Software Engineering

Debre Tabor University

Faculty of Technology

Department of Computer Science

November, 2021

Module Name: - Software engineering

Course Code: CoSc3061

Course Credit: 3 Cr. Hrs. / 5 ECTS

Course Prerequisites: None

Writer: Gizatie Desalegn (MSc. in Software Eng.)

Editor: Wondwosen Haile (MSc. in Software Eng.)

1 Table of Contents

MODULE INTRODUCTION

This module helps the students to deal with complexity through modelling, by introducing the students to the software development steps: requirement gathering, analysis, design, implementation, testing and maintenance. It helps the students to understand several software methodologies and equips them with the skills to perform system Requirement analysis, Software specification, Design methods, Software testing, Software project management techniques; Software project planning, Risk management; Software Quality Assurance; Software reuse; and Computer aided software engineering: with both the structured as well as OO approach. This particular module covers O-O concepts, tools, development life cycle, problem solving, modeling, analysis, and design, while utilizing UML (Unified Modeling Language) for O-O modeling. UML has become the standard notation for modeling O-O systems and is being embraced by major software developers like Microsoft and Oracle.

MODULE OBJECTIVES

The objective of this module is to provide a thorough discussion of the Object-Oriented Software Engineering issues. Its purpose is to present, as clearly and briefly as possible, theability to elicit, analyse, specify, validate and produce complete and consistent requirement document and manage requirement changes, and describe the object-oriented software development process, including object-oriented methodologies and workflows, understand the issues involved in software project management and manage software projects. Throughout the discussion, aspects of the system are viewed from the points of view of the basic principles of software engineering, to build many different models of a system and of the application domain, identify system development steps or problem solving steps: analysis, design, implementation, testing; describe the methods to deal with system development steps for developing applications, identify the different problems related to software development and techniques of handling them.

On successful completion of this module the learner will be able to

- ✓ Describe in detail the theory, concepts and methods pertaining to the Unified Modeling Language (UML).
- ✓ Create requirements using use case modeling concepts.
- ✓ Demonstrate conceptual and technical skills in the analysis, design and implementation of a software system using Object Oriented Concepts.

- ✓ Employ tools and techniques for Object Oriented Software Engineering,
- ✓ Demonstrate an ability to adapt and solve problems in software development activities from specification to testing individually and as part of a team.

ORGANIZATION OF THE MODULE

The module is organized into six chapters:

Chapter One: Provides an introduction of Object Oriented Software Engineering and looks at the Two Orthogonal view of software, software development process models like software process, software life cycle and process model, process activities, process assessment models, and software process metrics, Object oriented system development methodology, overview of unified approach, object oriented philosophy and basic concepts of objects, attributes, states and properties.

Chapter Two: Examines an overview of UML, Where Can the UML Be Used? Building Blocks of the UML, Relationships in the UML, Diagrams in the UML, Use Case Diagrams, Class Diagrams, Sequence diagram, State chart diagrams, Activity diagrams, Component diagram, Deployment diagram, Diagram extensions.

Chapter Three: Discusses about Requirements elicitation concepts, Functional requirements, Nonfunctional and pseudo requirements, Levels of description, Correctness, completeness, consistency, clarity, and realism, Verifiability and traceability, Requirements elicitation activities, Identifying actors, Identifying scenarios, Identifying use cases, Refining use cases, Identifying relationships among actors and use cases, Identifying initial analysis objects, Identifying nonfunctional requirements, Managing requirements elicitation, Eliciting information from users, Validating requirements: Usability testing, Documenting requirements elicitation.

Chapter Four: Discusses about a software project management, responsibility of software project managers, project planning, the organization of SPMP document, project size estimation metrics, project estimation technique, scheduling, organization and team structures, risk management and quality assurance monitoring plans.

Chapter Five: Discusses about an Overview of Analysis, Analysis Concepts, Entity, Boundary, and Control Objects, Association Multiplicity Revisited, Qualified Associations, Generalization, Analysis Activities: From Use Cases to Objects, Identifying Entity Objects, Identifying Boundary Objects, Identifying Control Objects, Modeling Interactions between Objects: Sequence Diagrams, Identifying Associations, Identifying Attributes, Reviewing the Analysis Model.

Chapter Six: Deals with system design, System design concepts, System design activities: From objects to subsystems, Documenting system design, An overview of object design, Object design concepts, Object design activities, Managing object design, Documenting object design.

Chapter Seven: Examines the overview of testing, Testing concepts, Testing activities, Managing testing, and impact of object oriented testing.

Chapter 1: Introduction

General Objective

The objective of this chapter is to introduce you to the idea of a software process—a coherent set of activities for software production.

Specific Objectives:

- Understand the concepts of software processes and software process models;
- Have been introduced to three generic software process models and when they might be used;
- Know about the fundamental process activities of software requirements engineering, software development, testing, and evolution;
- Understand why processes should be organized to cope with changes in the software requirements and design;
- Understand how the Rational Unified Process integrates good software engineering practice to create adaptable software processes.

1.1. Two Orthogonal view of software.

A *software development methodology* is a series of processes leads to the development of an application. The software processes describe how the work is to be carried out to achieve the original goal based on the system requirements. The software development process will continue to exist as long as the development system is in operation.

Object-oriented systems development methods differ from traditional development techniques in that the traditional techniques view software as a collection of programs (or functions) and isolated data. A program can be defined as : *Algorithms + Data Structures = Programs:*

"A software system is a set of mechanisms for performing certain action on certain data."

The main distinction between traditional system development methodologies and newer object-oriented methodologies depends on their primary focus

- Traditional approach
    - Focuses on the functions of the system
- Object-oriented systems development
    - Centers on the object, which combines data and functionality.

Table 1.1. : Comparison of Traditional Approach vs Object oriented system

TRADITIONAL APPROACH	OBJECT ORIENTED SYSTEM DEVELOPMENT
Collection of procedures(functions)	Combination of data and functionality
Focuses on function and procedures, different styles and methodologies for each	Focuses on object, classes, modules that can be easily replaced, modified and reused.
Moving from one phase to another phase is complex.	Moving from one phase to another phase is easier.
Increases duration of project	decreases duration of project
Increases complexity	Reduces complexity and redundancy

1.2. Software development process models

1.2.1. Software Process

A software process is a set of related activities that leads to the production of a software product. These activities may involve the development of software from scratch in a standard programming language like Java or C. However, business applications are not necessarily developed in this way. New business software is now often developed by extending and modifying existing systems or by configuring and integrating off-the-shelf software or system components.

When you build a product or system, it‘s important to go through a series of predictable steps—a road map that helps you create timely, high-quality result. The road map that you follow is called a _software process. The process provides interaction between users and designers, between users and evolving tools, and between designers and evolving tools [technology].

Who does it? Software engineers and their managers adapt the process to their needs and then follow it. In addition, the people who have requested the software play a role in the software process.

Why is it important? Because it provides stability, control, and organization to an activity that can, if left uncontrolled, become quite chaotic. Therefore, a software process is a set of related activities that leads to the production of a quality and effective software product.

There are many different software processes but all must include four activities that are fundamental to software engineering:

1. Software specification: The functionality of the software and constraints on its operation must be defined.

2. Software design and implementation: The software to meet the specification must be produced.

3. Software validation: The software must be validated to ensure that it does what the customer wants.

4. Software evolution: The software must evolve to meet changing customer needs.

In these activities, there are sub-activities such as requirements validation, architectural design and unit testing, etc. There are also supporting process activities such as documentation and software configuration management.

When we describe and discuss processes, we usually talk about the activities in these processes such as specifying a data model, designing a user interface, etc., and the ordering of these activities. However, as well as activities, process descriptions may also include:

1. Products, which are the outcomes of a process activity. For example, the outcome of the activity of architectural design may be a model of the software architecture.

2. Roles, which reflect the responsibilities of the people involved in the process. Examples of roles are project manager, configuration manager, programmer, analyst, etc.

3. pre-and post-conditions, which are statements that are true before and after a process activity has been enacted or a product produced. For example, before architectural design begins, a pre-condition may be that all requirements have been approved by the customer; after this activity is finished, a post-condition might be that the UML models describing the architecture have been reviewed.

Sometimes, software processes are categorized as either plan-driven or agile processes. Plan-driven processes are processes where all of the process activities are planned in advance and progress is measured against this plan. In agile processes planning is incremental and it is easier to change the process to reflect changing customer requirements. There is no hard and fast rules to follow but each approach is suitable for different types of software. Generally, you need to find a balance between plan-driven and agile processes.

1.2.2. Software life cycle and process models

1.2.2.1. Software Process model

A software process model is a simplified and abstract representation of a software process. Each process model represents a process from a particular perspective, and thus provides only partial information about that process. For example, a process activity model shows the activities and their sequence but may not show the roles of the people involved in these activities. In this section, you will be introduced a number of very general process models (sometimes called _process paradigms‘) from an architectural perspective. That is, we see the framework of the process but not the details of specific activities.

These generic models are not definitive descriptions of software processes. Rather, they are abstractions of the process that can be used to explain different approaches to software development. You can think of them as process frameworks that may be extended and adapted to create more specific software engineering processes.

The process models that we cover here are:

1. The waterfall model: This takes the fundamental process activities of specification, development, validation, and evolution and represents them as separate process phases such as requirements specification, software design, implementation, testing, and so on.
2. Incremental development: This approach interleaves the activities of specification, development, and validation. The system is developed as a series of versions (increments), with each version adding functionality to the previous version.
3. Reuse-oriented software engineering: This approach is 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 from scratch.

1. Water fall model

The waterfall life-cycle model was first put forward by Royce [1970]. Figure 2.2 shows the feedback loops for maintenance while the product is being developed. Figure 2.9 also shows the feedback loops for postdelivery maintenance.

A critical point regarding the waterfall model is that no phase is complete until the documentation for that phase has been completed and the products of that phase have been approved by the software quality assurance (SQA) group. This carries over into modifications; if the products of

an earlier phase have to be changed as a consequence of following a feedback loop, that earlier phase is deemed to be complete only when the documentation for the phase has been modified and the modifications have been checked by the SQA group.

Why called Waterfall?

Consider Tis Abay falls (Blue Nile falls in Lake Tana at Bahir Dar). Once the water on the cliff of the mountain has flowed over the edge of the cliff and has begun its journey down the side of the mountain, it cannot turn back. It is the same with waterfall development. Once a phase of development is completed, the development proceeds to the next phase and there is no turning back.

Because of the cascade from one phase to another, this model is known as the ‗waterfall model' or software life cycle. The waterfall model is an example of a plan-driven process—in principle, you must plan and schedule all of the process activities before starting work on them. The unmodified ―waterfall model‖ progress flows from the top to the bottom, like a cascading waterfall. It's a non-iterative (or sequential) process in which progress is seen as flowing steadily downwards (like a waterfall) through the development phases. Thus the waterfall model maintains that one should move to a phase only when its preceding phase is reviewed and verified. That is, each phase must be completed before the next phase can begin. In waterfall model phases do not overlap. But there are various modified waterfall models, and hence, may include slight or major modifications. These modifications include returning to the previous cycle after flaws are found downstream, or returning all the way to the design phase if downstream phases deem insufficient.

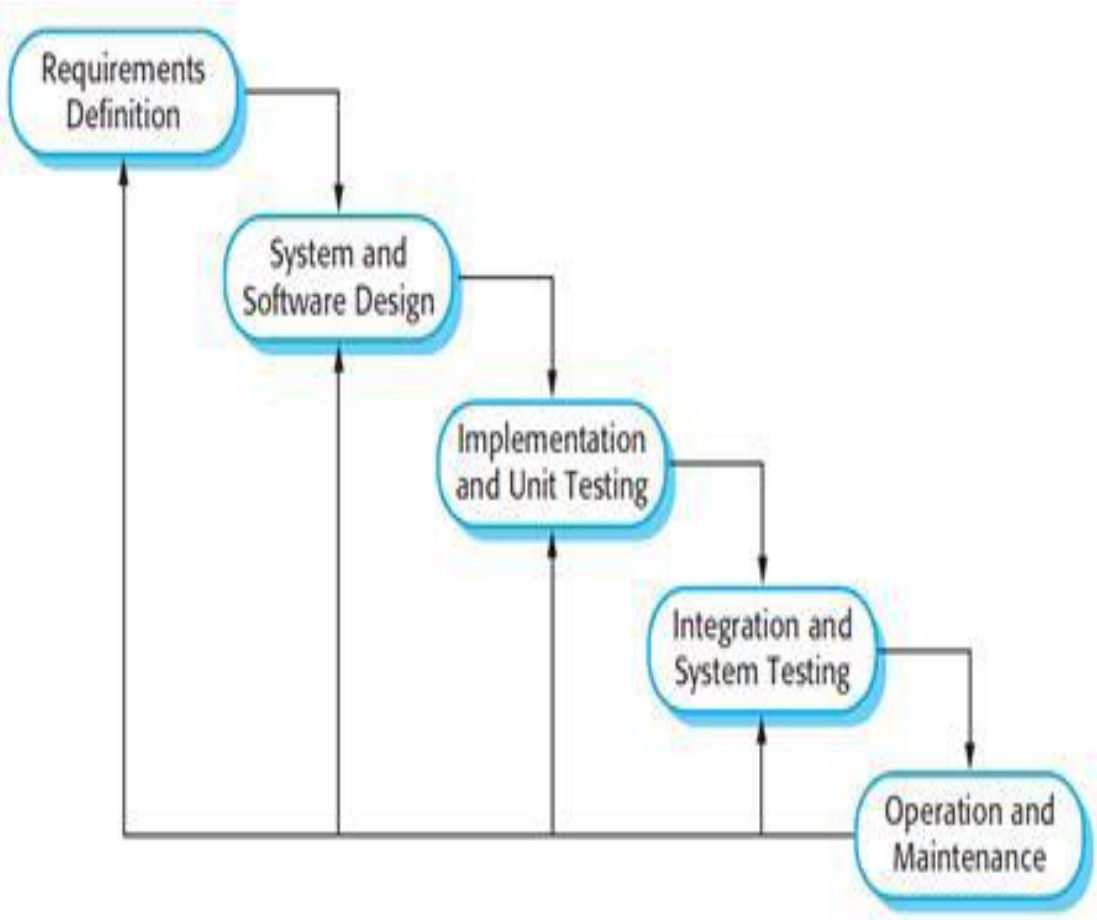


Figure 1.1 The waterfall model

These models are, for large systems development, used together, by harvesting the best features amongst them. They are not mutually exclusive and it makes sense to combine some of the best features of the waterfall and the incremental development models.

