one may choose to represent the interface with one or more design classes.

Designing entity classes

During analysis, entity classes represent manipulated units of information; entity objects are often passive and persistent. In analysis, these entity classes may have been identified and associated with the analysis mechanism for persistence.

Designing control classes

A control object is responsible for managing the flow of a use case and thus coordinates most of its actions; control objects encapsulate logic that is not particularly related to user interface issues (boundary objects), or to data engineering issues (entity objects).

Define operations and methods

To identify Operations on design classes:

- Study the responsibilities of each corresponding analysis class, creating an operation for each responsibility. Use the description of the responsibility as the initial description of the operation.
- Study the use-case realizations in the class participates to see how the operations are used by the use-case realizations. Extend the operations, one use-case realization at the time, refining the operations, their descriptions, return types and parameters. Each use-case realization's requirements as regards classes are textually described in the Flow of Events of the use-case realization.
- Study the use case Special Requirements, in order not to miss implicit requirements on the operation that might be stated there.

Operations are required to support the messages that appear on sequence diagrams because scripts; messages (temporary message specifications) which have not yet been assigned to operations describe the behaviour the class is expected to perform.

Use-case realizations cannot provide enough information to identify all operations. To find the remaining operations, consider the following:

- Is there a way to initialize a new instance of the class, including connecting it to instances of other classes to which it is associated?
- Is there a need to test to see if two instances of the class are equal?
- Is there a need to create a copy of a class instance?
- Are any operations required on the class by mechanisms which they use (for example, a garbage collection mechanism may require that an object be able to drop all of its references to all other objects in order for unused resources to be freed)?

For each operation, you should define the following:

- *The operation name.* The name should be short and descriptive.
- *The return type.*
- *A short description.* Give the operation a short description consisting of a couple of sentences, written from the operation user's perspective.
- *The parameters.* The brief description of parameters should include the following:
 - a. The meaning of the parameters (if not apparent from their names).
 - b. Whether the parameter is passed by value or by reference
 - c. Parameters which must have values supplied
 - d. Parameters which can be optional, and their default values if no value is provided
 - e. Valid ranges for parameters (if applicable)
 - f. What is done in the operation.
 - g. Which by reference parameters are changed by the operation.

Once the operations have been defined the sequence diagrams are completed with information about which operations are invoked for each message. For each operation, identify the export visibility of the operation:

- *Public:* the operation is visible to model elements other than the class itself.
- *Implementation:* the operation is visible only within to the class itself.
- *Protected:* the operation is visible only to the class itself, to its subclasses, or to *friends* of the class (language dependent)
- *Private:* the operation is only visible to the class itself and to *friends* of the class

A method specifies the implementation of an operation. In many cases, methods are implemented directly in the programming language. The method describes how the operation works, not just what it does.

The method, if described, should discuss:

- How operations are to be implemented.
- How attributes are to be implemented and used to implement operations.
- How relationships are to be implemented and used to implement operations.
- What other objects and their operations are to be used?

Sequence diagrams are an important source for this. From these it is clear what operations are used in other objects when an operation is performed.

State chart diagram

For some operations, the behaviour of the operation depends upon the state the receiver object is in. A state machine is a tool for describing the states the object can assume and the events that cause the object to move from one state to another.

Each state transition event can be associated with an operation. Depending on the object's state, the operation may have a different behaviour. States are often represented using attributes; the state-chart diagrams serve as input into the attribute identification step.

State machines are used to model the dynamic behaviour of a model element. They are used to define state-dependent behaviour, or behaviour that varies depending on the state in which the model element is in. Model elements whose behaviour does not vary with its state of the element are passive classes whose primary responsible is to manage data.

A state machine consists of states, linked by transitions. A state is a condition of an object in which it performs some activity or waits for an event. An object may remain in a state for a finite amount of time. A transition is a relationship between two states which is triggered by some event, which performs certain actions or evaluations, and which results in a specific end-state. Internal transitions are transitions that are handled without causing a change in state. Figure 8.21. shows the UML representations of state. In the right side case the lower part of state can be used to define actions done by the object.



Figure 8.21. UML representations of state.

A transition has five properties: Source state, Event trigger, Guard condition, Action, Target state.

The *event trigger* is the event that makes the transition eligible if its guard condition is satisfied.

The *guard condition* is a boolean expression that is evaluated when the transition is triggered by the reception of the event trigger.

The *action* is an executable atomic computation, meaning that it cannot be interrupted by an event and therefore runs to completion. Entry and exit actions allow the same action to be dispatched every time the state is entered or left respectively. Internal transitions allow events to be handled within the state without leaving the state, thereby avoiding triggering entry or exit actions. The Figure 8.22. represent the actions related to a state.

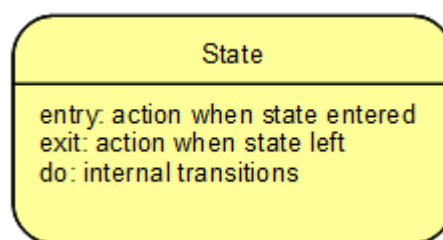


Figure 8.22. Representation of actions relating a state.

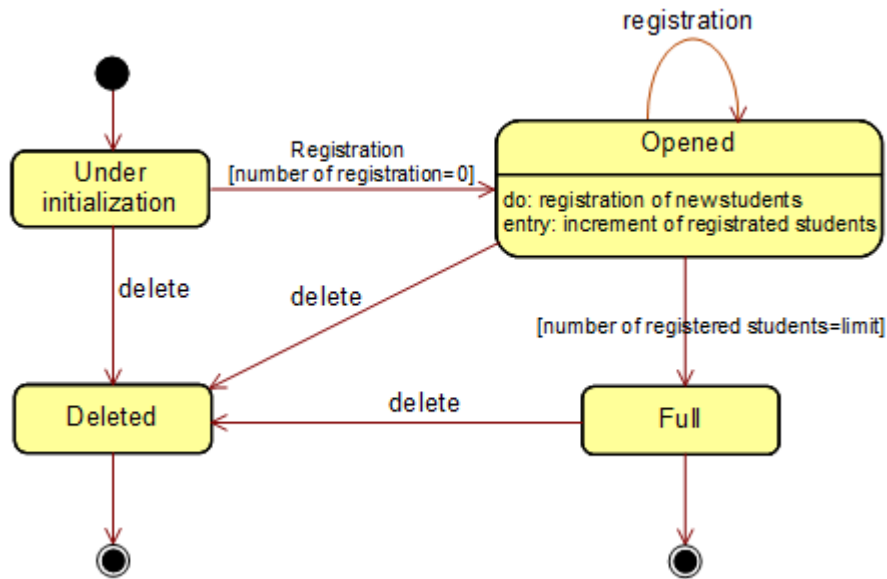


Figure 8.23. State-chart diagram of UML-01 course object.

Figure 8.23. shows the state transitions of UML-01 course object introduced in the sequence diagram in figure 8.8. It shows four states and five transitions. Following the initial state symbol the object enter the state *under initialization*. The object may have two final states. The object may enter the state *full* when the number of course students reaches a limit number. The another final states is the state *deleted*. Object enters this state due to any course delete event. The fourth state is the state *opened*. The object remains in this state until the number of registered course students number is under the course limit.

Define attributes

Attributes provide information storage for the class instance, and are often used to represent the state of the class instance. Any information the class itself maintains is done through its attributes. For each attribute, define the following:

1. Its *name*, which should follow the naming conventions of both the implementation language and the project.
2. Its *type*, which will be an elementary data type supported by the implementation language.
3. Its *default or initial value*, to which it is initialized when new instances of the class are created.
4. Its *visibility*: Public, Protected, Private or Implementation.
5. For persistent classes, whether the attribute is persistent (the default) or transient.

Define dependencies

For each case where the communication between objects is required dependency between sender and receiver has to be established in the following case:

- The reference to the receiver passed as an parameter to the operation. The visibility of link is set to „parameter” in *collaboration diagram*.
- The receiver a global object. In the *collaboration diagram* the visibility of link is set to „global”.
- The receiver a temporary object created and destroyed during the operation itself. The visibility of link is set to „local” in the *collaboration diagram*.

8.5 Implementation

The purpose of implementation is:

- to define the organization of the code, in terms of implementation subsystems organized in layers
- to implement classes and objects in terms of components (source files, binaries, executables, and others)
- to test the developed components as units
- to integrate the results produced by individual implementers (or teams), into an executable system

The implementation is consists of the following workflow details:

- *Structure of implementation model.* The purpose of this workflow detail is to ensure that the implementation model is organized in such a way as to make the development of components and the build process as conflict-free as possible. A well-organized model will prevent configuration management problems and will allow the product to built-up from successively larger integration builds.
- *Plan the integration.* The purpose of this workflow detail is to plan which subsystems should be implemented, and the order in which the subsystems should be integrated in the current iteration.
- *Implement components.* Writing source code, adapt existing components, compile, link and perform unit tests, as they implement the classes in the design model. If defects in the design are discovered, the implementer submits rework feedback on the design. The implementers also fix code defects and perform unit tests to verify the changes. Then the code is reviewed to evaluate quality and compliance with the Programming Guidelines.
- *Integrate each subsystem.* Integration of the new and changed components from the individual implementers into a new version of the implementation subsystem.
Integrate the System. Integration of the system, in accordance with the integration build plan, by adding the delivered implementation subsystems into the system integration workspace and creating builds. Each build is then integration tested by a tester.