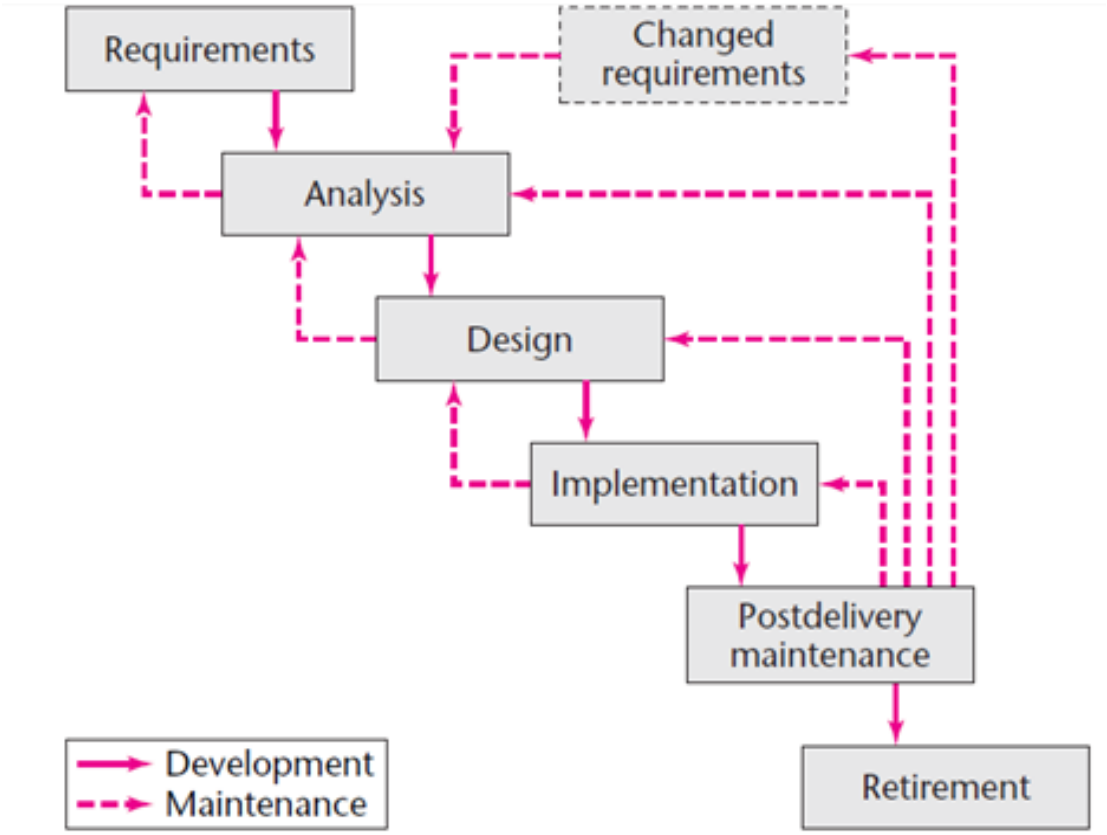


Figure 1.2 Waterfall Life Cycle

The principal stages of the waterfall model directly reflect the fundamental development activities:

1. Requirement's analysis and definition: The system 's services, constraints, and goals are established by consultation with system users. They are then defined in detail and serve as a system specification. The aim of the requirements analysis and specification phase is to understand the exact requirements of the customer and to document them properly. This phase consists of two distinct activities, namely

a. Requirements gathering and analysis b. Requirement's specification

The goal of the requirement's gathering activity is to collect all relevant information from the customer regarding the product to be developed. This is done to clearly understand the customer requirements so that incompleteness and inconsistencies are removed. The requirements analysis activity is begun by collecting all relevant data regarding the product to be developed from the users of the product and from the customer through interviews and discussions. For example, to perform the requirements analysis of a business accounting software required by an organization, the analyst might interview all the accountants of the organization to ascertain their requirements.

The data collected from such a group of users usually contain several contradictions and ambiguities, since each user typically has only a partial and incomplete view of the system. Therefore, it is necessary to identify all ambiguities and contradictions in the requirements and resolve them through further discussions with the customer. After all ambiguities, inconsistencies, and incompleteness have been resolved and all the requirements properly understood, the requirements specification activity can start. During this activity, the user requirements are systematically organized into a Software Requirements Specification (SRS) document. The customer requirements identified during the requirements gathering and analysis activity are organized into a SRS document. The important components of this document are functional requirements, the nonfunctional requirements, and the goals of implementation.

System and software design: The systems design process allocates the requirements to either hardware or software systems by establishing an overall system architecture.

Software design involves identifying and describing the fundamental software system abstractions and their relationships. The goal of the design phase is to transform the requirements specified in the SRS document into a structure that is suitable for implementation in some programming language. In technical terms, during the design phase the software architecture is derived from the SRS document. Two distinctly different approaches are available: the traditional design approach and the object-oriented design approach.

a. Traditional design approach -Traditional design consists of two different activities; first a structured analysis of the requirements specification is carried out where the detailed structure of the problem is examined. This is followed by a structured design activity. During structured design, the results of structured analysis are transformed into the software design.

b. Object-oriented design approach -In this technique, various objects that occur in the problem domain and the solution domain are first identified, and the different relationships that exist among these objects are identified. The object structure is further refined to obtain the detailed design.

3. Implementation and unit testing: During this stage, the software design is realized as a set of programs or program units. Unit testing involves verifying that each unit meets its specification. The purpose of the coding phase (sometimes called the implementation phase) of software development is to translate the software design into source code. Each component of the design is implemented as a program module. The end-product of this phase is a set of program modules that have been individually tested. During this phase, each module is unit tested to determine the correct

working of all the individual modules. It involves testing each module in isolation as this is the most efficient way to debug the errors identified at this stage.

4. Integration and system testing: The individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met. After testing, the software system is delivered to the customer. Integration of different modules is undertaken once they have been coded and unit tested. During the integration and system testing phase, the modules are integrated in a planned manner. The different modules making up a software product are almost never integrated in one shot. Integration is normally carried out incrementally over a number of steps. During each integration step, the partially integrated system is tested and a set of previously planned modules are added to it. Finally, when all the modules have been successfully integrated and tested, system testing is carried out. The goal of system testing is to ensure that the developed system conforms to its requirements laid out in the SRS document. System testing usually consists of three different kinds of testing activities:

a. α – testing: It is the system testing performed by the development team.

b. β –testing: It is the system testing performed by a friendly set of customers.

c. Acceptance testing: It is the system testing performed by the customer himself after the product delivery to determine whether to accept or reject the delivered product.

System testing is normally carried out in a planned manner according to the system test plan document. The system test plan identifies all testing-related activities that must be performed, specifies the schedule of testing, and allocates resources. It also lists all the test cases and the expected outputs for each test case.

5. Operation and maintenance: Normally (although not necessarily), this is the longest life cycle phase. Maintenance of a typical software product requires much more than the effort necessary to develop the product itself. The system is installed 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 enhancing the system ‘s services as new requirements are discovered.

In principle, the result of each phase is one or more documents that are approved (_signed off‘). The following phase should 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 feedback from one phase to another. Documents produced in each phase may then have to be modified to reflect the changes made.

Because of the costs of producing and approving documents, iterations can be costly and involve significant rework. Therefore, after a small number of iterations, it is normal to freeze parts of the development, such as the specification, and to continue with the later development stages. Problems are left for later resolution, ignored, or programmed around. This premature freezing of requirements may mean that the system won‘t do what the user wants. It may also lead to badly structured systems as design problems are circumvented by implementation tricks.

During the final life cycle phase (operation and maintenance) the software is put into use. Errors and omissions in the original software requirements are discovered. Program and design errors emerge and the need for new functionality is identified. The system must therefore evolve to remain useful. Making these changes (software maintenance) may involve repeating previous process stages.

The waterfall model is consistent with other engineering process models and documentation is produced at each phase. This makes the process visible so managers can monitor progress against the development plan. Its major problem is the inflexible partitioning of the project into distinct stages. Commitments must be made at an early stage in the process, which makes it difficult to respond to changing customer requirements.

In principle, the waterfall model should only be used when the requirements are well understood and unlikely to change radically during system development. However, the waterfall model reflects the type of process used in other engineering projects. As is easier to use a common management model for the whole project, software processes based on the waterfall model are still commonly used.

Advantages of waterfall model:

- This model is simple and easy to understand and use. The waterfall model provides a structured approach; the model itself progresses linearly through discrete, easily understandable and explainable phases and thus is easy to understand.
- It is easy manage due to the rigidity of the model – each phase has specific deliverables and a review process.
- In this model phases are processed and completed one at a time. Phases do not overlap.
- Waterfall model works well for smaller projects where requirements are very well understood.

Disadvantages of waterfall model:

- Once an application is in the testing stage, it is very difficult to go back and change something that was not well-thought in the first stage.
- No working software is produced until late during the life cycle. □□High amounts of risk and uncertainty
- Poor model for long and ongoing projects.
- Not suitable for projects where requirements are at a moderate to high risk of changing.

2. Incremental Development

Incremental development is based on the idea of developing an initial implementation, exposing this to user comment and evolving it through several versions until an adequate system has been developed (Figure 2.2).

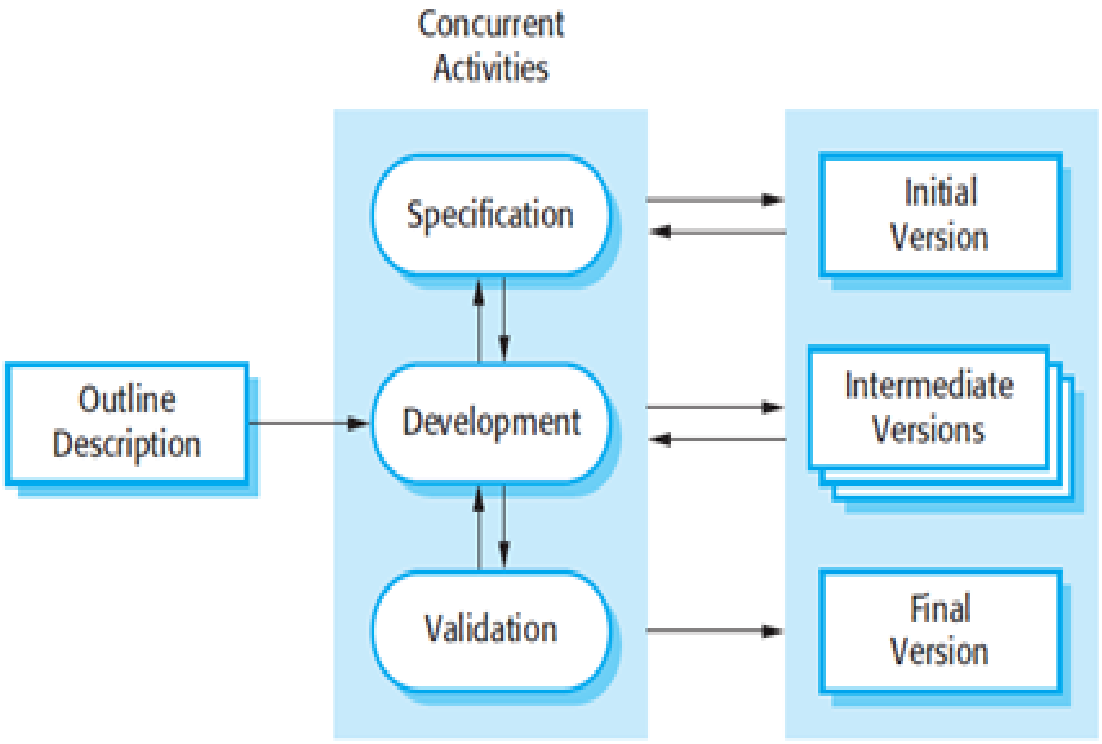


Figure 1.3 Incremental Development

Specification, development, and validation activities are interleaved rather than separate, with rapid feedback across activities.

For example, word-processing software developed using the incremental paradigm might deliver basic file management, editing, and document production functions in the first increment; more sophisticated editing and document production capabilities in the second increment; spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment.

This model delivers software in small but usable pieces, called ―increments. In general, each increment builds on those that have already been delivered. The model is iterative in nature and focuses on the delivery of an operational product with each increment. Early increments are stripped down versions of the final product, but they do provide capability that serves the user and also provide a platform for evaluation by the user.

Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project. Early increments can be implemented with fewer people. If the core product is well received, then additional staff (if required) can be added to implement the next increment. In addition, increments can be planned to manage technical risks. For example, a major system might require the availability of new hardware that is under development and whose delivery date is uncertain. It might be possible to plan early increments in a way that avoids the use of this hardware, thereby enabling partial functionality to be delivered to end-users without inordinate delay.

In incremental model the whole requirement is divided into various builds. Multiple development cycles take place here, making the life cycle a ―multi-waterfall‖ cycle. Cycles are divided up into smaller, more easily managed modules.

Advantages of Incremental Model:

- Generates working software quickly and early during the software life cycle. □□This model is more flexible – less costly to change scope and requirements. □□It is easier to test and debug during a smaller iteration.
- In this model customer can respond to each built. □□Lowers initial delivery cost.
- Easier to manage risk because risky pieces are identified and handled during its iteration.

Disadvantages of Incremental Model:

- Needs good planning and design
- Needs a clear and complete definition of the whole system before it can be broken down and built incrementally.

- Total cost is higher than waterfall.

3. Reuse-oriented software engineering

In the majority of software projects, there is some software reuse. This often happens informally when people working on the project know of designs or code that are similar to what is required. They look for these, modify them as needed, and incorporate them into their system.

This informal reuse takes place irrespective of the development process that is used. However, in the 21st century, software development processes that focus on the reuse of existing software have become widely used. Reuse-oriented approaches rely on a large base of reusable software components and an integrating framework for the composition of these components. Sometimes, these components are systems in their own right (COTS or commercial off-the-shelf systems) that may provide specific functionality such as word processing or a spreadsheet.

A general process model for reuse-based development is shown in Figure 2.3.

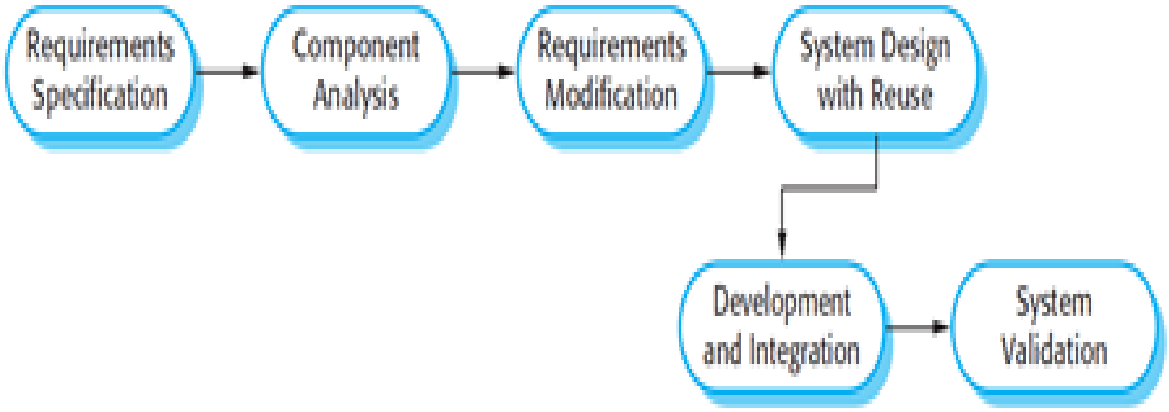


Figure 1.4 Reuse-oriented software engineering

Although the initial requirements specification stage and the validation stage are comparable with other software processes, the intermediate stages in a reuse-oriented process are different. These stages are:

1. Component analysis: Given the requirements specification, a search is made for components to implement that specification. Usually, there is no exact match and the components that may be used only provide some of the functionality required.

2. Requirement's modification: During this stage, the requirements are analyzed using information about the components that have been discovered. They are then modified to reflect the available components. Where modifications are impossible, the component analysis activity may be re-entered to search for alternative solutions.

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 and organize the framework to cater for this. 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 COTS systems are integrated to create the new system. System integration, in this model, may be part of the development process rather than a separate activity.

There are three types of software component that may be used in a reuse-oriented process:

1. Web services that are developed according to service standards and which are available for remote invocation.

2. Collections of objects that are developed as a package to be integrated with a component framework such as .NET or J2EE.

3. Stand-alone software systems that are configured for use in a particular environment. Advantages:

1. Reduced cost and risks

2. Faster delivery of the software (Accelerated development)

Disadvantages:

1. Compromising requirements

2. Some control over the system evolution is lost as new versions of the reusable components are not under the control of the organization using them.

3. Increased maintenance costs

4. Not-invented-here syndrome

5. Lack of tool support

1.2.3. Process Activities

Real software processes are interleaved sequences of technical, collaborative, and managerial activities with the overall goal of specifying, designing, implementing, and testing a software system. The four basic process activities of specification, development, validation, and evolution

are organized differently in different development processes. In the waterfall model, they are organized in sequence, whereas in incremental development they are interleaved. How these activities are carried out depends on the type of software, people, and organizational structures involved

1.2.3.1. Software Specification

Software specification or requirements engineering is the process of understanding and defining what services are required from the system and identifying the constraints on the system 's operation and development. The requirements engineering process (Figure 2.4) aims to produce an agreed requirements document that specifies a system satisfying stakeholder requirements.

Requirements are usually presented at two levels of detail. End-users and customers need a high-level statement of the requirements; system developers need a more detailed system specification.

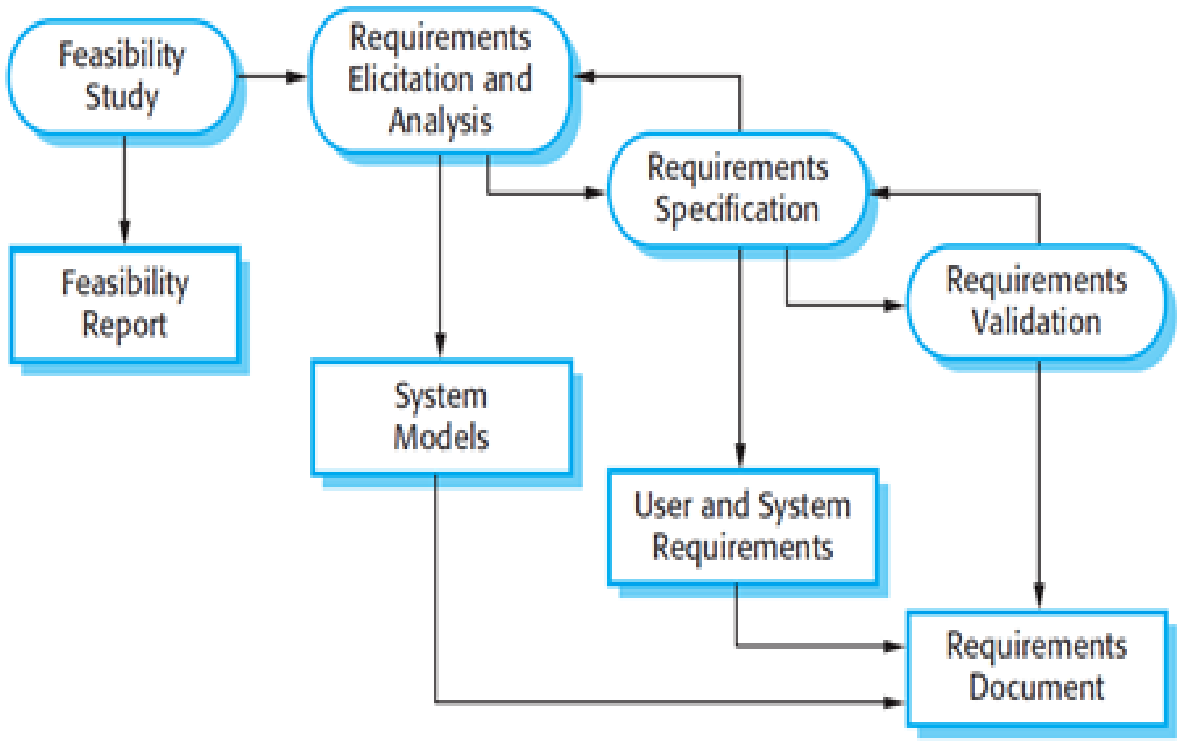


Figure 1.5 The Requirements engineering process

There are four main activities in the requirements engineering process:

1. Feasibility study: Feasibility shows viability and practicality of a software, project or idea. 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 if it can be developed within existing budgetary constraints. A feasibility study should be relatively cheap and quick. The result should inform the decision of whether or not to go ahead with a more detailed analysis.

2. Requirement's elicitation and analysis: This is the process of deriving the system requirements through observation of existing systems, discussions with potential users and procurers, task analysis, and so on. This may involve the development of one or more system models and prototypes. These help you understand the system to be specified.
3. *Requirement's specification:* Requirements specification is the activity of translating the information gathered during the analysis activity in to a document that defines a set of requirements. Two types of requirements may be included in this document. User requirements are abstract statements of the system requirements for the customer and end-user of the system; system requirements are a more detailed description of the functionality to be provided.
4. *Requirement's validation:* This activity checks the requirements for realism, consistency, and completeness. During this process, errors in the requirements document are inevitably discovered. It must then be modified to correct these problems.

1.2.3.2. Software Design and Implementation

The implementation stage of software development is the process of converting a system specification into an executable system. It always involves processes of software design and programming but, if an incremental approach to development is used, may also involve refinement of the software specification.

A software design is a description of the structure of the software to be implemented, the data models and structures used by the system, the interfaces between system components and, sometimes, the algorithms used. Designers do not arrive at a finished design immediately but develop the design iteratively. They add formality and detail as they develop their design with constant backtracking to correct earlier designs.

The activities in the design process vary, depending on the type of system being developed. For example, real-time systems require timing design but may not include a database so there is no database design involved. There are four activities that may be part of the design process for information systems:

1. Architectural design, where you identify the overall structure of the system, the principal components (sometimes called sub-systems or modules), their relationships, and how they are distributed.
2. Interface design, where you define the interfaces between system components. This interface specification must be unambiguous. With a precise interface, a component can be used without other components having to know how it is implemented. Once interface specifications are agreed, the components can be designed and developed concurrently.
3. Component design, where you take each system component and design how it will operate. This may be a simple statement of the expected functionality to be implemented, with the specific design left to the programmer. Alternatively, it may be a list of changes to be made to a reusable component or a detailed design model. The design model may be used to automatically generate an implementation.
4. *Database design*, where you design the system data structures and how these are to be represented in a database. Again, the work here depends on whether an existing database is to be reused or a new database is to be created.

1.2.3.3. Software Validation

Software validation or, more generally, verification and validation (V&V) is intended to show that a system both conforms to its specification and that it meets the expectations of the system customer. Program testing, where the system is executed using simulated test data, is the principal validation technique. Validation may also involve checking processes, such as inspections and reviews, at each stage of the software process from user requirements definition to program development. Because of the predominance of testing, the majority of validation costs are incurred during and after implementation.

1.2.3.4. Software Evolution

The flexibility of software systems is one of the main reasons why more and more software is being incorporated in large, complex systems. Once a decision has been made to manufacture hardware, it is very expensive to make changes to the hardware design. However, changes can be made to software at any time during or after the system development. Even extensive changes are still much cheaper than corresponding changes to system hardware.
Historically, there has always been a split between the process of software development and the process of software evolution (software maintenance). People think of software development as a

creative activity in which a software system is developed from an initial concept through to a working system. However, they sometimes think of software maintenance as dull and uninteresting. Although the costs of maintenance are often several times the initial development costs, maintenance processes are sometimes considered to be less challenging than original software development.

1.2.4. Process assessment models

The only rational way to improve any process is to measure specific attributes of the process, develop a set of meaningful metrics based on these attributes, and then use the metrics to provide indicators that will lead to a strategy for improvement.

We measure the efficacy of a software process indirectly. That is, we derive a set of metrics based on the outcomes that can be derived from the process. Outcomes include measures of errors uncovered before release of the software, defects delivered to and reported by end-users, work products delivered (productivity), human effort expended, calendar time expended, schedule conformance, and other measures. We also derive process metrics by measuring the characteristics of specific software engineering tasks.

Measurement enables managers and practitioners to improve the software process; assist in the planning, tracking, and control of a software project; and assess the quality of the product (software) that is produced. Measures of specific attributes of the process, project, and product are used to compute software metrics. These metrics can be analyzed to provide indicators that guide management and technical actions.

Process metrics enable an organization to take a strategic view by providing insight into the effectiveness of a software process. Project metrics are tactical. They enable a project manager to adapt project work flow and technical approach in a real-time manner.

Both size- and function-oriented metrics are used throughout the industry. Size oriented metrics use the line of code as a normalizing factor for other measures such as person-months or defects. The function point is derived from measures of the information domain and a subjective assessment of problem complexity.

Software quality metrics, like productivity metrics, focus on the process, the project, and the product. By developing and analyzing a metrics baseline for quality, an organization can correct those areas of the software process that are the cause of software defects.

Metrics are meaningful only if they have been examined for statistical validity. The control chart is a simple method for accomplishing this and at the same time examining the variation and location of metrics results.

Measurement results in cultural change. Data collection, metrics computation, and metrics analysis are the three steps that must be implemented to begin a metrics program. In general, a goal-driven approach helps an organization focus on the right metrics for its business. By creating a metrics baseline—a database containing process and product measurements—software engineers and their managers can gain better insight into the work that they do and the product that they produce.

1.2.5. Software process metrics

Software process metrics are used for strategic purposes. Software project measures are tactical. That is, project metrics and the indicators derived from them are used by a project manager and a software team to adapt project work flow and technical activities.

The first application of project metrics on most software projects occurs during estimation. Metrics collected from past projects are used as a basis from which effort and time estimates are made for current software work. As a project proceeds, measures of effort and calendar time expended are compared to original estimates (and the project schedule). The project manager uses these data to monitor and control progress.

As technical work commences, other project metrics begin to have significance. Production rates represented in terms of pages of documentation, review hours, function points, and delivered source lines are measured. In addition, errors uncovered during each software engineering task are tracked. As the software evolves from specification into design, technical metrics are collected to assess design quality and to provide indicators that will influence the approach taken to code generation and testing.

The intent of project metrics is twofold. First, these metrics are used to minimize the development schedule by making the adjustments necessary to avoid delays and mitigate potential problems and risks. Second, project metrics are used to assess product quality on an ongoing basis and, when necessary, modify the technical approach to improve quality.

As quality improves, defects are minimized, and as the defect count goes down, the amount of rework required during the project is also reduced. This leads to a reduction in overall project cost.

Another model of software project metrics as many scholars suggest that every project should

measure:

• Inputs: measures of the resources (e.g., people, environment) required to do the work.

• Outputs: measures of the deliverables or work products created during the software engineering process.

• Results: measures that indicate the effectiveness of the deliverables.

The software process and the product it produces both are influenced by many parameters (e.g., the skill level of practitioners, the structure of the software team, the knowledge of the customer, the technology that is to be implemented, the tools to be used in the development activity), metrics collected for one project or product will not be the same as similar metrics collected for another project. In fact, there is often significant variability in the metrics we collect as part of the software process.

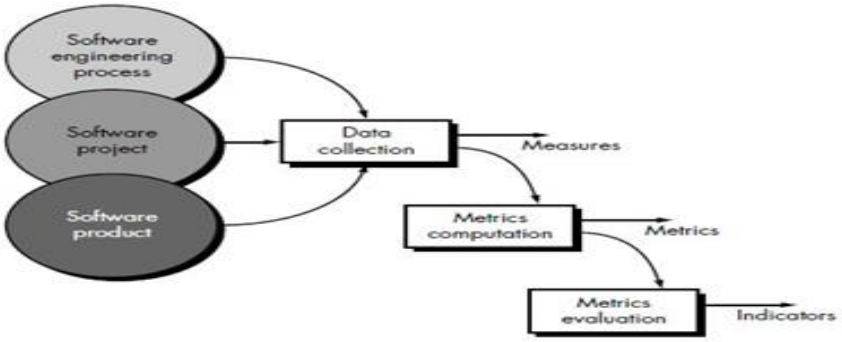


Figure 1.6 Software metrics collection process

1.3. Object oriented system development methodology.

Object-oriented development offers a different model from the traditional software development approach, which is based on functions and procedures. In simplified terms, object-oriented systems development is a way to develop software by building self-contained modules or objects that can be easily replaced, modified, and reused.

In an object-oriented environment,

- Software is a collection of discrete objects that encapsulate their data as well as the functionality to model real-world "objects."
- An object orientation yields important benefits to the practice of software construction
- Each object has attributes (data) and methods (functions).

- Objects are grouped into classes; in object-oriented terms, we discover and describe the classes involved in the problem domain.
- Everything is an object and each object is responsible for itself.

Example

Consider the Windows application needs Windows objects A Windows object is responsible for things like opening, sizing, and closing itself. Frequently, when a window displays something, that something also is an object (a chart, for example). A chart object is responsible for things like maintaining its data and labels and even for drawing itself.

Object-oriented methods enable us to create sets of objects that works together synergistically to produce software that better model their problem domains than similar systems produced by traditional techniques. The systems are easier to adapt to changing requirements, easier to maintain, more robust, and promote greater de-sign and code reuse. Object-oriented development allows us to create modules of functionality. Once objects are defined, it can be taken for granted that they will perform their desired functions and you can seal them off in your mind like black boxes. Your attention as a programmer shifts to what they do rather than how they do it. Here are some reasons why object orientation works.

1.3.1. Why an object oriented

- **Object oriented systems are**
  - Easier to adapt to changes
  - More robust
  - Easier to maintain
  - Promote greater design and code reuse
  - Creates modules of functionality
- **REASONS FOR WORKING OF OBJECT ORIENTED SYSTEMS:**

Higher level of abstraction

The object-oriented approach supports abstraction at the object level. Since objects encapsulate both data (attributes) and functions (methods), they work at a higher level of abstraction. The development can proceed at the object level and ignore the rest of the system for as long as necessary. This makes designing, coding, testing, and maintaining the system much simpler.

Seamless transition among different phases of software development

The traditional approach to software development requires different styles and methodologies for each step of the process. Moving from one phase to another requires a complex transition of perspective between models that almost can be in different worlds. This transition not only can slow the development process but also increases the size of the project and the chance for errors introduced in moving from one language to another. The object-oriented approach, on the other hand, essentially uses the same language to talk about analysis, design, programming, and database design. This seamless approach reduces the level of complexity and redundancy and makes for clearer, more robust system development.

Encouragement of good programming techniques

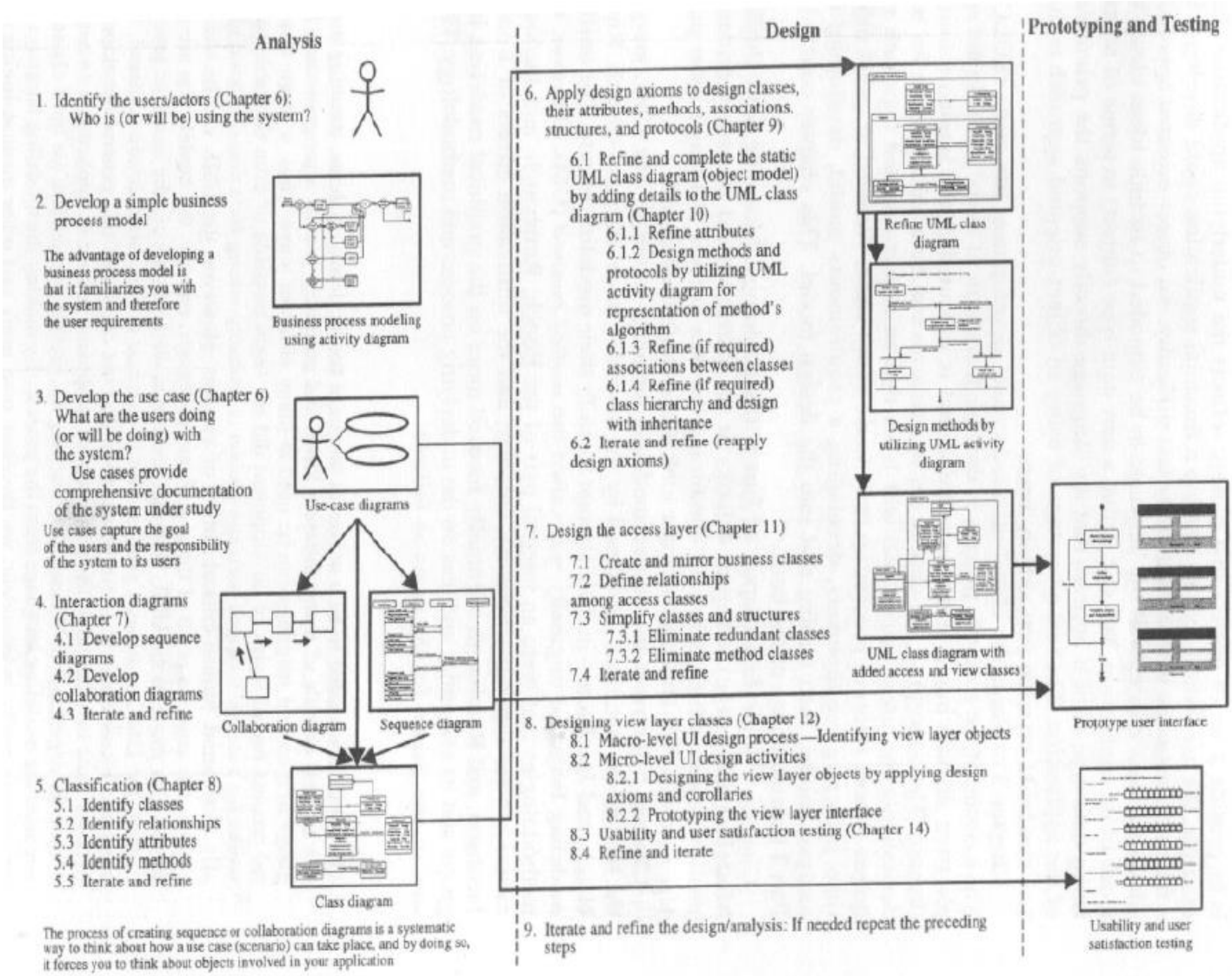
A class in an object-oriented system carefully delineates between its interfaces the routines and attributes within a class are held together tightly. In a properly designed system, the classes will be grouped into subsystems but remain independent; therefore, changing one class has no impact on other classes, and so, the impact is minimized. However, the object-oriented approach is not a panacea; nothing is magical here that will promote perfect design or perfect code.