8.6 Deployment

The Deployment Discipline describes the activities associated with ensuring that the software product is available for its end users. The Deployment Discipline describes several modes of product deployment. The main activities of Plan Deployment workflow detail are the followings:

- *Plan the product deployment.* Deployment planning needs to take into account how and when the product will be available to the end user.
- *Forming a high degree of customer collaboration.* A successful conclusion to a software project can be severely impacted by factors outside the scope of software development such as the building, hardware infrastructure not being in place, and the staff being ill-prepared for cut-over to the new system.
- *Support of the system.* The Deployment Plan needs to address not only the deliverable software, but also the development of training material and system support material to ensure that end users can successfully use the delivered software product.

Once the product has been tested at the development site it is prepared for delivery to the customer. The support materials that is required to install, operate, use and maintain the

delivered system are parts of the delivered system. In order to introduce the software as product to the software market and deliver it to end users the following activities are included in deployment discipline:

- *Develop support material.* The purpose of this workflow detail is to produce the support materials needed to effectively deploy the product to its users.
- *Manage acceptance test.* Acceptance testing is formal testing conducted to determine whether or not a system satisfies its acceptance criteria, and to enable the customer, user or authorized entity to determine whether or not to accept the system. Acceptance testing is often conducted at the development site, and then at the customer site using the target environment.
- *Produce deployment unit.* In this workflow a deployment unit is created that consists of the software, and the necessary accompanying artifacts required to effectively install and use it.
- *Package product.* This workflow detail describes the necessary activities to create the final software product by packaging the deployment unit, installation scripts, and user manuals together.
- *Provide access to download site.* The purpose of this workflow detail is to make the product available for purchase, and download over the internet.
- *Beta test product.* Development of a beta version of software helps to get feedback information on the product from users.

8.7 Exercises

1. List the workflow details of Analysis & Design discipline!
2. What is software architecture?
3. What are the responsibilities of classes?
4. What is the use case realisation?
5. What is the dependency of software components?
6. What is stereotype?
7. List stereotypes are used Analysis phase!
8. List the graphical elements of collaboration diagram!
9. List the graphical elements of state diagram!
10. What is the function of association classes?
11. What are the different between aggregation and composition relationships?

9 Software testing

The general testing process starts with the testing of individual program units such as functions or objects. The aim of the component testing is to discover defects by testing individual program components. Components are usually integrated to form sub-systems and finally to the complete system. System testing focuses on establishing that the sub-system or the complete system meet its functional and non-functional requirements, and does not behave in unexpected ways. Component testing by developers is based on understanding of how the components should operate using test data. However, system testing is rigorously based on written system specifications [1].

The software testing process has the following main goals:

1. To demonstrate that the software meets its requirements. This means that there should be at least one test for every user and system requirements.
2. To discover faults or defects in the software. The objective of defect testing is the exploration and elimination of all kinds of undesirable system behaviour, such as system crashes, unwanted interactions with other systems, incorrect computations and data corruption.

The first goal is called validation testing. In this case the system is tested by a given set of test cases that reflect the system's expected use. For validation testing, a test is considered to be successful if the system performs correctly. The second goal leads to defect testing, where the test cases are designed to expose defects. In this case, a successful test is one that exposes a defect that causes the system to perform incorrectly.

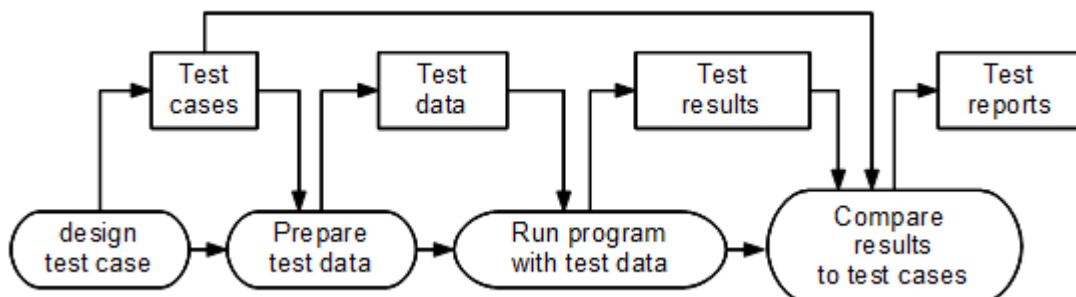


Figure 9.1. A model of the software testing process.

A general model of the testing process is shown in Figure 9.1. Test cases are specifications of the inputs to the test, the expected output from the system and a statement of what is being tested. Test data are the inputs that have been created to test the system. Testing is based on a subset of possible test cases.

9.1 Unit testing

Unit testing tests the individual components in the system. It is a defect testing process so its purpose is to locate faults in these components. In most cases, the developers of components are responsible for component testing. The components may be tested at unit testing are the followings:

1. Object classes, functions or methods within an object.
2. Composite components that are consist of several different objects or functions. The composite components have a defined interface that can be used to access their functionality.

Functions or methods are tested by a set of calls to these routines with different input parameters. In the case of testing object classes the tests have to cover all of the features of the object. Object class testing includes the following tests:

1. The testing all operations associated with the object
2. The testing adjustability of all attributes associated with the object
3. The testing all possible states of the object. This means that all events that cause a state change in the object have to be simulated.

In the case of inheritance the tests of object classes are more difficult. Where a superclass provides operations that are inherited by a number of subclasses, all of these subclasses should be tested with all inherited operations.

9.1.1 Interface testing

In a software system many components are not simple functions or objects but they are composite components that consist of several interacting objects. The functionality of these components can be accessed through their defined interface. Testing these composite components is primarily concerned with testing the interface of the composite component created by combining these components. Interface testing is important for object-oriented and component-based development. Objects and components are defined by their interfaces and may be reused in combination with other components in different systems.

9.2 System testing

System testing involves integrating two or more components that implement any system functions or features and then testing this integrated system. In an iterative development process, system testing is concerned with testing an increment to be delivered to the customer. In the case of a waterfall development process, system testing is concerned with testing the entire system. Therefore, system testing may have two distinct phases:

1. *Integration testing*. After integrating a new component the integrated system is tested. When a problem is discovered, the developers try to find the source of the problem and identify the components that have to be debugged.
2. *Functional testing*. The version of the system that could be released to users is tested. Here, the main objective is to validate the system that it meets its requirements and ensure that the system is dependable. Where customers are involved in release testing, this is called *acceptance testing*. If the release is good enough, the customer may then accept it for use.

Fundamentally, the integration testing is considered as the testing of incomplete systems composed of clusters or groupings of system components. Functional testing is concerned with testing the system release that is intended for delivery to customers. Generally, the priority in integration testing is to discover defects in the system and the priority in system testing, is to validate that the system meets its requirements.

9.2.1 Integration testing

The process of system integration involves building a system from its components and testing the resultant system for problems that arise from component interactions. Integration testing checks how these components work together across their interfaces.

System integration integrates clusters of components that deliver some system functionality and integrating these by adding code that makes them work together. Sometimes, the overall skeleton of the system is developed first, and components are added to it. This is called *top-down integration*. Alternatively, the infrastructure components are

integrated first providing common services, such as network and database access, and functional components added after. This is *bottom-up integration*.

In order to locate errors easier an incremental approach to the system integration and testing is suggested. Initially, a minimal system configuration should be integrated and tested. Then the new components are added to this minimal configuration and tested after each added increment. If any problem arises in these tests, this probably means that they are due to interactions with the new component.

Before the process of integration, it is necessary to decide the order of integration of components. Usually, system integration is driven by customer priorities. When the customer is not involved in the developments, the components that have the most frequently used functionality are integrated first. Integrating and testing a new component can change the already tested component interactions. Errors may occur that were not exposed in the tests of the simpler configuration. This means that when a new increment is integrated, it is important to rerun the tests for previous increments as well. Rerunning an existing set of tests is called *regression testing*.

9.2.2 Functional testing

A release of the software is its final version that will be distributed to customers. The objective of release testing is to show that the software product delivers the specified functionality, performance and dependability, and that it does not fail during normal use. Another name of release testing is *functional testing* because it is only concerned with the functionality and not the implementation of the software.

Release testing is usually a black-box testing process where the tests are derived from the system specification. Figure 9.2. shows the model of black-box testing. The testing process presents inputs to the component or the system and examines the corresponding outputs. If the outputs are not those predicted, i.e. if the outputs are in set O_e , then the test has detected a problem with the software.