Promotion of reusability

Objects are reusable because they are modeled directly out of a real-world problem domain. Each object stands by itself or within a small circle of peers (other objects). Within this framework, the class does not concern itself with the rest of the system or how it is going to be used within a particular system.

1.3.2. Overview of the unified approach.

The *unified modeling language* (UML) is a set of notations and conventions used to describe and model an application. But, the UML does not specify a methodology or what steps to follow to develop an application; that would be the task of the UA. Figure 1.2 depicts the essence of the unified approach. The heart of the UA is Jacobson's use case. The use case represents a typical interaction between a user and a computer system to capture the users' goals and needs. The main advantage of an object-oriented system is that the class tree is dynamic and can grow. Your function as a developer in an object-oriented environment is to foster the growth of the class tree by defining new, more specialized classes to perform the tasks your applications require. After your first few projects, you will accumulate a repository or class library of your own, one that performs the operations your applications most often require. At that point, creating additional applications will require no more than assembling classes from the class library.



Fi***gure 1.7 The unified approach road map.***

The UML (Unified Modeling Language) is a set of notations and conventions used to describe and model an application. The Unified Approach (UA) specifies the tasks or steps to develop an application; the heart of UA is Jacobson's use-case. The UA consists of the following concepts:

- Use-case driven approach
- Utilizing the UML for modeling
- Object oriented analysis
- Object oriented design
- Repositories of reusable classes and maximum reuse

- The layered approach
- Incremental development and prototyping
- Continuous testing

1.3.3. An object-oriented philosophy (Reading Assignment)

1.3.4. Basic concepts of an object

1.3.4.1. INTRODUCTION

In an object-oriented system, the algorithm and the data structures are packaged together as an object, which has a set of attributes or properties. The state of these attributes is reflected in the values stored in its data structures. In addition, the object has a collection of procedures or methods-things it can do-as reflected in its package of methods. The attributes and methods are equal and inseparable parts of the object; one cannot ignore one for the sake of the other. For example, a car has certain attributes; such as *color, year, model,* and *price,* and can perform a number of operations, such as *go, stop, turn left,* and *turn right.*

1.3.4.2. OBJECTS

The term *object* was first formally utilized in the Simula language, and objects typically existed in Simula programs to simulate some aspect of reality. The term *object* means a combination of data and logic that represents some real-world entity.

An object is a chunk of structured data in a running software system. It can represent anything with which you can associate properties and behavior. Properties characterize the object, describing its current state. Behavior is the way an object acts and reacts, possibly changing its state.

Figure 1.3 shows some of the objects and their properties that might be important to a particular banking system. The notation used in Figure 1.3 to represent objects is UML

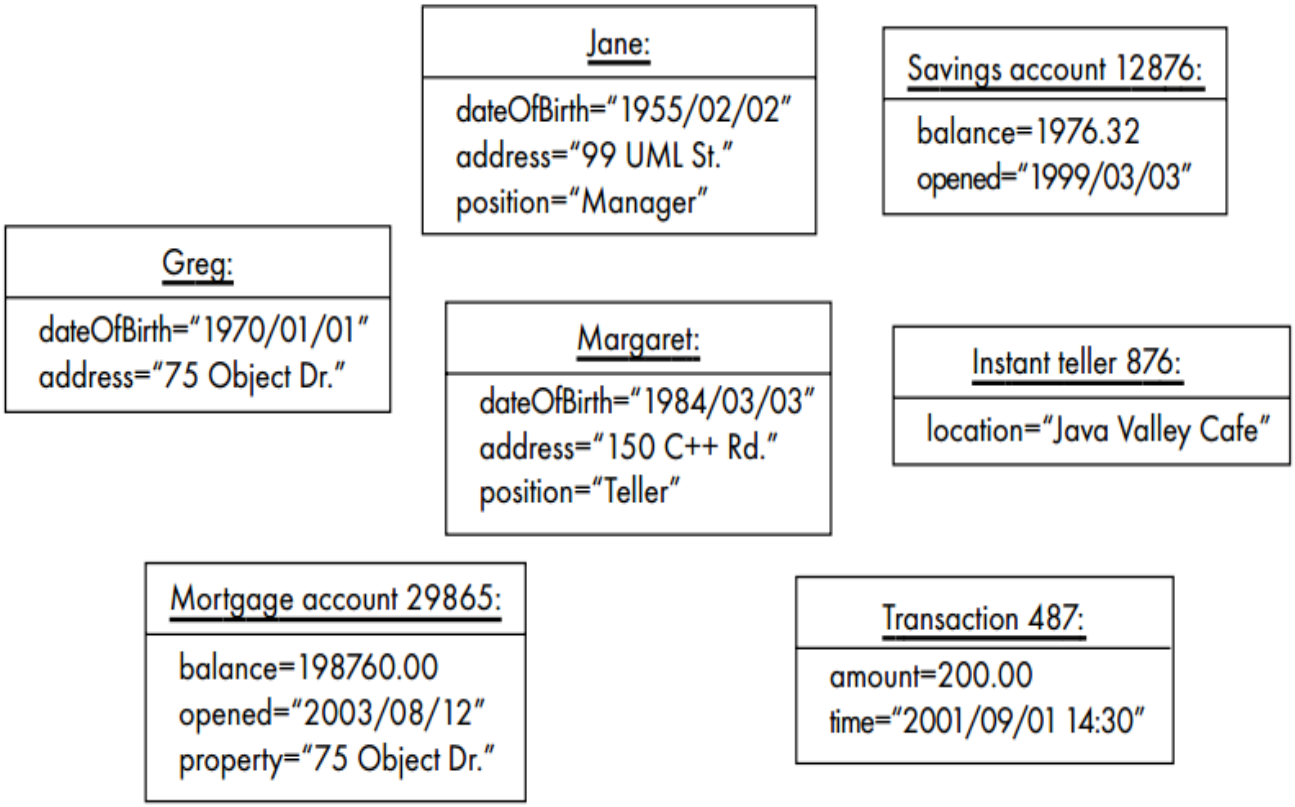


Figure 1.8 Several objects in a banking application

For example, consider a Saab automobile. The Saab can be represented in a computer program as an object. The "data" part of this object would be the car's name, color, number of doors, price, and so forth. The "logic" part of the object could be a collection of programs.

The following are some other examples of objects:

- In a payroll program, there would be objects representing each individual employee.
- In a university registration program, there would be objects representing each student, each course and each faculty member.
- In a factory automation system, there might be objects representing each assembly line, each robot, each item being manufactured, and each type of product.

In the above examples, all the objects represent things that are important to the users of the program. You use a process often called object-oriented analysis to decide which objects will be important to the users, and to work out the structure, relationships and behavior of these objects.
In an object-oriented system, everything is an object: A spreadsheet, a cell in a spreadsheet, a bar chart, a title in a bar chart, a report, a number or telephone number, a file, a folder, a printer, a word or sentence, even a single character all are examples of an object. Each of us deals with objects daily. Some objects, such as a telephone, are so common that we find them in many places. Other objects, like the folders in a file cabinet or the tools we use for home repair, may be located in a certain place.

1.3.4.3. Classes and their instances

Classes are the units of data abstraction in an object-oriented program. More specifically, a class is a software module that represents and defines a set of similar objects, its instances. All the objects with the same properties and behavior are instances of one class.
For example, Figure 1.4 shows how the bank employees Jane and Margaret from Figure 2.2 can be represented as instances of a single class Employee. Class Employee declares that all its instances have a name, a date Of Birth, an address and a position.

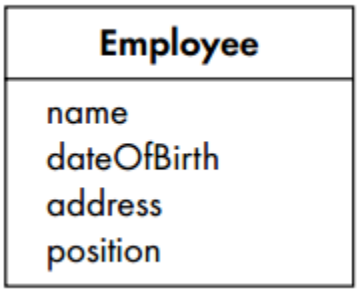


Figure 1.9 A class, representing similar objects from Figure 1.3

- As a software module, a class contains all of the code that relates to its objects, including: Code describing how the objects of the class are structured – i.e. the data stored in each object that implement the properties.
- The procedures, called methods, that implement the behavior of the objects.
  In other words, in addition to defining properties such as name and address, as shown in Figure 1.3, an Employee class would also provide methods for creating a new employee,

and changing an employee's name, address and position. We will talk more about the contents of a class in Sections 1.3 and 1.4.

Sometimes it is hard for beginners to decide what should be a class and what should be an instance. The following two rules can help:

- In general, something should be a class if it could have instances.
- In general, something should be an instance if it is clearly a single member of the set defined by a class.

For example, in an application for managing hospitals, one of the classes might be Doctor, and another might be Hospital. You might think that Hospital should be an instance if there is only one of them in the system; however, the fact that in theory there could be multiple hospitals tells us that Hospital should be a class.

1.3.5. Attributes of an object, its state and properties.

Properties represent the state of an object. Often, we want to refer to the description of these properties rather than how they are represented in a particular programming language. In our example, the properties of a car, such as color, manufacturer, and cost, are abstract descriptions (Figure 1.3).

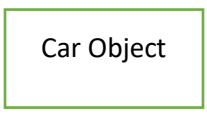
Figure 1.10. The attributes of a car object.

1.3.5.1. OBJECTS RESPOND TO MESSAGES:

The capability of an object's is determined by the methods defined for it. To do an operation, a message is sent to an object. Objects represented to messages according to the methods defined in its class.

For example:

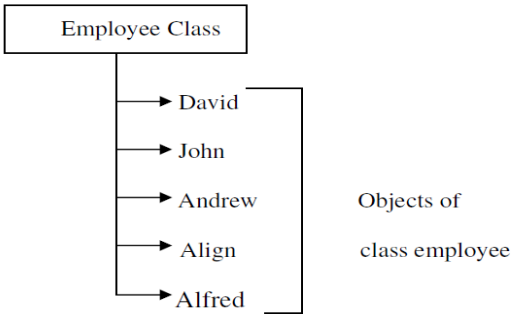
When we press on the brake pedal of a car, we send a stop message to the car object. The car object knows how to respond to the stop message since brake have been designed with specialized parts such as brake pads and drums precisely respond to that message.



Different object can respond to the same message in different ways. The car, motorcycle and bicycle will all respond to a stop message, but the actual operations performed are object specific. It is the receiver's responsibility to respond to a message in an appropriate manner. This gives the great deal or flexibility, since different object can respond to the same message in different ways. This is known as polymorphism.

Objects are grouped in classes: The classification of objects into various classes is based its properties (states) and behavior (methods). Classes are used to distinguish are type of object from another. An object is an instance of structures, behavior and inheritance for objects. The chief rules are the class is to define the properties and procedures and applicability to its instances.

For example:



Class Hierarchy:

An object-oriented system organizes classes into a subclass super class hierarchy. The properties and behaviors are used as the basis for making distinctions between classes are at the top and more specific are at the bottom of the class hierarchy. The family car is the subclass of car. A subclass inherits all the properties and methods defined in its super class.

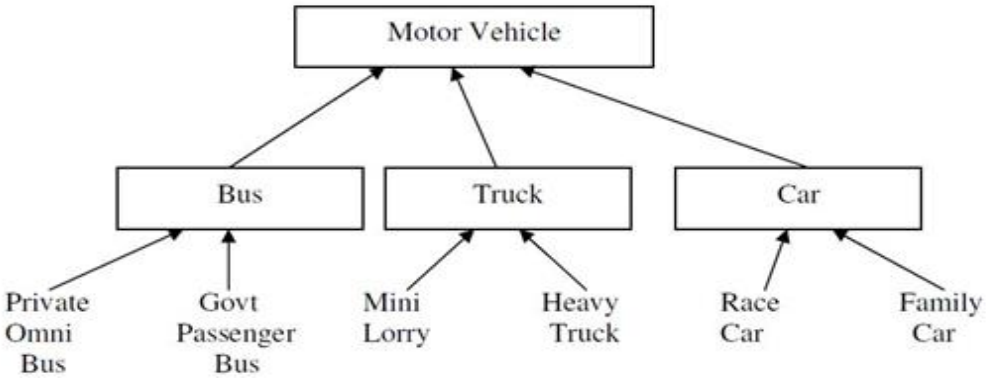


Figure 1.11 Supper Class/Sub Class Hierarchy

Inheritance:

It is the property of object-oriented systems that allow objects to be built from other objects. Inheritance allows explicitly taking advantage of the commonality of objects when constructing new classes. Inheritance is a relationship between classes where one class is the parent class of another (derived) class. The derived class holds the properties and behavior of base class in addition to the properties and behavior of derived class

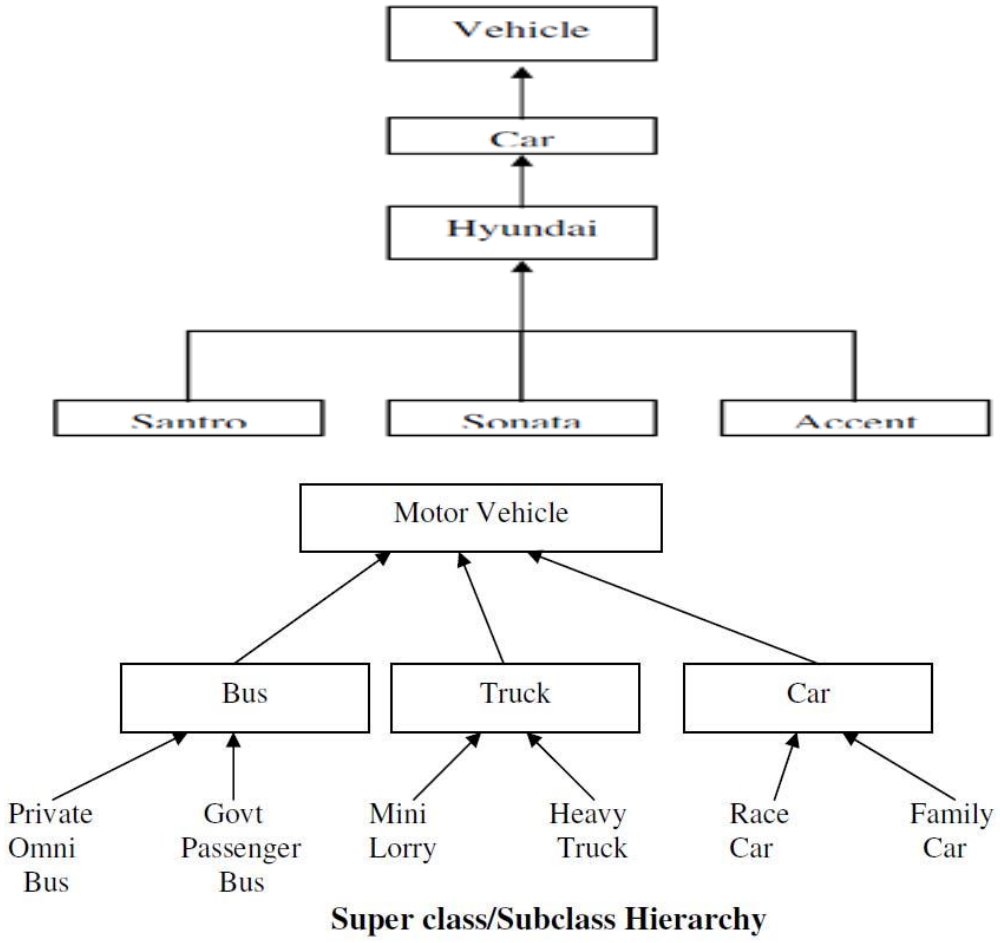


Figure 1.12 Inheritances Allows Reusability

Dynamic inheritance

It allows objects to change and evolve over time. Since base classes provide properties and attributes for objects, hanging base classes changes the properties and attributes of a class.

Example

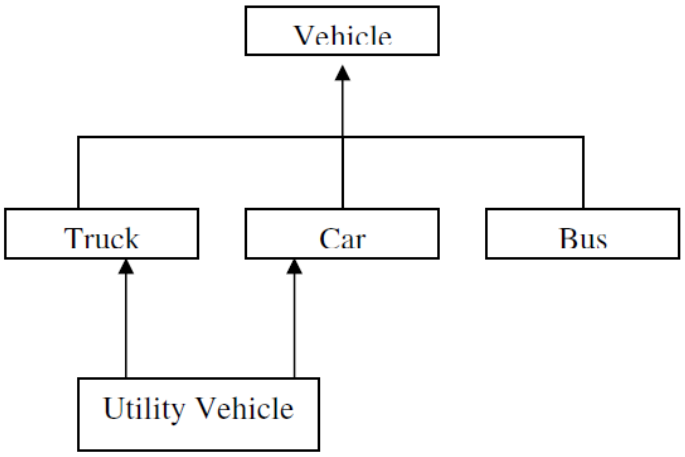
A window objects change to icon and basic again. When we double click the folder the contents will be displayed in a window and when close it, changes back to icon. It involves changing a base class between a windows class and icon class.

Multiple Inheritances:

Some object-oriented systems permit a class to inherit its state (attributes) and behavior from more than one super class. This kind or inheritance is referred to as multiple inheritances.

For example:

Utility vehicle inherits the attributes from the Car and Truck classes.



Summary

- ✓ Object –Oriented paradigm is an approach to the solution of problems in which all computations are performed in the context of objects.
- ✓ Object-oriented programming allows decomposition of a problem into a number of objects and then builds data and functions around these objects.
- ✓ In an object-oriented environment everything is an object and each object is responsible for itself.
- ✓ Object oriented systems are easier to adapt to changes, more robust, and easier to maintain.
- ✓ Software engineering is the process of solving customers' problems by the systematic development and evolution of large, high-quality software systems within cost, time and other constraints.
- ✓ A *software development methodology* is a series of processes leads to the development of an application.

- ✓ Software processes are the activities involved in producing a software system. Software process models are abstract representations of these processes.
- ✓ General process models describe the organization of software processes. Examples of these general models include the waterfall model, incremental development, and reuse-oriented development.
- ✓ Requirements engineering is the process of developing a software specification. Specifications are intended to communicate the system needs of the customer to the system developers.
- ✓ Design and implementation processes are concerned with transforming a requirements specification into an executable software system. Systematic design methods may be used as part of this transformation.
- ✓ Software validation is the process of checking that the system conforms to its specification and that it meets the real needs of the users of the system.
- ✓ Software evolution takes place when you change existing software systems to meet new requirements. Changes are continuous and the software must evolve to remain useful.

References

- Ian Sommerville,Software Engineering (9th edition),Pearson Education
- Behforooz and F. J. Hudson (1996), Software Engineering Fundamentals, Oxford University Press.
- Schach, Stephen R. (2002), Classical and Object-Oriented Software Engineering, 5th ed. IRWIK
- Hoffer, Jeffrey A.; Joey F. George; and Joseph S. Valacicli (1999), Modern Systems Analysis and Design. Massachusetts: Addison-Weslev.
- Roger S. Pressman, Software Engineering: A Practitioner‘s Approach, 5th Ed. Timothy C. Lethbridge and Robert Laganière, Object-Oriented Software Engineering, 2nd Ed.

Chapter One Exercises

1. One is not amongst the disadvantages of Incremental model. (A) Needs good planning and design

(B) Total cost is insignificant compared to waterfall.

(C) Needs a complete definition of the whole system before work break down (D) Needs to capsule divided "increments-pieces" together.

2. Which is the incorrect statement about Waterfall Model?

(A)No phase is complete until the documentation for that phase has been completed and the products of that phase have been approved by the software quality assurance group.

(B)It is called waterfall, because of the cascade from one phase to another. (C)It is an iterative process in which progress is not sequentially flowing. (D) In this model, phases don't overlap.

3. ____is a testing performed by the customer himself after the product delivery to determine whether to accept or reject the delivered product.

(A) Alpha Testing (B) Beta Testing

(C) Acceptance Testing (D) Pilot Testing

4. What is the goal of requirements gathering activity? (A) To build a software that meets standards.

(B) To gather all relevant information from customers. (C) To reveal the inner parts of a certain product.

(D) To design and conceptualize a certain software.

5. When to use the

a. Waterfall model? b. Incremental model?

6. What are the problems that are encountered when waterfall model is applied?

7. Discuss important benefits of Incremental development compared to Waterfall model. 8. Amongst the disadvantages of reusing software, discuss about:

a. Not-invented-here syndrome and b. Lack of tool support.

Chapter 2: Unified Modeling Language (UML)

OBJECTIVES

After the completion of this chapter students will be able to:

- ⇨ Understand basics of UML and its modeling diagrams.
- ⇨ Learn What the UML is
- ⇨ Model software using UML
- ⇨ Appreciate that UML is a language, *not* a methodology

INTRODUCTION

Before we start writing documentation for software that solves a problem, you should have a thorough understanding of Unified Modeling Language. You should also know how to organize the different diagrams that constitute the unified modeling language. In the first chapter we have talked about Unified Modeling Language in brief. But in this chapter we discuss the history of unified modeling language, the different diagrams in unified modeling language and the organization of those diagrams in detail. The concepts present here are crucial in describing the overall system using unified modeling language.

We introduce unified modeling language's things, diagrams and relationships, three of the building blocks that are vital in describing our system. We devote this chapter to better explain unified modeling language using Emergency Report case study.

2.1. An Overview Of UML

The Unified Modeling Language (UML) is a standard graphical language for modeling object-oriented software. It was developed in the mid-1990s as a collaborative effort by James Rumbaugh, Grady Booch and Ivar Jacobson, eachof whom had developed their own notation in the early 1990s. The unified modeling language is a language for specifying, constructing, visualizing, and documenting the software system and its components. The UML is a graphical language with sets of rules and semantics. The rules and semantics of a model are expressed in English, in a form known as *object constraint language* (OCL). OCL is a specification language that uses simple logic for specifying the properties of a system. The UML is not intended to be a visual programming language in the sense of having all the necessary visual and semantic support to

replace programming languages. However, the UML does have a tight mapping to a family of object-oriented languages, so that you can get the best of both worlds.

The goals of the unification efforts were to keep it simple; to cast away elements of existing Booch, OMT, and OOSE methods that did not work in practice; to add elements from other methods that were more effective; and to invent new methods only when an existing solution was unavailable. Because the UML authors, in effect, were designing a language (albeit a graphical one), they had to strike a proper balance between minimalism (everything is text and boxes) and over engineering (having a symbol or fig for every conceivable modeling element).

The primary goals in the design of the UML were as follows:

- Provide users a ready-to-use, expressive visual modeling language so they can develop and exchange meaningful models.
- Provide extensibility and specialization mechanisms to extend the core concepts.
- Be independent of particular programming languages and development processes.
- Provide a formal basis for understanding the modeling language.
- Encourage the growth of the 00 tools market.
- Support higher-level development concepts.
- Integrate best practices and methodologies.

2.2. Building Blocks Of UML

As UML describes the real time systems it is very important to make a conceptual model and then proceed gradually. Conceptual model of UML can be mastered by learning the following three major elements:

- UML building blocks
- Rules to connect the building blocks
- Common mechanisms of UML

This chapter describes all the UML building blocks. The building blocks of UML can be defined as:

- Things
- Relationships
- Diagrams

(1) Things:

Things are the most important building blocks of UML. Things can be:

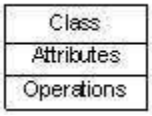
- Structural
- Behavioral
- Grouping
- An notational

Structural things:

The **Structural things** define the static part of the model. They represent physical and conceptual elements. Following are the brief descriptions of the structural things.

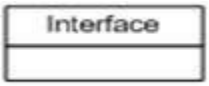
Class:

Class represents set of objects having similar responsibilities.



Interface:

Interface defines a set of operations which specify the responsibility of a class.



Collaboration:

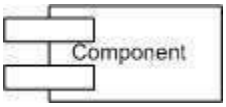
Collaboration defines interaction between elements.

Use case:

Use case represents a set of actions performed by a system for a specific goal.

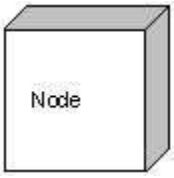
Component:

Component describes physical part of a system.



Node:

A node can be defined as a physical element that exists at run time.

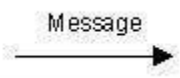


Behavioral things:

A behavioral thing consists of the dynamic parts of UML models. Following are the behavioral things:

Interaction:

Interaction is defined as a behavior that consists of a group of messages exchanged among elements to accomplish a specific task.



State machine:

State machine is useful when the state of an object in its life cycle is important. It defines the sequence of states an object goes through in response to events. Events are external factors responsible for state change. 

Grouping things:

Grouping things can be defined as a mechanism to group elements of a UML model together. There is only one grouping thing available:

Package:

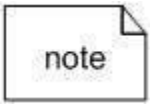
Package is the only one grouping thing available for gathering structural and behavioral things.

An notational things:

An notational things can be defined as a mechanism to capture remarks, descriptions, and comments of UML model elements. **Note** is the only one An notational thing available.

Note:

A note is used to render comments, constraints etc. of an UML element.



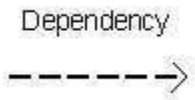
(2) Relationship :

Relationship is another most important building block of UML. It shows how elements are associated with each other and this association describes the functionality of an application.

There are four kinds of relationships available.

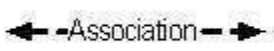
Dependency:

Dependency is a relationship between two things in which change in one element also affects the other one.



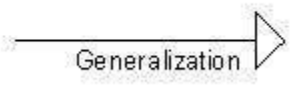
Association:

Association is basically a set of links that connects elements of an UML model. It also describes how many objects are taking part in that relationship.



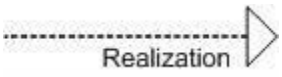
Generalization:

Generalization can be defined as a relationship which connects a specialized element with a generalized element. It basically describes inheritance relationship in the world of objects.



Realization:

Realization can be defined as a relationship in which two elements are connected. One element describes some responsibility which is not implemented and the other one implements them. This relationship exists in case of interfaces.



2.3. UML Diagrams:

UML diagrams are the ultimate output of the entire discussion. All the elements, relationships are used to make a complete UML diagram and the diagram represents a system.

The visual effect of the UML diagram is the most important part of the entire process. All the other elements are used to make it a complete one.

UML includes the following nine diagrams and the details are described in the following chapters.

- Class diagram
- Object diagram
- Use case diagram
- Sequence diagram
- Collaboration diagram
- Activity diagram
- State chart diagram
- Deployment diagram
- Component diagram

2.3.1. Use Case Diagrams

Use cases and actors

Actors are external entities that interact with the system. Examples of actors include a user role (e.g., a system administrator, a bank customer, a bank teller) or another system (e.g., a central database, a fabrication line). Actors have unique names and descriptions.

Use cases describe the behavior of the system as seen from an actor's point of view. Behavior described by the use case model is also called **external behavior**. A use case describes a function provided by the system as a set of events that yields a visible result for the actors. Actors initiate a use case to access the system functionality. The use case can then initiate other use cases and gather more information from the actors. When actors and use cases exchange information, they are said to **communicate**. We will see later that we represent these exchanges with communication relationships.

For example, in an accident management system, field officers, such as, a police officer or a fireman have access to a wireless computer that enables them to interact with a dispatcher. The dispatcher in turn can visualize the current status of all its resources, such as, police cars or trucks, on a computer screen and dispatch a resource by issuing commands from a workstation. In this example, FieldOfficer and Dispatcher are actors.

Figure 2.1 depicts the actor FieldOfficer who invokes the use case ReportEmergency to notify the actor Dispatcher of a new emergency. As a response, the Dispatcher invokes the OpenIncident use case to create an incident report and initiate the incident handling. The Dispatcher enters preliminary information from the FieldOfficer in the incident database and orders additional units to the scene with the AllocateResources use case.

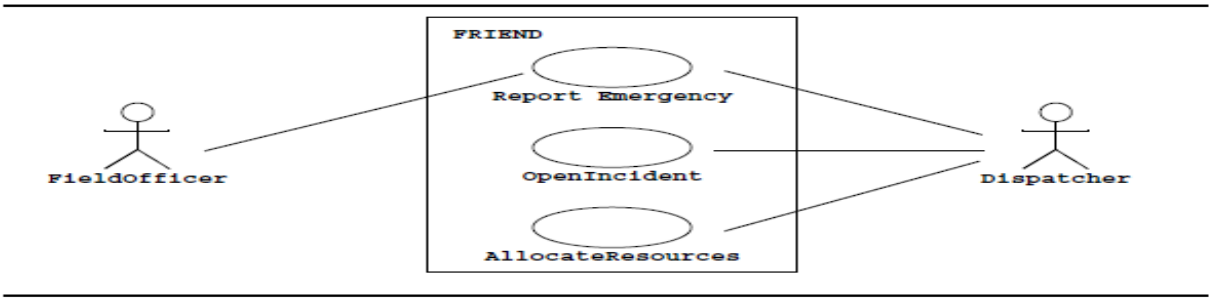


Figure 2.1 An example of a UML use case diagram

Incident initiation in an accident management system. Associations between actors and use cases represent information flows. In UML, these associations are bidirectional: They can represent the actor initiating a use case (e.g., FieldOfficer initiates ReportEmergency) or a use case providing information to an actor (e.g., ReportEmergency notifies Dispatcher).

To describe a use case, we use a template composed of six fields (see also Figure 2.2):

- The **name** of the use case is unique across the system so that developers (and project participants) can unambiguously refer to the use case.
- **Participating actors** are actors interacting with the use case.
- **Entry conditions** describe the conditions that need to be satisfied before the use case is initiated.
- The **flow of events** describes the sequence of actions of the use case, which are be numbered for reference. The common case (i.e., cases that occur frequently) and the exceptional cases (i.e., cases that seldom occur, such as errors and unusual conditions) are described separately in different use cases for clarity.
- **Exit conditions** describe the conditions that are satisfied after the completion of the use case.
- **Special requirements** are requirements that are not related to the functionality of the system. These include constraints on the performance of the system, its implementation, the hardware platforms it runs on, and so on.

Use cases are written in natural language. This enables developers to use them for communicating with the client and the users, who generally do not have an extensive knowledge of software

engineering notations. The use of natural language also enables participants from other disciplines to understand the requirements of the system.

Use case name	`ReportEmergency`
Participating actor	Invoked by `FieldOfficer` Communicates with `Dispatcher`
Entry condition	1. The `FieldOfficer` activates the "Report Emergency" function of her terminal. FRIEND responds by presenting a form to the officer.
Flow of events	2. The `FieldOfficer` fills the form by selecting the emergency level, type, location, and brief description of the situation. The `FieldOfficer` also describes possible responses to the emergency situation. Once the form is completed, the `FieldOfficer` submits the form, at which point the `Dispatcher` is notified. 3. The `Dispatcher` reviews the submitted information and creates an `Incident` in the database by invoking the `OpenIncident` use case. The `Dispatcher` selects a response and acknowledges the emergency report.
Exit condition	4. The `FieldOfficer` receives the acknowledgment and the selected response.
Special requirements	The `FieldOfficer`'s report is acknowledged within 30 seconds. The selected response arrives no later than 30 seconds after it is sent by the `Dispatcher`.

Figure 2.2An example of a use case: the ReportEmergency use case.

Scenarios

A use case is an abstraction that describes all possible scenarios involving the described functionality. A **scenario** is an instance of a use case describing a concrete set of actions. Scenarios are used as examples for illustrating common cases. Their focus is on understandability. Use cases are used to describe all possible cases. Their focus is on completeness. We describe a scenario using a template with three fields:

- The **name** of the scenario enables us to refer to it unambiguously. The name of a scenario is underlined to indicate that it is an instance.
- The **participating actor instances** field indicates which actor instances are involved in this scenario. Actor instances also have underlined names.
- The **flow of events** of a scenario describes the sequence of events step by step.