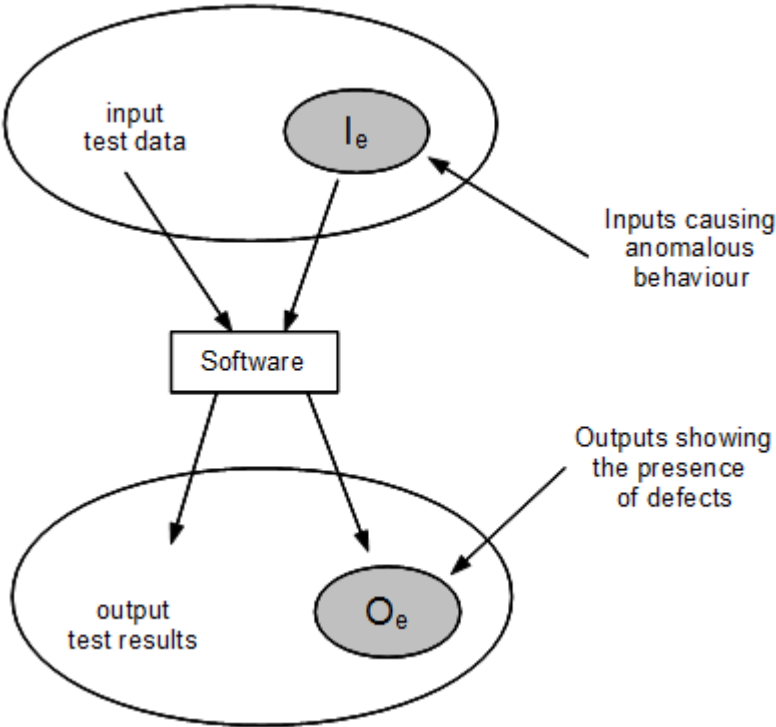


Figure 9.2. Black-box testing model.

During release testing, software is often tested by choosing test cases that are in the set I_e in Figure 9.2. These inputs have a high probability of generating system failures i.e. outputs in set O_e .

To validate that the system meets its requirements, the best approach to use is scenario-based testing, where test cases are developed from scenarios. Usually, the most likely scenarios are tested first, and unusual or exceptional scenarios considered later. If use-cases are used to describe the system requirements, these use-cases and associated sequence diagrams can be a basis for system testing.

9.2.3 Performance testing

The objective of performance tests is to ensure that the system can process its intended load. An effective way to discover defects is the *stress testing*, i.e. to design tests around the limits of the system. In performance testing, this means stressing the system by making demands that are over the design limits of the software. Stress testing has two functions:

1. It tests the failure behaviour of the system under circumstances where the load placed on the system exceeds the maximum load.
2. It continuously stresses the system and may cause such defects that would not normally be discovered.

9.3 Exercises

What are the objectives of software testing?

Review the process of software testing!

What are the types of test?

What does the defect testing mean?

What software components are tested by unit test?

What is the regression testing?

What is the different between integration and functional testing?

What is the different between release and acceptance testing?

Explain the functional testing!

What is the objective of performance testing?

10 Embedded system development

An embedded system is a computer system with a dedicated function within a larger mechanical or electrical system that serves a more general purpose, often with real-time computing constraints. It is embedded as part of a complete device often including hardware and mechanical parts.

Embedded systems are designed to do some specific task, rather than be a general-purpose computer for multiple tasks. Some also have real-time performance constraints that must be met, for reasons such as safety and usability; others may have low or no performance requirements, allowing the system hardware to be simplified to reduce costs. Since the embedded system is dedicated to specific tasks, design engineers can optimize it to reduce the size and cost of the product and increase the reliability and performance.

The processors used in embedded systems may be types ranging from rather general purpose to very specialized in certain class of computations, or even custom designed for the application at hand. A common standard class of dedicated processors is the digital signal processor (DSP).

The program instructions written for embedded systems are referred to as firmware, and are stored mainly in read-only memory or Flash memory chips. They run with limited computer hardware resources: little memory, small or non-existent keyboard or screen. As with other software, embedded system designers use compilers, assemblers, and debuggers to develop embedded system software.

The embedded system interacts directly with hardware devices and mostly must respond, in real time, to events from the system's environment. In the real-time systems the *embedded real-time software* must react to events generated by the hardware and issue control signals in response to these events.

Embedded systems control many devices in common use today. They are commonly found in consumer, cooking, industrial, automotive, medical, commercial and military applications. Physically, embedded systems range from portable devices such as digital watches and MP3 players, to large stationary installations like traffic lights, factory controllers and largely complex systems like hybrid vehicles, MRI. Telecommunications systems employ numerous embedded systems from telephone to cell phones. Many household appliances, such as microwave ovens, washing machines and dishwashers, include embedded systems to provide flexibility, efficiency and features. Transportation systems from flight to automobiles increasingly use embedded systems.

Computers are used to control a wide range of systems from simple domestic machines to entire manufacturing plants. These computers interact directly with hardware devices. The software in these systems is *embedded real-time software* that must react to events generated by the hardware from the environment of system and issue control signals in response to these events.

Software failures are relatively usual. In most cases, these failures cause inconvenience but no serious, long-term damage. However, in some systems failure can result in significant economic losses, physical damage or threats to human life. These systems are called *critical systems*. Critical systems are technical or socio-technical systems that people or businesses depend on. If these systems fail to deliver their services as expected then serious problems and significant losses may result. Modern electronic systems increasingly make use of embedded computer systems to add functionality, increase flexibility, controllability and performance. However, the increased use of embedded software to control systems brings

with it certain risks. This is especially significant in safety critical systems where human safety is dependent upon the correct operation of the system.

The objective of this chapter is to introduce main characteristics of critical systems and the implementation techniques that are used in the development of critical and real-time systems.

10.1 Critical systems

There are three main types of critical systems [1]:

1. *Safety-critical systems.* A system whose failure may result in injury, loss of life or serious environmental damage.
2. *Mission-critical systems.* A system whose failure may result in the failure of some goal-directed activity.
3. *Business-critical systems.* A system whose failure may result in very high costs for the business using that system.

The most important property of a critical system is its *dependability*. The term dependability covers the related systems attributes such as availability, reliability, safety and security.

There are three system components where critical systems failures may occur:

1. System hardware components may fail because of mistakes in their design or manufacturing errors.
2. System software may fail due to mistakes in its specification, design or implementation.
3. Human operators of the system may fail to operate the system correctly.

Because of the high cost of critical systems failure, trusted methods and well-known techniques must be used for development of these systems. Most critical systems are socio-technical systems where people monitor and control the operation of computer-based systems. Operators in these systems must successfully treat unexpected situations and cope with additional workload. However, this may cause more stress and so on mistakes.

10.1.1 System dependability

The dependability is a property of systems. A dependable computer system provides a trustworthy operation to users. This means that system is expected to not fail in normal use. There are four principle attributes to dependability:

1. *Availability.* The availability of a system is the probability that the system can provide the services requested by users at any time.
2. *Reliability.* The reliability of a system is the probability, over a given period of time, that the system will correctly deliver services.
3. *Safety.* The safety of a system shows the extent of damage may be caused by the system to people or its environment.
4. *Security.* The security of a system shows that how the system can resist accidental or deliberate unauthorized intrusions.

Besides these four attributes, other system properties can also be related to dependability:

1. *Reparability.* Disruption caused by any failure can be minimized if the system can be repaired as soon as possible. Therefore, it is important to be able to diagnose the problem, access the component that has failed and make changes to fix that component.
2. *Maintainability.* Once new requirements are emerged it is important to maintain the system by integration of new functionalities required.

3. *Survivability*. Survivability is the ability of a system to continue operation of the service during a possible attack, even at the loss of certain parts of the system.
4. *Error tolerance*. This property is considered as part of usability and shows how the system is designed to avoid and tolerate user input errors.

System developers have usually to prioritize system performance and system dependability. Generally, high levels of dependability can only be achieved at the expense of system performance. Because of the additional design, implementation and validation costs, increasing the dependability of a system can significantly increase development costs.

10.1.1.1 Availability and reliability

The *reliability* of a system is the probability that the system correctly provides services as defined in its specification. In other words, the reliability of software can be related to the probability that the system input will be a member of the set of inputs, which cause an erroneous output to occur. If an input causing an erroneous output is associated with a frequently used part of the program, then failures will be frequent. However, if it is associated with rarely used code, then users will hardly complain about failures.

The *availability* of a system is the probability that the system will provide its services to users when they request them. If users need for continuous service then the availability requirements are high.

Reliability and availability are primarily compromised by system failures. These may be a failure to provide a service, a failure to deliver a service as specified, or the delivery of a service unsafely and insecurely. However, many failures are a consequence of erroneous system behaviour that derives from faults in the system. To increase the reliability of a system the following approaches can be used:

1. *Fault avoidance*. Program development techniques are used that can minimize the possibility of mistakes and/or eliminate mistakes before they cause system faults.
2. *Fault detection and removal*. The use of verification and validation techniques that effectively helps to detect and remove the faults before the system is used.
3. *Fault tolerance*. Techniques that ensure that faults in a system do not result in system errors or that ensure that system errors do not result in system failures.

10.1.2 Safety

The essential feature of safety-critical systems is that system operation is always safe. These systems never compromise or damage people or the environment of the system, even if the system fail. Safety-critical software has two groups:

1. *Primary safety-critical software*. This software is usually embedded as a controller in a system. Malfunctioning of such software can cause a hardware malfunction, which results in human injury and/or environmental damage.
2. *Secondary safety-critical software*. This is software that can indirectly result in injury. For an example, software used for design has a fault can causes the malfunction of designed system and this may results in injury to people.

The safe operation, i.e. ensuring either that accidents do not occur or that the consequences of an accident are minimal, can be achieved in the next ways:

1. *Hazard avoidance*. This type of system is designed so that hazards are avoided. For example, a safe cutting system equipped with two control buttons, where the two buttons can be operated by using separate hands.

2. *Hazard detection and removal.* The system is designed so that hazards are detected and removed before they result in an accident. For example, pressure control in a chemical reactor system can reduce the detected excessive pressure before an explosion occurs.
3. *Damage limitation.* These systems have a functionality that can minimize the effects of an accident. For example, automatic fire extinguisher systems.

10.1.3 Security

Security has become increasingly important attributes of systems connecting to the Internet. Internet connections provide additional system functionality, but it also allows systems to be attacked by people with hostile intentions. Security is a system attribute that shows the ability of the system to protect itself from against accidental or deliberate external attacks. In some critical systems such as systems for electronic commerce, military systems, etc., security is the most important attribute of system dependability.

Examples of attacks might be viruses, unauthorized use of system services and data, unauthorized modification of the system, etc. Security is an important attribute for all critical systems. Without a reasonable level of security, the availability, reliability and safety of the system may be compromised if external attacks cause some damage to the system. There are three types of damage that may be caused by external attack:

1. *Denial of service.* In this case of attack the system is forced into a state where its normal services become unavailable.
2. *Corruption of programs or data.* The software components of the system are damaged affecting reliability and safety of system.
3. *Disclosure of confidential information.* Confidential information managed by the system is exposed to unauthorized people as a consequence of the external attack.

The security of a system may be assured using the following methods:

1. *Vulnerability avoidance.* The system is designed not to be vulnerable. For example, if a system is not connected to Internet there is no possibility of external attacks.
2. *Attack detection and neutralization.* The system is designed so that it detects and removes vulnerabilities before any damage occurs. An example of vulnerability detection and removal is the use of a virus checker to remove infected files.
3. *Exposure limitation.* In these methods the consequences of attack are minimized. An example of exposure limitation is the application of regular system backups.

10.2 Critical systems development

Due to the quick progress in computer technology, improvement of software development methods, better programming languages and effective quality management the dependability of software has significantly improved in the last two decades. In system development special development techniques may be used to ensure that the system is safe, secure and reliable. There are three complementary approaches can be used to develop dependable software:

1. *Fault avoidance.* The design and implementation process are used to minimize the programming errors and so on the number of faults in a program.
2. *Fault detection.* The verification and validation processes are designed to discover and remove faults in a program before it is deployed for operational use.
3. *Fault tolerance.* The system is designed so that faults or unexpected system behaviour during execution are detected and managed in such a way that system failure does not occur.

Redundancy and diversity are fundamental to the achievement of dependability in any system. Examples of redundancy are the components of critical systems that replicate the functionality of other components or an additional checking mechanism that is added to system but not strictly necessary for the basic operation of system. Faults can therefore be detected before they cause failures, and the system may be able to continue operating if individual components fail. If the redundant components are not the same as other components, is the case of diversity, a common failure in the same, replicated component will not result in a complete system failure.

Software engineering research intended to develop tools, techniques and methodologies that lead to the production of fault-free software. Fault-free software is software that exactly meets its specification. Of course, this does not mean that the software will never fail. There may be errors in the specification that may be reflected in the software, or the users may misunderstand or misuse the software system. In order to develop fault-free software the following software engineering techniques must be used:

1. *Dependable software processes.* The use of a dependable software process with appropriate verification and validation activities can minimize the number of faults in a program and detect those that do slip through.
2. *Quality management.* The software development organization must have a development culture in which quality drives the software process. Design and development standards should be established that provide the development of fault-free programs.
3. *Formal specification.* There must be a precise system specification that defines the system to be implemented.
4. *Static verification.* Static verification techniques, such as the use of static analysers, can find anomalous program features that could be faults.
5. *Strong typing.* A strongly typed programming language such as Java must be used for development. If the programming language has strong typing, the language compiler can detect many programming errors.
6. *Safe programming.* Some programming language constructs are more complex and error-prone than others. Safe programming means avoiding or at least minimizing the use of these constructs.
7. *Protected information.* Design and implementation processes based on information hiding and encapsulation is to be followed. Object-oriented languages such as Java satisfy this condition.

Although, development of fault-free software by application of these techniques is possible, it is economically disadvantageous. The cost of finding and removing remaining faults rises exponentially as faults in the program are discovered and removed. While the software becomes more dependable more tests are needed to find fewer and fewer faults.

10.2.1 Fault tolerance

A fault-tolerant system can continue its operation even after some of its part is faulty or not reliable. The fault-tolerance mechanisms in the system ensure that these system faults do not cause system failure. Where system failure could cause a catastrophic accident or where a loss of system operation would cause large economic losses it is necessary to develop fault-tolerant system. There are four complementary approaches to ensure fault-tolerance of a system:

1. *Fault detection.* The system must detect a fault that causes a system failure. Generally, this based on checking consistency of the system state.

2. *Damage assessment.* The parts of the system state that have been affected by the fault must be detected.
3. *Fault recovery.* The system restores its state to a known safe state. This may be achieved by correcting the damaged state or by restoring the system to a known safe state.
4. *Fault repair.* This involves modifying the system so that the fault does not recur.

10.2.1.1 Fault detection and damage assessment

The first stage in ensuring fault tolerance is to detect that a fault either has occurred or will occur unless some action is taken immediately. To achieve this, the illegal values of state variables must be recognized. Therefore, it is necessary to define state constraints that define the conditions that must always hold for all legal states. If these predicates are false, then a fault has occurred.

Damage assessment involves analyzing the system state to estimate the extent of the state corruption. The role of the damage assessment procedures is not to recover from the fault but to assess what parts of the state space have been affected by the fault. Damage can only be assessed if it is possible to apply some validity function that checks whether the state is consistent.

10.2.1.2 Fault recovery and repair

The purpose of fault recovery process is to modify the state of the system so that the effects of the fault are eliminated or reduced. The system can continue to operate, perhaps in some degraded form. *Forward recovery* tries to correct the damaged system state and to create the intended state. Forward recovery is only possible in the cases where the state information includes built-in redundancy. *Backward recovery* restores the system state to a known correct state.

For an example, most database systems include backward error recovery. When a user starts a database operation, a transaction is initiated. The changes made during that transaction are not immediately incorporated in the database. The database is only updated after the transaction is finished and no problems are detected. If the transaction fails, the database is not updated.

10.3 Real-time software design

The real-time embedded systems are significantly different from other types of software systems. Their correct operation is dependent on the system responding to events within a short time interval. The real-time system can be shortly defined as follows:

A real-time system is a software system where the correct operation of the system depends on the results produced by the system and the time at which these results are produced.

Timely response is an important factor in all embedded systems but, in some cases, very fast response is not necessary. The real-time system is a stimulus/response system. It must produce a corresponding response for a particular input stimulus. Therefore, the behaviour of a real-time system can therefore be defined by listing the stimuli received by the system, the associated responses and the time at which the response must be produced. Stimuli has two classes:

1. *Periodic stimuli.* These stimuli are generated at predictable time intervals.
2. *Aperiodic stimuli.* These stimuli occur irregularly.

Periodic stimuli in a real-time system are usually generated by sensors associated with the system and provide information about the state of the system's environment. The responses of

system are transmitted to actuators that may control some equipment. Aperiodic stimuli may be generated either by the actuators or by sensors. This sensor-system-actuator model of an embedded real-time system is illustrated in Figure 10.1.

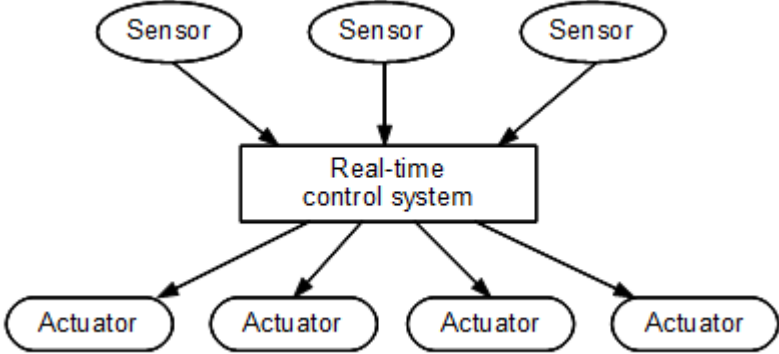


Figure 10.1. General model for a real-time system.

A real-time system must be able to respond to stimuli that occur at different times. Therefore, architecture should be designed so that, as soon as a stimulus is received, control is transferred to the correct handler. This cannot be achieved using sequential programs. Consequently, real-time systems are normally designed as a set of concurrent and cooperating processes. In order to manage these concurrent processes most real-time systems include a *real-time operating system*.

The stimulus-response model of a real-time system consists of three processes. Each type of sensor has a sensor management process, computational processes to compute the required response for the stimuli received by the system and control processes for actuator to manage their operation. This stimulus-response model enables rapid collection of data from the sensor and allows the computational processes and actuator responses to be carried out later.

10.3.1 System design

Designing a real-time system it is necessary to decide first which system capabilities are to be implemented in software and which in hardware. Then the design process of real-time software focuses on the stimuli rather than the objects and functions. The design process has a number of overlapped stages:

1. Identification of the stimuli that the system must process and the associated responses.
2. Specifying the timing constraints for each stimulus and associated response.
3. Selection of hardware components and the real-time operating system to be used.
4. Aggregation of the stimulus and response processing into a number of concurrent processes. It is usual in real-time systems design is to associate a concurrent process with each class of stimulus and response as shown in Figure 10.2.
5. Design of algorithms of the required computations for each stimulus and response.
6. Design a scheduling system ensuring that processes are started and completed in time.