Communication relationships

Actors and use cases **communicate** when information is exchanged between them. Communication relationships are depicted by a solid line between the actor and use case symbol. In Figure 2.1, the actors FieldOfficer and Dispatcher communicate with the ReportEmergency use case. Only the actor Dispatcher communicates with the use cases OpenIncident and AllocateResources. Communication relationships between actors and use cases can be used to denote access to functionality. In the case of our example, a FieldOfficer and a Dispatcher are provided with different interfaces to the system and have access to different functionalities.

Include relationships

When describing a complex system, its use case model can become quite complex and can contain redundancy. We reduce the complexity of the model by identifying commonalities in different use cases. For example, assume that the Dispatcher can press at any time a key to access Help. This

can be modeled by a use case HelpDispatcher that is included by the use cases OpenIncident and AllocateResources (and any other use cases accessible by the Dispatcher). The resulting model only describes the HelpDispatcher functionality once, thus reducing complexity. Two use cases are related by an include relationship if one of them includes the second one in its flow of events. In UML, **include relationships** are depicted by a dashed arrow originating from the including use case. Include relationships are labeled with the string <<include>>.

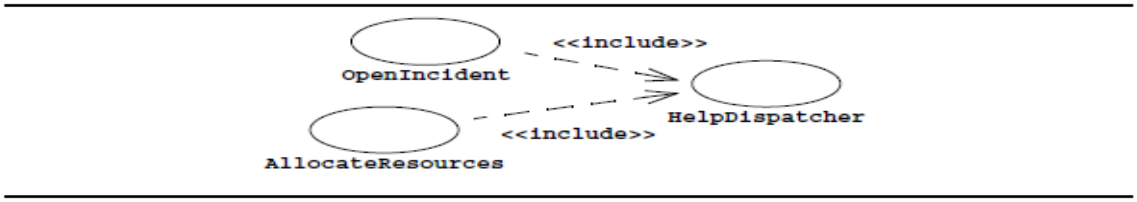


Figure 2.3 An example of an <<include>>relationship (UML use case diagram).

We represent include relationships in the use case itself with one of two ways. If the included use case can be included at any point in the flow of events (e.g., the HelpDispatcher use case), we indicate the inclusion in the *Special requirements* section of the use case. If the included use case is explicitly invoked during an event, we indicate the inclusion in the flow of events.

Extend relationships

Extend relationships are an alternate means for reducing complexity in the use case model. A use case can extend another use case by adding events. An extend relationship indicates that an instance of an extended use case may include (under certain conditions) the behavior specified by the extending use case. A typical application of extend relationships is the specification of exceptional behavior. For example (Figure 2-19), assume that the network connection between the Dispatcher and the FieldOfficer can be interrupted at any time. (e.g., if the FieldOfficer enters a tunnel). The use case ConnectionDown describes the set of events taken by the system and the actors while the connection is lost. ConnectionDown extends the use cases OpenIncident and AllocateResources. Separating exceptional behavior from common behavior enables us to write shorter and more focused use cases.

In the textual representation of a use case, we represent extend relationships as entry conditions of the extending use case. For example, the extend relationships depicted in Figure 2.4 are represented as an entry condition of the ConnectionDown use case

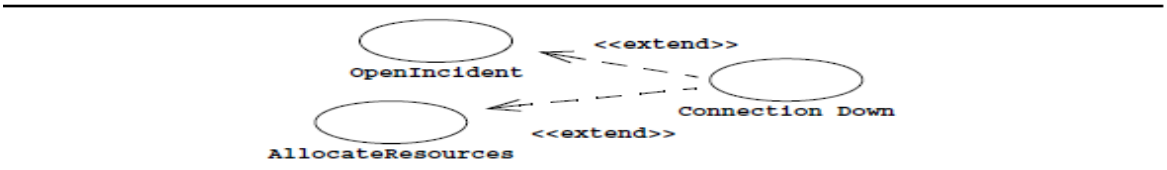


Figure 2.4 An example of an <<extend>>relationship (UML use case diagram).

Generalization relationships

Generalization/specialization relationships are a third mechanism for reducing the complexity of a model. A use case can specialize a more general one by adding more detail. For example, FieldOfficers are required to authenticate before they can use FRIEND. During early stages of requirements elicitation, authentication is modeled as a high-level Authenticate use case. Later, developers describe the Authenticate use case in more detail and allow for several different hardware platforms. This refinement activity results in two more use cases, AuthenticateWithPasswordwhich enables FieldOfficers to login without any specific hardware, and AuthenticateWithCard, which enables FieldOfficers to login using a smart card. The two new use cases are represented as specializations of the Authenticate use case.

2.3.2. Class Diagrams

Classes and objects

Class diagrams describe the structure of the system in terms of classes and objects. **Classes** are abstractions that specify the attributes and behavior of a set of objects. **Objects** are entities that encapsulate state and behavior. Each object has an identity: It can be referred individually and is distinguishable from other objects.

In UML, classes and objects are depicted by boxes including three compartments. The top compartment displays the name of the class or object. The center compartment displays its attributes, and the bottom compartment displays its operations. The attribute and operation compartment can be omitted for clarity. Object names are underlined to indicate that they are instances. By convention, class names start with an uppercase letter. Objects in object diagrams may be given names (followed by their class) for ease of reference. In that case, their name starts with a lowercase letter.

The main symbols shown on class diagrams are:

- **Classes**, which represent the types of data themselves.
- **Associations**, which show how instances of classes reference instances of otherclasses.
- **Attributes**, which are simple data found in instances.
- **Operations**, which represent the functions performed by the instances.
- **Generalizations**, which are used to arrange classes into inheritance hierarchies.

In the FRIEND, Bob and Alice are field officers, represented in the system as FieldOfficer objects called bob: FieldOfficer and alice:FieldOfficer. FieldOfficer is a class describing all FieldOfficer objects, whereas Bob and Alice are represented by two individual FieldOfficer objects.

In Figure 2.5, the FieldOfficer class has two attributes: a name and a badgeNumber. This indicates that all FieldOfficer objects have these two attributes. the bob:FieldOfficerand alice:FieldOfficer objects have specific values for these attributes: “Bob. D.” and “Alice W.”, respectively. In Figure 2.5, the FieldOfficer.name attribute is of type String, which indicates that only instances of String can be assigned to the FieldOfficer.name attribute. The type of an attribute is used to specify the valid range of values the attribute can take. Note that when attribute types are not essential to the

definition of the system, attribute type decisions can be delayed until object design. This allows the developers to concentrate on the functionality of the system and to minimize the number of trivial changes when the functionality of the system is revised.

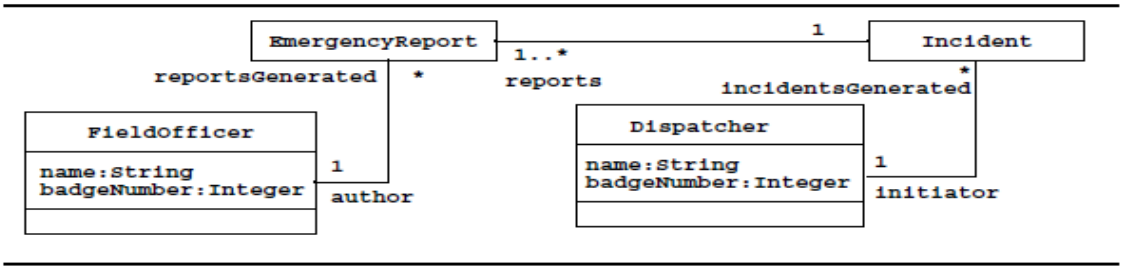


Figure 2.5 An example of a UML class diagram

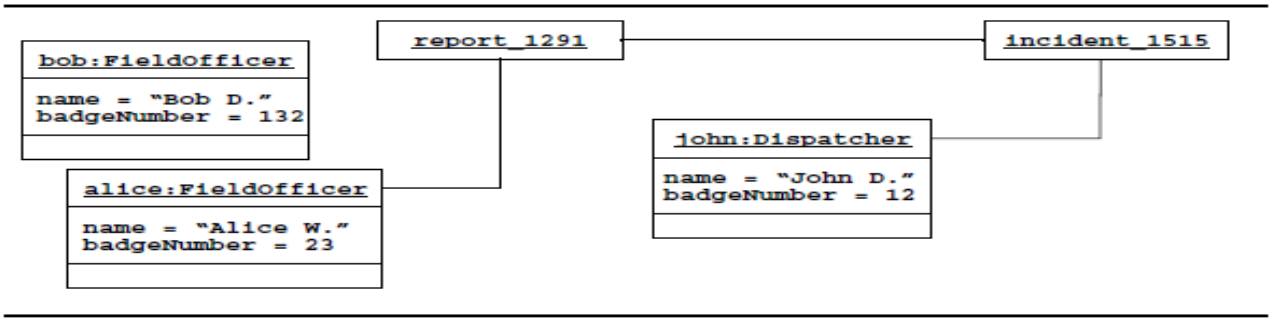
Classes that participate in the ReportEmergency use case. Detailed type information is usually omitted until object design.

Associations and links

A **link** represents a connection between two objects. **Associations** are relationships between classes and represent groups of links. Each FieldOfficer object also has a list of EmergencyReports that has been written by the FieldOfficer. In Figure 2.5, the line between the FieldOfficer class and the EmergencyReport class is an association. In Figure 2.6, the line between the alice:FieldOfficerobject and the report_1291:EmergencyReporotbject is a link. This link represents state that is kept in the system to denote that alice:FieldOfficergenerated report_1291:EmergencyReport.

Roles

Each end of an association can be labeled by a string called **role**. In Figure 2.5, the roles of the association between the EmergencyReport and FieldOfficer classes are author and reportsGenerated. Labeling the end of associations with roles allows us to distinguish multiple



associations originating from a class. Moreover, roles clarify the purpose of the association.

Figure 2.6 An example of a UML object diagram: objects that participate in the warehouse On Fire scenario.

Multiplicity

Each end of an association can be labeled by a set of integers indicating the number of links that can legitimately originate from an instance of the class connected to the association end. The association end author has a multiplicity of l. This means that all EmergencyReports are written

by exactly one FieldOfficer. In other terms, each EmergencyReport object has exactly one link to an object of class FieldOfficer. The multiplicity of the association end reportsGenerated role is "many," shown as a star. The "many" multiplicity is shorthand for 0..n. This means that any given FieldOfficer may be the author of zero or more EmergencyReports.

Association class

Associations are similar to classes, in that they can have attributes and operations attached to them. Such an association is called an **association class** and is depicted by a class symbol, containing the attributes and operations, connected to the association symbol with a dashed line.

For example, in Figure 2.7, the allocation of FieldOfficers to an Incident is modeled as anassociation class with attributes role and notificationTime.

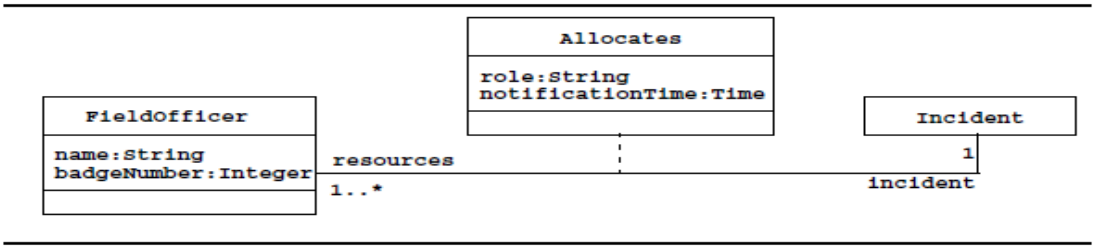


Figure 2.7 An example of an association class (UML class diagram).

Any association class can be transformed into a class and simple associations as shown in Figure 2.8. Although both representations are similar, the association class representation is clearer: An association cannot exist without the classes it links. Similarly, the Allocation object cannot exist without a FieldOfficer and an Incident object. Although Figure 2.8 carries the same information, this diagram requires careful examination of the association multiplicity.

Aggregation

Associations are used to represent a wide range of connections among a set of objects. In practice, a special case of association occurs frequently: composition. For example, a State contains many Counties, which in turn contains many Townships. A PoliceStation is composed of PoliceOfficers. Another example is a Directory that contains a number of Files. Such relationships could be modeled using a one-to-many association. Instead, UML provides the concept of an **aggregation** to denote composition. An aggregation is denoted by a simple line with a diamond at the container end of the association. Although one-to-many associations and aggregations can be used alternatively, aggregations are preferable because they emphasize the hierarchical aspects of the relationship.

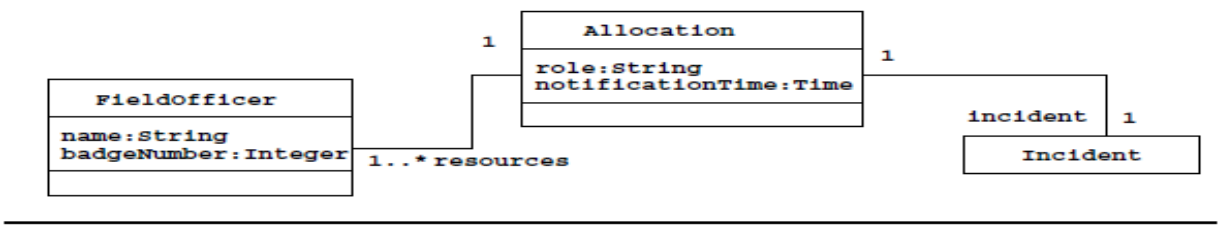


Figure 2.8 Alternative model for Allocation (UML class diagram).

Generalization

Generalization is the relationship between a general class and one or more specialized classes. Generalization enables us to describe all the attributes and operations that are common to a set of classes. For example, FieldOfficer and Dispatcher both have name and badgeNumber attributes. However, FieldOfficer has an association with EmergencyReport, whereas Dispatcher has an association with Incident. The common attributes of FieldOfficer and Dispatcher can be modeled by introducing a *PoliceOfficer* class that is specialized by the FieldOfficer and the Dispatcher classes (see Figure 2.9). *PoliceOfficer*, the generalization, is called a **superclass.** FieldOfficer and Dispatcher, the specializations, are called the **subclasses**.

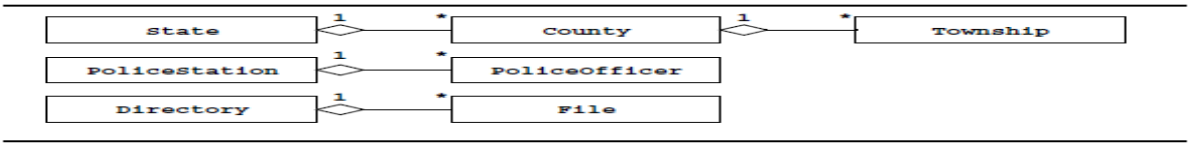


Figure 2.9 An example of a generalization (UML class diagram).

PoliceOfficer is an abstract class which defines the common attributes and operations of the FieldOfficer and Dispatcher classes.

For example, consider a watch with two buttons (hereafter called 2Bwatch). Setting the time on 2Bwatch requires the actor 2BWatchOwner to first press both buttons simultaneously, after which 2Bwatch enters the set time mode. In the set time mode, 2Bwatch blinks the number being changed (e.g., the hours, the minutes, or the seconds, day, month, year). Initially, when the 2BWatchOwner enters the set time mode, the hours blink. If the actor presses the first button, the next number blinks (e.g, if the hours are blinking and the actor presses the first button, the hours stop blinking and the minutes start blinking. If the actor presses the second button, the blinking number is incremented by one unit. If the blinking number reaches the end of its range, it is reset to the beginning of its range (e.g., assume the minutes are blinking and its current value is 59, its new value is set to 0 if the actor presses the second button). The actor exits the set time mode by pressing both buttons simultaneously. Figure 2.10 depicts a sequence diagram for an actor setting his 2Bwatch one minute ahead.

Each column represents an object that participates in the interaction. The vertical axis represents time from top to bottom. Messages are shown by arrows. Labels on arrows represent message names and may contain arguments. Activations (i.e., executing methods) are depicted by vertical rectangles. Actors are shown as the leftmost column.

Sequence diagrams can be used to describe either an abstract sequence (i.e., all possible interactions) or concrete sequences (i.e., one possible interaction, as in Figure 2.10). When describing all possible interactions, sequence diagrams also provide notations for conditionals and iterators.

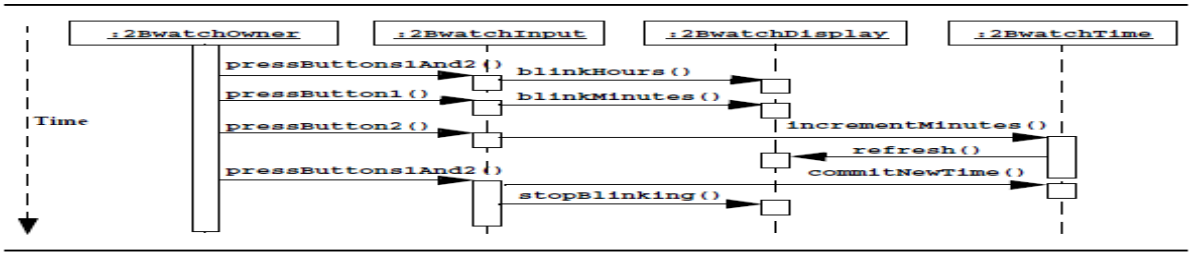


Figure 2.10 Example of a sequence diagram: setting the time on 2Bwatch.

2.3.3. State chart Diagram

A **UML statechart** is a notation for describing the sequence of states an object goes through in response to external events. Statecharts are extensions of the finite state machine model. On the one hand, statecharts provide notation for nesting states and state machines (i.e., a state can be described by a state machine). On the other hand, statecharts provide notation for binding transitions with message sends and conditions on objects.

A **state** is a condition that an object satisfies. A state can be thought of as an abstraction of the attribute values of a class. For example, an Incident object in FRIEND can be in four states: Active, Inactive, Closed, and Archived. An active Incident denotes a situation that requires a response (e.g., an ongoing fire, a traffic accident). An inactive Incident denotes a situation that was handled but for which reports still need to be written (e.g., the fire has been put out but damage estimates have not yet been performed). A closed Incident denotes a situation that has been handled and documented. An archived Incident is a closed Incident whose documentation has been moved to off-site storage.

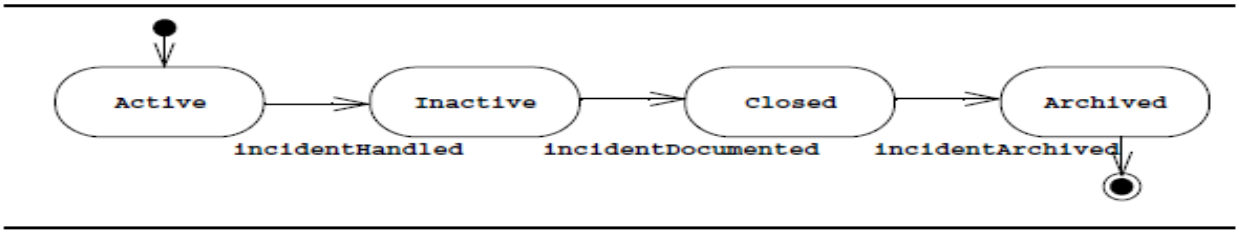


Figure 2.11 A UML statechart diagram for the Incident class.

A **transition** represents changes of state triggered by events, conditions, or time. there are three transitions: from the Active state into the Inactive state, from the Inactive state to the Closed state, and from the Closed state to the Archived state.

A state is depicted by a rounded rectangle. A transition is depicted by arrows connecting two states. States are labeled with their name. A small solid black circle indicates the initial state. A circle surrounding a small solid black circle indicates a final state.

displays another example, a state chart for the 2Bwatch (for which we constructed a sequence diagram). At the highest level of abstraction, 2Bwatch has two states, MeasureTime and SetTime. 2Bwatch changes states when the user presses and releases both buttons simultaneously. When 2Bwatch is first powered on, it is in the SetTime state. This is indicated by the small solid black

circle, which represents the initial state. When the battery of the watch runs out, the 2Bwatch is permanently out of order.

The statechart diagram does not represent the details of measuring or setting the time. These details have been abstracted away from the statechart diagram and can be modeled separately using either internal transitions or a nested statechart. Internal transitions are transitions that remain within a single state. They can also have actions associated with them. Entry and exit are displayed as an internal transition, given that their actions do not depend on the originating and destination states. Nested statecharts reduce complexity. They can be used instead of internal transitions. In Figure 2.20, the current number is modeled as nested state, whereas actions corresponding to modifying the current number are modeled using internal transitions. Note that each state could be modeled as a nested statechart (e.g., the BlinkHours statechart would have 24 substates that correspond to the hours in the day; transitions between these states would correspond to pressing the second button).

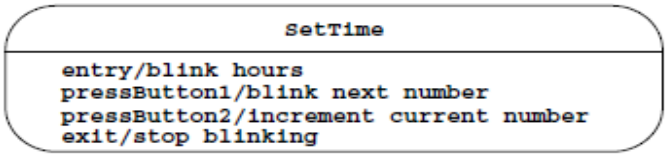


Figure 2.12 Internal transitions associated with the SetTime state (UML statechart diagram).

2.3.4. Activity diagrams

The outgoing transitions are triggered by the completion of an action associated with the state. This is called an **action state**. By convention, the name of a state denotes a condition, whereas the name of an action state denotes an action. **Activity diagrams** are statechart diagrams whose states are action states. is an activity diagram corresponding to the state diagram. An alternate and equivalent view of activity diagrams is to interpret action states as control flow between activities and transitions; that is, the arrows are interpreted as sequential constraints between activities.

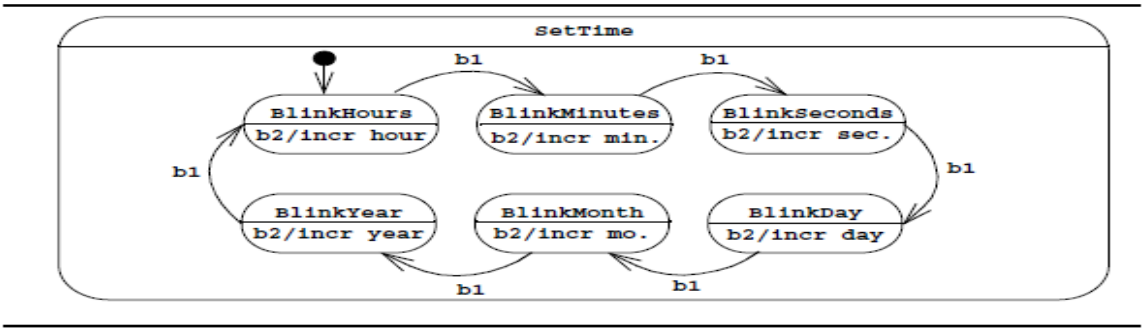


Figure 2.13 Refined statechart associated with the SetTime state (UML statechart diagram).

During the action state HandleIncident, the Dispatcher receives reports and allocates resources. Once the Incident is closed, the Incident moves to the DocumentIncident activity during which all participating FieldOfficers and Dispatchers document the Incident. Finally, the ArchiveIncident activity represents the archival of the Incident related information onto slow access medium.

Decisions are branches in the control flow. They denote alternative transitions based on a condition of the state of an object or a set of objects. Decisions are depicted by a diamond with one or more incoming arrows and two or more outgoing arrows. The outgoing arrows are labeled with the conditions that select a branch in the control flow. The set of all outgoing transitions from a decision represents the set of all possible outcomes.

2.3.5. Diagram organization

Models of complex systems quickly become complex as developers refine them. The complexity of models can be dealt with by grouping related elements into **packages**. A package is a grouping of model elements, such as use cases, classes, or activities, defining scopes of understanding.

For example, Figure 2.14 depicts use cases of the FRIEND system, grouped by actors. Packages are displayed as rectangles with a tab attached to their upper-left corner. Use cases dealing with incident management (e.g., creating, resource allocation, documentation) are grouped in the IncidentManagement package. Use cases dealing with incident archive (e.g., archiving an incident, generating reports from archived incidents) are grouped in the IncidentArchive package. Use cases dealing with system administration (e.g., adding users, registering end stations) are grouped in the SysAdministration package. This enables the client and the developers to organize use cases into related groups and to focus on only a limited set of use cases at a time.

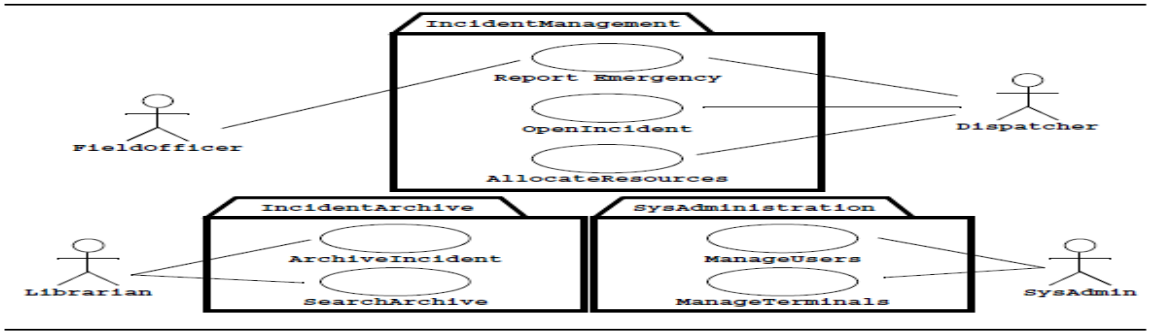
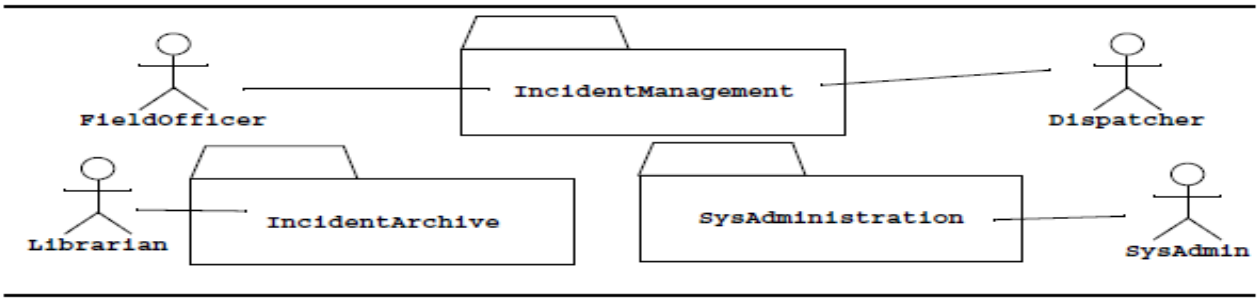


Figure 2.14 Example of packages: use cases of FRIEND organized by actors (UML use case diagram).

Figures 2.14 and 2.15 are examples of class diagrams using packages. Classes from the ReportEmergency use case are organized according to the site where objects are created. FieldOfficer and EmergencyReport are part of the FieldStation package, and Dispatcher and Incident are part of the DispatcherStation. Figure 2.14 displays the packages with the model elements they contain while Figure 2.15 displays the same information without the contents of each package. Figure 2.15 is a higher level picture of the system and can be used for discussing system-level issues, whereas Figure 2.14 is a more detailed view, which can be used to discuss the content of specific packages.

Figure 2.15 Example of packages:



This figure displays the same packages as Figure 2-39 except that the details of each packages are suppressed (UML use case diagram).

Packages (Figure 2.16) are used to deal with complexity the same way a user organizes files and subdirectories into directories. However, packages are not necessarily hierarchical: The same class may appear in more than one package. To reduce inconsistencies, classes (more generally model elements) are owned by exactly one package, whereas the other packages are said to refer to the modeling element. Note that packages are organizing constructs, not objects.

They have no behavior associated with them and cannot send and receive messages. A **note** is a comment attached to a diagram. Notes are used by developers for attaching information to models and model elements. This is an ideal mechanism for recording outstanding issues relevant to a model, clarifying a complex point, or recording to-dos or reminders. Although notes have no semantics per se, they are sometimes used to express constraints that cannot otherwise be expressed in UML.

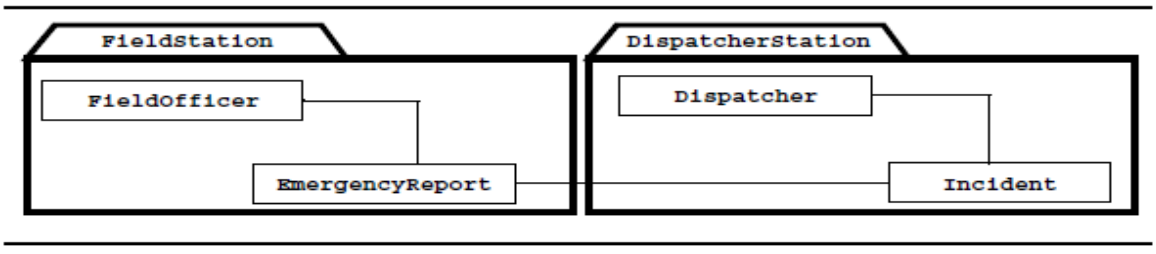


Figure 2.16 Example of packages.

2.3.6. Diagram Extensions

The goal of the UML designers is to provide a set of notations to model a broad class of software systems. They also recognized that a fixed set of notations could not achieve this goal, because it is impossible to anticipate the needs encountered in all application and solution domains. For this reason, UML provides a number of extension mechanisms enabling the modeler to extend the language. In this section, we describe two such mechanisms, **stereotypes** and **constraints**.

A **stereotype** is a string enclosed by angle brackets (e.g., <<subsystem>>), which is attached to a UML element, such as a class or an association. This enables modelers to create new kinds of building blocks that are needed in their domain. For example, during analysis, we classify objects into three types: entity, boundary, and control. The base UML language knows only objects. To introduce these three additional types, we use three stereotypes, <<entity>>, <<boundary>>, and <<control>>to represent the object type (Figure 2.17).

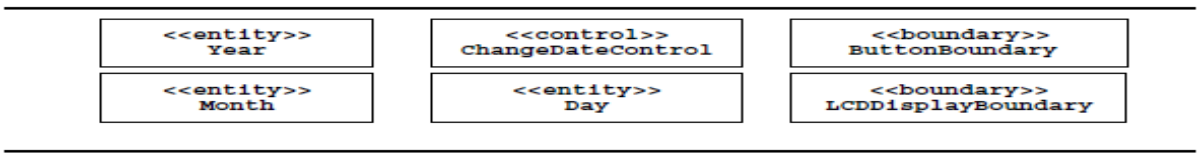


Figure 2.17 Examples of stereotypes (UML class diagram).

A **constraint** is a rule that is attached to a UML building block. This allows to represent phenomena that cannot otherwise be expressed with UML.

Summary

- The Unified Modeling Language (UML) is a standard graphical language for modeling object-oriented software. The Unified Modeling Language (UML) is a language for specifying, constructing, visualizing, and documenting the software system and its components. Object Constraint language (OCL) is a specification language that uses simple logic for specifying the properties of a system.
- Unified Modeling Language has things, diagrams, and relationships as its building blocks.
- **Structural things** define the static part of the model. **Behavioral things** consist the dynamic model of the UML models.
- **Annotational things** can be defined as a mechanism to capture remarks, descriptions, and comments of UML model elements.
- **Grouping things** can be defined as a mechanism to group elements of a UML model together.
- Generalization can be defined as a relationship which connects a specialized element with a generalized element.
- Static and Dynamic diagrams are the two types of diagrams in UML. Actors are external entities that interact with the system.

- Use Cases describe the behavior of the system as seen from an actor's point of view.Scenario is an instance of a use case describing concrete set of actions.
- **Sequence diagrams** describe patterns of communication among a set of interacting objects.A **UML statechart**is a notation for describing the sequence of states an object goes through in response to external events.
- State charts are extensions of the finite state machine model. A **state** is a condition that an object satisfies. A **transition** represents changes of state triggered by events, conditions, or time. **Activity diagrams** are statechart diagrams whose states are action states.A **stereotype** is a string enclosed by angle brackets (e.g., <<subsystem>>), which is attached to a UML element, such as a class or an association.A **constraint** is a rule that is attached to a UML building block.