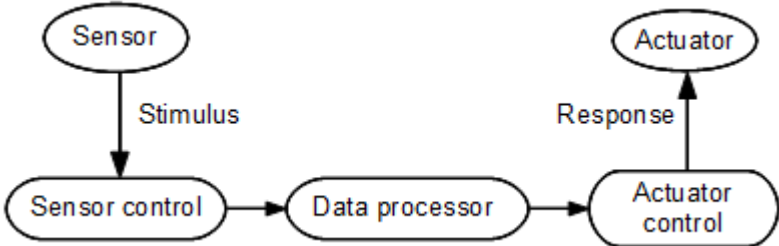


Figure 10.2. Sensor – actuator control process.

Processes must be coordinated in a real-time system. Process coordination mechanisms ensure mutual exclusion to shared resources. Once the process architecture has been designed and scheduling policy has been decided it should be checked that the system will meet its timing requirements.

Timing constraints or other requirements often mean that some system functions, such as signal processing, should be implemented in hardware rather than in software. Hardware components can provide a better performance than the equivalent software.

10.3.1.1 Real-time system modelling

Real-time systems have to respond to events occurring at irregular intervals. These stimuli often cause the system to move to a new state. For this reason, state machine models are often used to model real-time systems. Application of state machine models is an effective way to represent the design of a real-time system. The UML supports the development of state models based on state-charts. A state model of a system assumes that the system, at any time, is in one of a number of possible states. When a stimulus is received it may cause a transition to a different state.

10.3.2 Real-time operating systems

Most of the embedded systems have real-time performance constraints that mean they have to work in conjunction with a real-time operating system (RTOS). Real-time operating systems guarantee a certain capability within a specified time constrain. It manages processes and resource allocation in a real-time system. It can starts and stops processes and allocate memory and processor resources, so that stimuli can be handled as concurrent processes.

Real-time operating systems usually include the following components:

1. *Real-time clock*. This provides information to schedule processes periodically.
2. *Interrupt handler*. This manages aperiodic requests for service.
3. *Process manager*. It is responsible for scheduling processes to be executed.
4. *Resource manager*. Resource manager allocates resources (memory, processor, etc.) to processes.
5. *Dispatcher*. It is responsible for starting the execution of a process.

10.3.2.1 Process management

Real-time systems have to respond events from the hardware in real time. The processes handling events must be scheduled for execution and must be allocated processor resources to provide their deadline. In real-time operating systems the process manager is responsible for selecting the next process to be executed, allocating resources such as processor and memory resources and starting and stopping the process.

The process manager has to manage processes having different priority. Real-time operating systems define different priority levels for system processes:

1. *Interrupt level*. This is the highest priority level. It is allocated to processes that need a very fast response.
2. *Clock level*. This level of priority is assigned to periodic processes.
3. *Background processes level*. This is the lowest priority level. It is allocated to background processes that have no timing constraints. These processes are scheduled for execution when processor capacity is available.

In most real-time systems, there are several types of periodic processes. They usually control the data acquisition and the actions of actuators. Periodic processes have different execution time and deadline. The timing requirements of all processes are specified by the

application program. The real-time operating system manages the execution of periodic processes and ensures that every process have to be completed by their deadline.

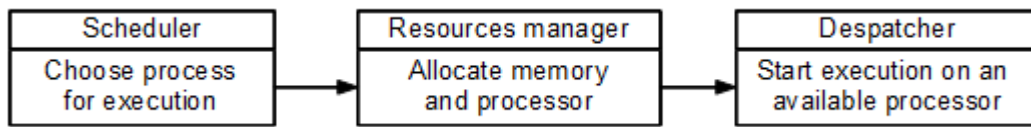


Figure 10.3. RTOS actions required to start a process.

Figure 10.3. shows the sequence of activities that are performed by the operating system for periodic process management. The scheduler examines all the periodic processes and chooses a process to be executed. The choice depends on the process priority, the process periods, the expected execution times and the deadlines of the ready processes.

10.4 Exercises

1. What does the embedded system mean?
2. Give examples of embedded system!
3. What are the three types of critical systems. What is the different between them?
4. What are the main dimensions of system dependability?
5. What approaches can be used for dependable software development?
6. List some programming techniques that are not recommended in safe programming!
7. What does the fault-tolerant system mean?
8. What systems are called real-time systems?
9. What types of stimulus are the real-time systems designed for?
10. Explain the process management of real-time systems!

11 Project management

Software engineering projects are always subject to organizational budget and schedule constraints. Software project managers are the responsible person for planning and scheduling development project. They supervise the development work to ensure that it is carried out to the organizational standards and monitor the progress of the project. Project manager is also responsible for checking the development cost. Software engineering is differs from other engineering activities in a number of ways [1].

The manager of a civil engineering project can see how a product is being developed. However, software is an intangible product. Software project managers can only see the progress of software development by documents produced to review the software process. In manufacturing the production processes are continuously tried and tested and finally they may be standardized. The engineering processes for many systems are well understood. However, there are no standard software processes and software processes are very different from one organization to another. Due to the rapid technological changes in computers and communications software projects are usually different in some ways from previous projects. Experiences form previous projects usually may not be transferred to a new development project.

Because of these problems some software projects may late, over budget and behind schedule.

11.1 Management activities

The job of a software manager depends on the organization and the software product being developed. However, most managers have a certain responsibility for the following activities relating to a software project:

- *Proposal writing.* The first task of managers in a software project is writing a proposal. It describes the objectives of the project and how it will be carried out. It usually includes cost and schedule estimates.
- *Project planning and scheduling.* Project planning is concerned with identifying the activities, milestones and deliverables produced by the development project.
- *Project cost estimation.* Cost estimation is concerned with estimating the resources required to accomplish the project plan.
- *Project monitoring and reviews.* Project monitoring is a continuing project activity. The manager monitors the progress of the project and compares the actual and planned progress and costs. Project reviews are concerned with reviewing overall progress and technical development of the project and checking whether the project and the goals are still aligned.
- *Personnel selection and evaluation.* Project managers are usually responsible for selecting people with appropriate skill and experience to work on the project.
- *Report writing and presentations.* Project managers are usually responsible for reporting on the project to both the client and contractor organizations. They must be able to present this information during progress reviews.

11.2 Project planning

The effective management of a software project greatly depends on careful planning the progress of the project. The plan prepared at the start of a project is considered an initial plan and it should be used as the driver for the entire project. The initial plan should be the best possible plan given by the available information. It evolves as the project progresses and more information becomes available.

The subsection 11.2.1. discusses the structure of the software project plan. In addition to the project plan managers have to prepare other types of plans such as quality plan, maintenance plan, etc. The planning is an iterative process. Effectively, the project plan can only be considered to be completed when the project has finished. As more information becomes available during the project, the project plan should be revised.

The planning process starts by defining constraints affecting the project (required delivery time, availability of labour, the total budget, etc.). In conjunction with this, project parameters such as its structure, size and project deliverables are defined. The process then enters a loop. Milestones and project schedule are defined and the scheduled activities are started. After a certain time the progress of project is reviewed and the deviation from the planned schedule are noted.

Project manager has to revise his assumptions related to project plan as more information becomes available. If the project is delayed, project has to be rescheduled and the project constraints and deliverables have to be negotiated with the customer.

11.2.1 The project plan

The project plan sets out the resources available to the project, the work breakdown and a schedule for carrying out the work. Most project plans includes the following sections:

1. *Introduction.* This summarizes the objectives of the project and sets out the budget, time and other constraints.
2. *Project organization.* This defines the development team, its members and the roles in the team.
3. *Hardware and software resource requirements.* This specifies the hardware and the support software allocated to development activities.
4. *Work breakdown.* This sets out the breakdown of the project into activities and identifies the milestones and deliverables associated with each activity.
5. *Project schedule.* Project schedule shows the dependencies between activities, the estimated time required to complete activities and the allocation of people to activities.
6. *Risk analysis.* This describes the possible project risks and the strategies to manage them.
7. *Monitoring and reporting mechanisms.* This defines the management reports that should be produced, when these should be produced and the project monitoring mechanisms used.

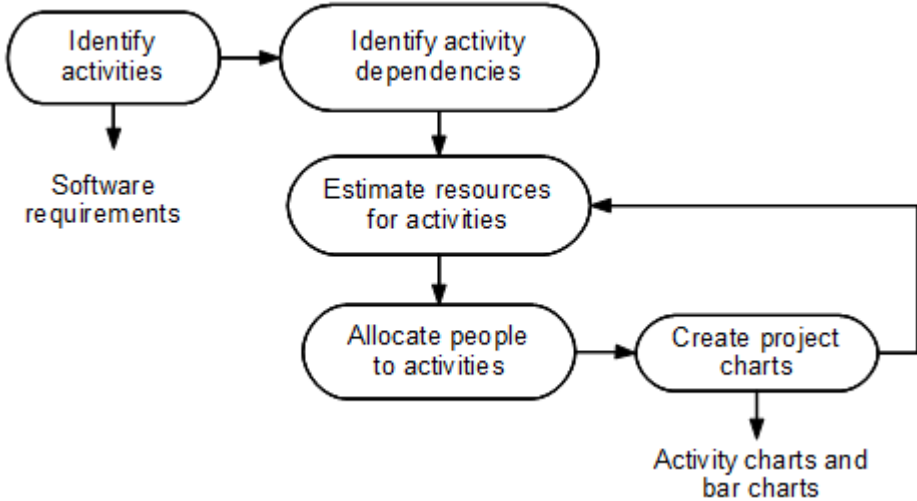


Figure 11.1. The project scheduling process.

11.3 Project scheduling

In the project scheduling process project managers estimate the time and resources required to complete activities and organize them into a coherent sequence (Figure 11.1.). Usually, some of activities can be carried out in parallel. Managers have to coordinate these parallel activities and organize the work so that the labour is used optimally.

The duration of project activities are normally at least a week. Project manager has to estimate the resources needed to complete each task. The principal resource is the human effort required. Other resources are related to the hardware and software required to development activities.

Project schedules are usually represented as a set of charts showing the work breakdown, activities dependencies and staff allocations. Bar charts (for example *Gantt* chart) and activity networks are graphical notations that are used to illustrate the project schedule. Bar charts show who is responsible for each activity and when the activity is scheduled to begin and end. Activity networks show the dependencies between the different activities. Activity networks help to identify which activities can be carried out in parallel and which must be executed in sequence because of a dependency on an earlier activity.

11.4 Risk management

Risk management is one of the main activities of project managers. It involves anticipating risks that might affect the project schedule or the quality of the software being developed and taking action to avoid these risks. The results of the risk analysis should be documented in the project plan along with an analysis of the consequences of a risk occurring. There are three categories of risk:

1. *Project risks*. These risks affect the entire project schedule.
2. *Product risks*. Product risks have influence on the quality and performance of the software.
3. *Business risks*. These risks affect the organization developing or procuring the software.

Project manager has to get ready for risks, understand the impact of these risks on the project, the product and the business, and take steps to avoid these risks. Project manager should prepare contingency plans so that, if the risks occur, he can take immediate actions.

The risk management process is shown in Figure 11.2. It has four stages:

1. *Risk identification*. Possible project, product and business risks are identified.
2. *Risk analysis*. Using risk analysis the probability and consequences of risks are analyzed.
3. *Risk planning*. The objective of risk planning is to avoid the risks or minimize its effects.
4. *Risk monitoring*. The identified risk is continuously assessed and the avoidance strategies are revised as more information becomes available about the risk.

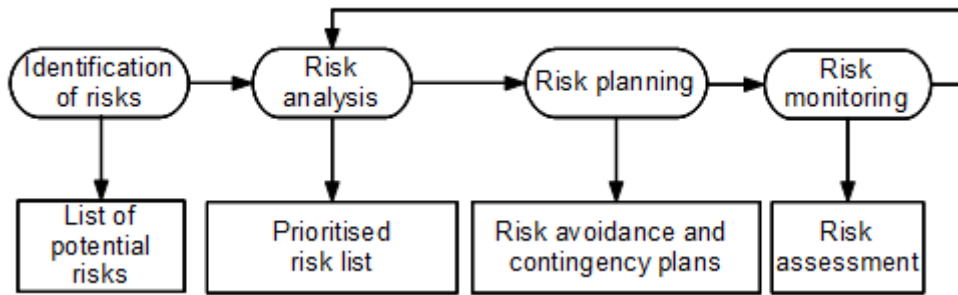


Figure 11.2. The risk management process.

The risk management process is an iterative process which continues throughout the project. As more information about the risks becomes available, the risks have to be reanalyzed and the avoidance and contingency plans has to be also modified. The outcomes of the risk management process should be documented in a risk management plan.

11.4.1 Risk identification

The first stage of risk management is the identification of risks. It is concerned with discovering possible risks to the project. Risk identification may be carried out as a team work based on the experience of team members. Helping the process risks are usually classified into separate groups:

1. *Technology risks.* These risks are related to the software or hardware technologies that are used to develop the system.
2. *People risks.* Risk that are associated with the people in the development team.
3. *Organizational risks.* Risk that derive from the changes in organizational environment.
4. *Tools risks.* Risk that derive from the CASE tools and other support software used in the development process.
5. *Requirements risks.* Risk that derive from changes to the customer requirements and the process of managing the requirements change.
6. *Estimation risks.* Risk that derive from the management estimates of the system characteristics and the resources required to build the system.

Table 11.1. gives some examples of possible risks in each of these categories.

Table 11.1. Risks and risk types.

Risk type	Possible risks
Technology	Purchased software component is defective.
People	Development manager regularly gets ill.
Organizational	The management of software company is restructured and other project management is responsible for the project.
Tools	The class skeleton generated by UML editor inefficient.
Requirements	Changes to software requirements which require development of additional increments.

Estimation	The time required to develop a software components is underestimated.
------------	---

11.4.2 Risk analysis

During the risk analysis process, the probability and the effects of identified risks are determined. This process is mainly based on the experience of project managers. The risk estimates are generally assigned to different bands:

1. The probability of the risk might be assessed as very low (<10%), low (10–25%), moderate (25-50%), high (50–75%) or very high (>75%).
2. The effects of the risk might be assessed as catastrophic, serious, tolerable or insignificant.

Table 11.2. illustrates results of risk analysis.

Table 11.2. Risk analysis.

Risk	Probability	Effects
It is not possible to recruit developers with appropriate skills	High	Catastrophic
One of the software developers who regularly gets a flu in every years gets a flu again in this year	High	Serious
The database used has a lower performance as expected.	Moderate	Serious
Development time for a software component is underestimated.	High	Serious

Once the risks have been analyzed and ranked, it is necessary to decide which risks are most significant. In general, the catastrophic risks and the serious risks having more than a moderate probability of occurrence should always be considered.

11.4.3 Risk planning

The risk planning process identifies strategies to manage the identified risks. There is no general procedure to establish risk management plans. It is mostly based on the judgement and experience of the project manager. Table 11.3. shows possible strategies that have been identified for the key risks from Table 11.2.

The risk management strategies have three categories:

1. *Avoidance strategies.* Following these strategies means that the probability that the risk will arise will be reduced.
2. *Minimization strategies.* The objective of these strategies is to reduce the effect of risk. An example of a risk minimization strategy is that for staff illness shown in Table 11.3.
3. *Contingency plans.* These are the strategies used when risk arises.

Table 11.3. Risk management strategies.

Risk	Strategy
Recruitment problems	Notify customer of potential difficulties, search for software components to purchase, looking for a sub-contractor company
Staff illness	Reorganize the development team so that more people can have the same competence
Database performance	Purchase of a higher performance database management system
Underestimated development time	Purchase of software components

11.4.4 Risk monitoring

The purpose of risk monitoring risk is to constantly assess each of the identified risks to decide whether its probability and effects have changed. Usually this cannot be observed directly and other factors should be monitored to detect the changes. Table 11.4. gives some examples of factors that may be helpful in assessing different types of risks.

Table 11.4. Risk factors.

Risk type	Potential indicators
Technology	One of the hardware supplier companies becomes bankrupt
People	Software engineer, who gets a flu in every year complains about weakness
Organizational	Poor managerial activity is seen
Tools	Using a CASE tool the first error occurs
Requirements	Customer complains about the performance of current version of software
Estimation	The status of the approved project schedule is very poor

11.5 Exercises

1. What are the main differences between a software engineering and an industrial manufacturing project?
2. List the main activities of a software project manager!
3. List the parts of a project plan!
4. Explain the project scheduling process!
5. Explain the steps of risk management!
6. What are the objectives for risk analysis?
7. What strategies can be applied in risk planning?

12 Software quality management

The quality of software has improved significantly over the past two decades. One reason for this is that companies have used new technologies in their software development process such as object-oriented development, CASE tools, etc. In addition, a growing importance of software quality management and the adoption of quality management techniques from manufacturing can be observed. However, software quality significantly differs from the concept of quality generally used in manufacturing mainly for the next reasons [1]:

1. The software specification should reflect the characteristics of the product that the customer wants. However, the development organization may also have requirements such as maintainability that are not included in the specification.
2. Certain software quality attributes such as maintainability, usability, reliability cannot be exactly specified and measured.
3. At the early stages of software process it is very difficult to define a complete software specification. Therefore, although software may conform to its specification, users don't meet their quality expectations.

Software quality management is split into three main activities:

1. *Quality assurance*. The development of a framework of organizational procedures and standards that lead to high quality software.
2. *Quality planning*. The selection of appropriate procedures and standards from this framework and adapt for a specific software project.
3. *Quality control*. Definition of processes ensuring that software development follows the quality procedures and standards.

Quality management provides an independent check on the software and software development process. It ensures that project deliverables are consistent with organizational standards and goals.

12.1 Process and product quality

It is general, that the quality of the development process directly affects the quality of delivered products. The quality of the product can be measured and the process is improved until the proper quality level is achieved. Figure 12.1. illustrates the process of quality assessment based on this approach.

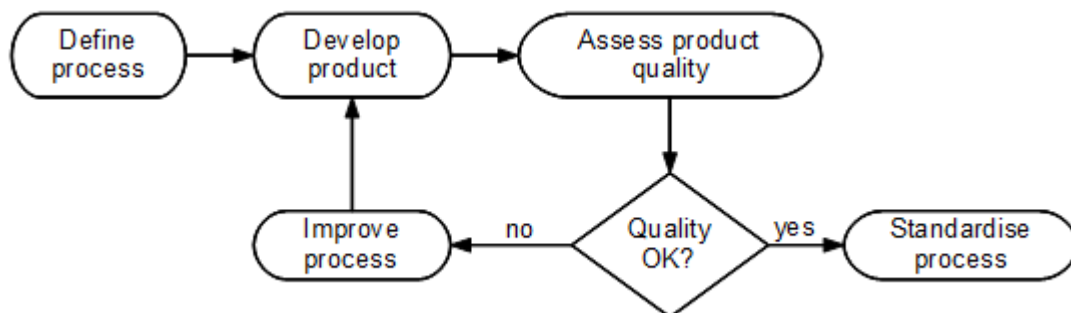


Figure 12.1. Process based quality assessment.

In manufacturing systems there is a clear relationship between production process and product quality. However, quality of software is highly influenced by the experience of software engineers. In addition, it is difficult to measure software quality attributes, such as maintainability, reliability, usability, etc., and to tell how process characteristics influence these attributes. However, experience has shown that process quality has a significant influence on the quality of the software.

Process quality management includes the following activities:

1. Defining process standards.
2. Monitoring the development process.
3. Reporting the software.

12.2 Quality assurance and standards

Quality assurance is the process of defining how software quality can be achieved and how the development organization knows that the software has the required level of quality. The main activity of the quality assurance process is the selection and definition of standards that are applied to the software development process or software product. There are two main types of standards. The product standards are applied to the software product, i.e. output of the software process. The process standards define the processes that should be followed during software development. The software standards are based on best practices and they provide a framework for implementing the quality assurance process.

The development of software engineering project standards is a difficult and time consuming process. National and international bodies such as ANSI and the IEEE develop standards that can be applied to software development projects. Organizational standards, developed by quality assurance teams, should be based on these national and international standards. Table 12.1. shows examples of product and process standards.

Table 12.1. Examples of product and process standards.

Product standards	Process standards
Requirements document structure	Project plan approval process
Method header format	Version release process
Java programming style	Change control process
Change request form	Test recording process

12.2.1 ISO

ISO 9000 is an international set of standards that can be used in the development of a quality management system in all industries. ISO 9000 standards can be applied to a range of organizations from manufacturing to service industries. ISO 9001 is the most general of these standards. It can be applied to organizations that design, develop and maintain products and develop their own quality processes. A supporting document (ISO 9000-3) interprets ISO 9001 for software development.

The ISO 9001 standard isn't specific to software development but includes general principles that can be applied to software development projects. The ISO 9001 standard describes various aspects of the quality process and defines the organizational standards and procedures that a company should define and follow during product development. These standards and procedures are documented in an organizational quality manual.

The ISO 9001 standard does not define the quality processes that should be used in the development process. Organizations can develop own quality processes and they can still be ISO 9000 compliant companies. The ISO 9000 standard only requires the definition of processes to be used in a company and it is not concerned with ensuring that these processes provide best practices and high quality of products. Therefore, the ISO 9000 certification doesn't mean exactly that the quality of the software produced by an ISO 9000 certified companies will be better than that software from an uncertified company.

12.2.2 Documentation standards

Documentation standards in a software project are important because documents can represent the software and the software process. Standardized documents have a consistent appearance, structure and quality, and should therefore be easier to read and understand. There are three types of documentation standards:

1. *Documentation process standards.* These standards define the process that should be followed for document production.
2. *Document standards.* These standards describe the structure and presentation of documents.
3. *Documents interchange standards.* These standards ensure that all electronic copies of documents are compatible.

12.3 Quality planning

Quality planning is the process of developing a quality plan for a project. The quality plan defines the quality requirements of software and describes how these are to be assessed. The quality plan selects those organizational standards that are appropriate to a particular product and development process. Quality plan has the following parts:

1. Introduction of product.
2. Product plans.
3. Process descriptions.
4. Quality goals.
5. Risks and risk management.

The quality plan defines the most important quality attributes for the software and includes a definition of the quality assessment process. Table 12.2. shows generally used software quality attributes that can be considered during the quality planning process.

Table 12.2. Software quality attributes.

Safety	Understandability	Portability
Security	Testability	Usability
Reliability	Adaptability	Reusability
Resilience	Modularity	Efficiency
Robustness	Complexity	Learnability
Maintainability		

12.4 Quality control

Quality control provides monitoring the software development process to ensure that quality assurance procedures and standards are being followed. The deliverables from the software development process are checked against the defined project standards in the quality control process. The quality of software project deliverables can be checked by regular quality reviews and/or automated software assessment. Quality reviews are performed by a group of people. They review the software and software process in order to check that the project standards have been followed and that software and documents conform to these standards.

Automated software assessment processes the software by a program that compares it to the standards applied to the development project.

12.4.1 Quality reviews

Quality reviews are the most widely used method of validating the quality of a process or product. They involve a group of people examining part or all of a software process, system, or its associated documentation to discover potential problems. The conclusions of the review are formally recorded and passed to the author for correcting the discovered problems. Table 12.3. describes several types of review, including quality reviews.

Table 12.3. Types of review.

Review type	Principal purpose
Design or program inspections	To detect detailed errors in the requirements, design or code.
Progress reviews	To provide information for management about the overall progress of the project.
Quality reviews	To carry out a technical analysis of product components or documentation to find mismatches between the specification and the component design, code or documentation and to ensure that defined quality standards of the organization have been followed.

12.5 Software measurement and metrics

Software measurement provides a numeric value for some quality attribute of a software product or a software process. Comparison of these numerical values to each other or to standards draws conclusions about the quality of software or software processes. Software product measurements can be used to make general predictions about a software system and identify anomalous software components.

Software metric is a measurement that relates to any quality attributes of the software system or process. It is often impossible to measure the external software quality attributes, such as maintainability, understandability, etc., directly. In such cases, the external attribute is related to some internal attribute assuming a relationship between them and the internal attribute is measured to predict the external software characteristic. Three conditions must be hold in this case:

1. The internal attribute must be measured accurately.
2. A relationship must exist between what we can measure and the external behavioural attribute.
3. This relationship has to be well understood, has been validated and can be expressed in terms of a mathematical formula.

12.5.1 The measurement process

A software measurement process as a part of the quality control process is shown in Figure 12.2. The steps of measurement process are the followings:

1. *Select measurements to be made.* Selection of measurements that are relevant to answer the questions to quality assessment.
2. *Select components to be assessed.* Selection of software components to be measured.

3. *Measure component characteristics.* The selected components are measured and the associated software metric values computed.
4. *Identify anomalous measurements.* If any metric exhibit high or low values it means that component has problems.
5. *Analyse anomalous components.* If anomalous values for particular metrics have been identified these components have to be examined to decide whether the anomalous metric values mean that the quality of the component is compromised.

Generally each of the components of the system is analyzed separately. Anomalous measurements identify components that may have quality problems.

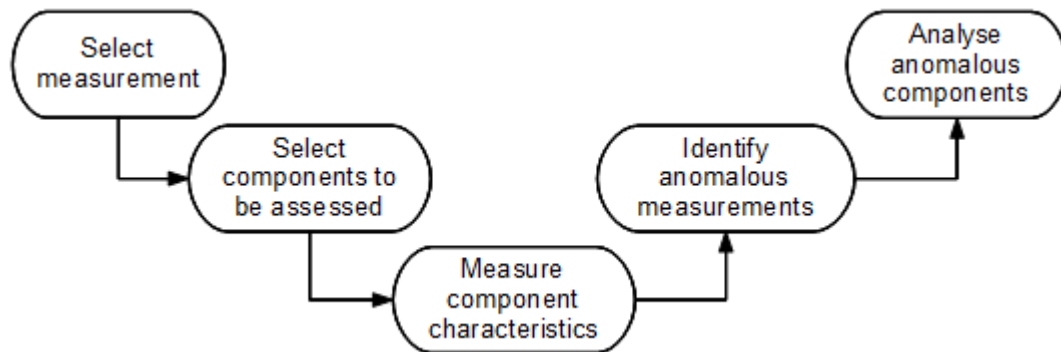


Figure 12.2. The software measurement process.

12.5.2 Product metrics

The software characteristics that can be easily measured such as size do not have a clear and consistent relationship with quality attributes such as understandability and maintainability. Product metrics has two classes:

1. *Dynamic metrics.* These metrics (for example execution time) are measured during the execution of a program.
2. *Static metrics.* Static metrics are based on measurements made of representations of the system such as the design, program or documentation.

Dynamic metrics can be related to the efficiency and the reliability of a program. Static metrics such as code size are related to software quality attributes such as complexity, understandability, maintainability, etc.

12.6 Exercises

1. Give examples of software quality attributes!
2. What is the objective of product standards?
3. What is the objective of process standards?
4. Give examples of product and process standards!
5. What international standards can be used in software quality assessment?
6. What are the types of document standard?
7. What are the main parts of a quality plan?
8. What are the steps of software measurement process?
9. Give examples of static software metric!

13 Software cost estimation

Software development processes are split into a number of separate activities. Cost estimation of software development project focuses on how associating estimates of effort and time with the project activities. Estimation involves answering the following questions [1]:

1. How much effort is required to complete each activity?
2. How much calendar time is needed to complete each activity?
3. What is the total cost of each activity?

Project cost estimation and project scheduling are usually carried out together. The costs of development are primarily the costs of the effort involved, so the effort computation is used in both the cost and the schedule estimate. The initial cost estimates may be used to establish a budget for the project and to set a price for the software for a customer. The total cost of a software development project is the sum of following costs:

1. Hardware and software costs including maintenance.
2. Travel and training costs.
3. Effort costs of paying software developers.

For most projects, the dominant cost is the effort cost. Effort costs are not just the salaries of the software engineers who are involved in the project. The following overhead costs are all part of the total effort cost:

1. Costs of heating and lighting offices.
2. Costs of support staff (accountants, administrators, system managers, cleaners, technicians etc.).
3. Costs of networking and communications.
4. Costs of central facilities (library, recreational facilities, etc.).
5. Costs of social security and employee benefits such as pensions and health insurance.

The aim of software costing is to accurately predict the cost of developing the software. The price of software is normally the sum of development cost and profit. During the development project managers should regularly update their cost and schedule estimates. This helps with the planning process and the effective use of resources. If actual expenditure is significantly greater than the estimates, then the project manager must take some action. This may involve applying for additional resources for the project or modifying the work to be done.

13.1 Software productivity estimation

Productivity estimates help to define the project cost and schedule. In a software development project managers may be faced with the problem of estimating the productivity of software engineers. For any software problem, there may be many different solutions, each of which has different attributes. One solution may execute more efficiently while another may be more readable and easier to maintain and comparing their production rates is very difficult.

Productivity estimates are usually based on measuring attributes of the software and dividing this by the total effort required for development. Software metrics have two main types:

1. *Size-related software metrics.* These metrics are related to the size of software. The most frequently used size-related metric is lines of delivered source code.
2. *Function-related software metrics.* These are related to the overall functionality of the software. For example, function points and object points are metrics of this type.

The productivity estimation defined as lines of source code per programmer month is widely used software productivity metric. It is computed by counting the total number of lines of source code divided by the total development time in programmer-months required to complete the project.

In other cases, the total number of function points in a program measures or estimates the following program features:

1. external inputs and outputs,
2. user interactions,
3. external interfaces,
4. files used by the system.

Obviously, some features are more complex than others and take longer time to code. The function point metric takes this into account by application of a weighting factor. The unadjusted function point count (UFC) is computed by multiplying the number of a given feature by its estimated weight for all features and finally summing products:

$$UFC = \sum (\text{number of a given feature}) \times (\text{weight for the given feature})$$

Object points are an alternative to function points. The number of object points is computed by the estimated weights of objects:

1. *Separate screens that are displayed.* Simple screens count as 1 object point, moderately complex screens count as 2, and very complex screens count as 3 object points.
2. *Reports that are produced.* For simple reports, count 2 object points, for moderately complex reports, count 5, and for reports that are likely to be difficult to produce, count 8 object points.
3. *Modules that are developed to supplement the database programming code.* Each of these modules counts as 10 object points.

The final code size is calculated by the number of function points and the estimated average number of lines of code (AVC) required to implement a function point. The estimated code size is computed as follows:

$$CODE\ SIZE = AVC \times UFC$$

The problem with measures based on the amount produced in a given time period is that they take no account of quality characteristics such as reliability and maintainability.

13.2 Development cost estimation techniques

In the early stage in a project it is very difficult to accurately estimate system development costs. There is no simple way to make an accurate estimate of the effort required to develop software. Techniques listed in Table 13.1. may be used to make software effort and cost estimates. All of these techniques based on the experience of project managers who use their knowledge of previous projects to estimate of the resources required for the project.

Table 13.1. Cost estimation techniques.

Technique	Description
Algorithmic cost modelling	Relating some software metric a mathematical model is developed to estimate the project cost.
Expert judgement	Several experts on the proposed software development techniques and the application domain are asked to estimate

	the project cost. The estimation process iterates until an agreed estimate is reached.
Estimation by previous projects	The cost of a new project is estimated by a completed project in the same application domain.
Application of Parkinson's Law	Parkinson's Law states that work expands to fill the time available and the cost is determined by the resources used.
Pricing to win	The software cost is estimated by the price what the customer has available to spend on the project.

Each estimation technique listed in Table 13.1. has its own strengths and weaknesses. It is recommended to use several cost estimation techniques and compare their results. If they predict significantly different costs more information is required about the product, the software process, development team, etc.

13.3 Algorithmic cost modelling

Algorithmic cost modelling uses a mathematical expression to predict project costs based on estimates of the project size, the number of software engineers, and other process and product factors. An algorithmic cost model can be developed by analyzing the costs and attributes of completed projects and finding the closest fit mathematical expression to actual project. In general, an algorithmic cost estimate for software cost can be expressed as:

$$EFFORT = A \times SIZE^B \times M$$

In this equation A is a constant factor that depends on local organizational practices and the type of software that is developed. Variable $SIZE$ may be either the code size or the functionality of software expressed in function or object points. M is a multiplier made by combining process, product and development attributes, such as the dependability requirements for the software and the experience of the development team. The exponential component B associated with the size estimate expresses the non-linearity of costs with project size. As the size of the software increases, extra costs are emerged. The value of exponent B usually lies between 1 and 1.5.

All algorithmic models have the same difficulties:

1. *It is difficult to estimate SIZE in the early stage of development.* Function or object point estimates can be produced easier than estimates of code size but are often inaccurate.
2. *The estimates of the factors contributing to B and M are subjective.* Estimates vary from one person to another person, depending on their background and experience with the type of system that is being developed.

The number of lines of source code in software is the basic software metric used in many algorithmic cost models. The code size can be estimated by previous projects, by converting function or object points to code size, by using a reference component to estimate the component size, etc. The programming language used for system development also affects the number of lines of code to be implemented. Furthermore, it may be possible to reuse codes from previous projects and the size estimate has to be adjusted to take this into account.

13.4 Exercises

1. What is the objective of software cost estimation?

2. What are the main costs of a software development project?
3. How programmer productivity can be measured by function points?
4. How programmer productivity can be measured by object points?
5. List some cost estimation techniques that is not based on any size related metric of software!
6. Explain the expression of algorithmic cost modelling!

14 References

1. Ian Sommerville. *Software Engineering, Eight Edition*. Addison-Wesley, 2007.
2. Kondorosi Károly, László Zoltán, Szirmay-Kalos László. *Objektum-Orientált Szoftverfejlesztés*, ComputerBooks, 1999.
3. Raffai Mária. *Egységesített megoldások a fejlesztésben, UML modellező nyelv, RUP módszertan*, Novadat Kiadó, 2001.
4. Sike Sándor és Varga László. *Szoftvertechnológia és UML*, ELTE Eötvös Kiadó. 2008.
5. Robert A. Maksimchuk és Eric J. Naiburg. *UML for Mere Mortals*, Addison-Wesley Professional, 2004.
6. Harald Störrle. *UML 2 Unified Modeling Language*, Panem Könyvkiadó Kft, 2007.
7. Raffai Mária. *UML 2 Modellező Nyelvi Kézikönyv*, Palatia Nyomda és Kiadó, 2007.
8. Erich Gamma, Ralph Johnson, Richard Helm, John Vlissides. *Programtervezési minták, Újrahasznosítható elemek objektumközpontú programokhoz*, Kiskapu Kft., 2004.
9. Object Management Group. *UML Resource Page*, <http://www.uml.org/>
10. Roger Oberg, Leslee Probasco, and Maria Ericsson. *Applying Requirements Management with Use Cases*, Rational Software White Paper, 2000.
11. RUP: Overview <http://sce.uhcl.edu/helm/RationalUnifiedProcess/indexRUP.htm>
12. *Rational Unified Process Best Practices for Software Development Teams*, Rational Software White Paper, <http://www.ibm.com>, 1998
13. Nyékyné Gaizler Edit. *Programozási nyelvek*, Kiskapu Kft., 2003.
14. Roger S. Pressman. *Software Engineering, A Practitioner's Approach*, McGraw Hill, 2005.
15. Robert L. Glass. *Facts and Fallacies of Software Engineering*, Addison-Wesley, 2003.
16. Beck K. *Extreme Programming Explained*. Addison-Wesley, 2000.
17. Vég Csaba. Rational Unified Process áttekintés, <http://www.logos2000.hu/docs/RUP.pdf>, 2001.
18. Dr. Sziray József, Kovács Katalin. *Az UML nyelv használata*, HEFOP 3.3.1-P.-2004-09-0102/1.0 pályázat, 2004.
19. Scrum (software development), Wikipedia: [http://en.wikipedia.org/wiki/Scrum_\(software_development\)](http://en.wikipedia.org/wiki/Scrum_(software_development))
20. Extreme programming, Wikipedia: http://en.wikipedia.org/wiki/Extreme_programming
21. Feature-driven development, Wikipedia: http://en.wikipedia.org/wiki/Feature-driven_development
22. Rational Unified Process, Wikipedia: http://en.wikipedia.org/wiki/Rational_Unified_Process
23. Agile software development, Wikipedia: http://en.wikipedia.org/wiki/Agile_software_development