Review Questions & Problems

1. Which of the following are the primary goals in the design of the UML?

A. Encourage the growth of the OO tools market
B. Support higher-level development concepts
C. Integrate best practice and methodologies
D. All

2. _____ Can be defined as a relationship which connects as a specialized element with a generalized element.
   A. Generalization.
   B. Inheritance
   C. Polymorphism
   D. A & B
3. Which of the following is not among the type of the structural things?
   A. Class
   B. Interface
   C. Collaboration
   D. Interaction
4. _____ is a relationship between two things in which changes in one element also effects the other one.
   A. Association
   B. Aggregation
   C. Dependency
   D. Realization
5. _____ Can be defined as a relationship in which two elements are connected. One element describes the function and other one impalements them.
   A. Realization
   B. Interfaces
   C. Generalization
   D. A & B
6. What kind of association exists between a student and Identification card?
   A. One to One
   B. One to Many
   C. Many to Many
   D. None
7. Which type to diagram is among the UML Static Diagram?
   A. Class Diagram
   B. Sequence Diagram
   C. Activity Diagram
   D. State Chart Diagram
8. In State chart diagram, _____ represents changes of state triggered by events, conditions, or time.

References:

- The Unified Modeling Language User Guide SECOND EDITION Addison Wesley Professional USA

- Applying UML and Patterns 2nd edition, Craig Larman
- Scott, Kendall (2004) Fast Track UML 2.0 Après USA

Chapter 3: Requirements Elicitation

OBJECTIVES

After the completion of this chapter students will be able to:

- ⇨ Define Requirements
- ⇨ Identify actors and use cases.

INTRODUCTION

Requirement's elicitation focuses on describing the purpose of the system. The client, the developers, and the users identify a problem area and define a system that addresses the problem. Such a definition is called a **system specification** and serves as a contract between the client and the developers. The system specification is structured and formalized during analysis to produce an analysis model (see Figure 3.1). Both system specification and analysis model represent the same information. They differ only in the language and notation they use. The system specification is written in natural language, whereas the analysis model is usually expressed in a formal or semiformal notation.

3.1. An Overview of Requirements Elicitation

The system specification supports the communication with the client and users. The analysis model supports the communication among developers. They are both models of the system in the sense that they attempt to accurately represent the external aspects of the system. Given that both models represent the same aspects of the system, requirements elicitation and analysis occur concurrently and iteratively.

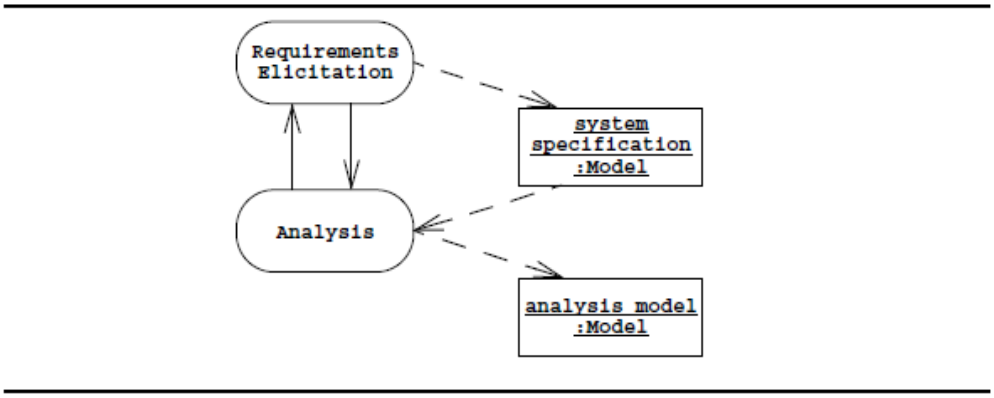


Figure 3.1 Products of requirements elicitation and analysis (UML activity diagram).

Requirements elicitation and analysis focus only on the user's view of the system. For example, the system functionality, the interaction between the user and the system, the errors that the system can detect and handle, and the environmental conditions in which the system functions, are part of the requirements. The system structure, the implementation technology selected to build the system, the system design, the development methodology, and other aspects not directly visible to the user are not part of the requirements.

Requirements elicitation includes the following activities.

- **Identifying actors**. During this activity, developers identify the different types of users the future system will support.
- **Identifying scenarios**. During this activity, developers observe users and develop a set of detailed scenarios for typical functionality provided by the future system. Scenarios are concrete examples of the future system in use. Developers use these scenarios to communicate with the user and deepen their understanding of the application domain.
- **Identifying use cases**. Once developers and users agree on a set of scenarios, developers derive from the scenarios a set of use cases that completely represent the future system. Whereas scenarios are concrete examples illustrating a single case, use cases are abstractions describing all possible cases. When describing use cases, developers determine the scope of the system.
- **Refining use cases**. During this activity, developers ensure that the system specification is complete, by detailing each use case and describing the behavior of the system in the presence of errors and exceptional conditions.
- **Identifying relationships among use cases**. During this activity, developers consolidate the use case model by eliminating redundancies. This ensures that the system specification is consistent.
- **Identifying nonfunctional requirements**. During this activity, developers, users, and clients agree on aspects that are visible to the user but not directly related to functionality. These include constraints on the performance of the system, its documentation, the resources it consumes, its security, and its quality.

During requirements elicitation, developers access many different sources of information, including client-supplied documents about the application domain, manuals and technical documentation of legacy systems that the future system will replace, and, most important, the users

and clients themselves. Developers interact the most with users and clients during requirements elicitation. We focus on three methods for eliciting information and making decisions with users and clients:

Joint Application Design (JAD)

- ✓ focuses on building consensus among developers, users, and clients by jointly developing the system specification.
- ✓ JAD is a means to bring together the key users, managers, and systems analysts involved in the analysis of a current system.
- ✓ The goal of JAD is to collect systems requirements simultaneously from the key people involved in the system.
- ✓ JAD sessions are usually conducted in a location away from where the people normally work, to keep them away from all distractions, so that they can concentrate fully on systems analysis.
- ✓ The people involved in a JAD are
    - **JAD session leader**: To organize and run the JAD- should be trained in
        - Facilitation and group management and is usually a systems analyst. Sets
        - the agenda, resolves conflicts and disagreements, keeps the session on
        - track and gets all ideas from participants
    - **Users:** Key users of the existing and new system
    - **Mangers of User group:** To provide information about the
        - organization's direction and the motivations and impacts of the systems,
        - also to support the requirements determined during the JAD.
    - **Systems Analysts:** To learn from users and mangers, may participate
        - and advise on IS issues
    - **Sponsor**: A senior person in the organization who is supporting and
        - providing the budget for the development, usually attends only at start
        - and end of session
    - **Scribe**: To take notes during the sessions and record the outcomes of the
        - meeting (on a laptop or PC if possible)
    - **IS staff**: To learn from the discussion , may also contribute ideas on
        - technical feasibility or limitations of systems

- ✓ JAD sessions are held in a special room equipped with white boards, audiovisual tools, overhead projector, flip charts and computer generated displays.

Knowledge Analysis of Tasks (KAT)

- ✓ Focuses on eliciting information from users through observation.

Usability testing

- ✓ Focuses on validating the requirements elicitation model with the user through a variety of methods.

3.2. **Requirements Elicitation Concepts**

In this section, we describe the main requirements elicitation concepts we use in this chapter. In particular, we describe:

- Functional requirements
- Nonfunctional and pseudo-requirements
- Levels of descriptions
- Correctness, completeness, consistency, clarity, and realism
- Verifiability and traceability
- Greenfield engineering, reengineering, and interface engineering

3.2.1. Functional Requirements

Functional requirements describe the interactions between the system and its environment independent of its implementation. The environment includes the user and any other external system with which the system interacts. For example, the following is an example of functional requirements for SatWatch, a watch that resets itself without user intervention:

Example:

Functional requirements for SatWatch

SatWatch is a wrist watch that displays the time based on its current location. SatWatch uses GPS satellites (Global Positioning System) to determine its location and internal data structures to convert this location into a time zone. The information stored in the watch and its accuracy measuring time (one hundredth of second uncertainty over five years) is such that the watch owner never needs to reset the time. SatWatch adjusts the time and date displayed as the watch owner crosses time zones and political boundaries (e.g., standard time vs. daylight savings time). For this reason, SatWatch has no buttons or controls available to the user.

SatWatch has a two-line display showing, on the top line, the time (hour, minute, second, time zone) and, on the bottom line, the date (day of the week, day, month, year). The display technology used is such that the watch owner can see the time and date even under poor light conditions. When a new country or state institutes different rules for daylight savings time, the watch owner may upgrade the software of the watch using the WebifyWatch serial device (provided when the watch is purchased) and a personal computer connected to the Internet. SatWatch complies with the physical, electrical, and software interfaces defined by WebifyWatch API 2.0.

The above functional requirements only focus on the possible interactions between SatWatch and its external world (i.e., the watch owner, GPS, and WebifyWatch). The above description does not focus on any of the implementation details (e.g., processor, language, display technology).

3.2.2. Non-Functional and Pseudo-Requirements

Nonfunctional requirements describe user-visible aspects of the system that are not directly related with the functional behavior of the system. Nonfunctional requirements include quantitative constraints, such as response time (i.e., how fast the system reacts to user commands) or accuracy (i.e., how precise are the system's numerical answers). The following are the nonfunctional requirements for SatWatch:

Example:

Nonfunctional requirements for SatWatch

SatWatch determines its location using GPS satellites, and as such, suffers from the same limitations as all other GPS devices (e.g., ~100 feet accuracy, inability to determine location at certain times of the day in mountainous regions). During blackout periods, SatWatch assumes that it does not cross a time zone or a political boundary. SatWatch corrects its time zone as soon as a blackout period ends.

The battery life of SatWatch is limited to 5 years, which is the estimated life cycle of the housing of SatWatch. The SatWatch housing is not designed to be opened once manufactured, preventing battery replacement and repairs. Instead, SatWatch is priced such that the watch owner is expected to buy a new SatWatch to replace a defective or old SatWatch.

Pseudo-requirements are requirements imposed by the client that restrict the implementation of the system. Typical pseudo-requirements are the implementation language and the platform on which the system is to be implemented. For life-critical developments, pseudo-requirements often include process and documentation requirements (e.g., the use of a formal specification method, the complete release of all work products). Pseudo-requirements have usually no direct effect on the users' view of the system. The following are the pseudo functional requirements for SatWatch: Example:

Pseudorequirement for SatWatch

All related software associated with SatWatch, including the onboard software, will be written using Java, to comply with current company policy.

3.2.3. Levels of Description

Requirements describe a system and its interaction with the surrounding environment, such as the users, their work processes, and other systems. Most requirements analysis methods have focused on describing the system. When using use cases and scenarios, it becomes apparent, however, that it is also necessary to describe the environment in which the system will operate. First, developers usually do not initially know and understand the operating environment and need to check their understanding with the users. Second, the environment is likely to change, and thus, developers should capture all the assumptions they make about the environment. In general, there are four levels of description, which can uniformly be described with use cases. We list them below from most-general to most-specific:

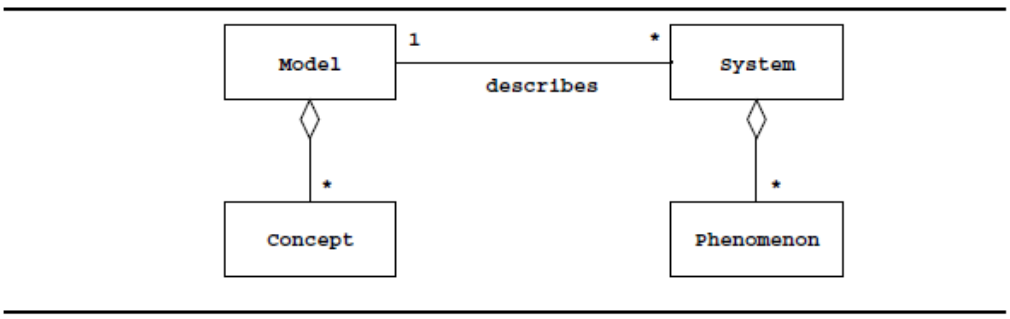


Figure 3.2 A System is a collection of real world Phenomena.

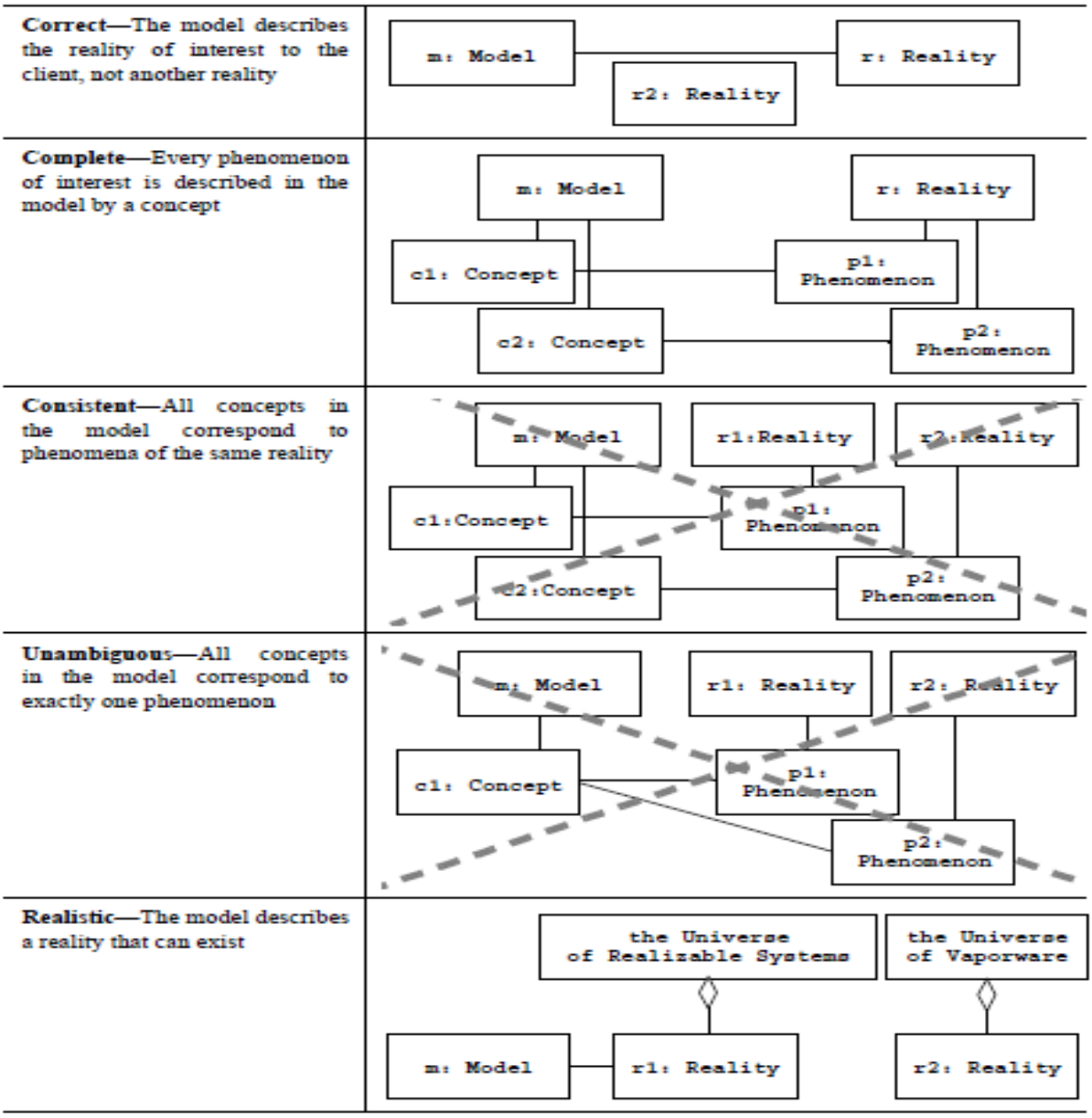
A model is a collection of concepts that represent the system's phenomena. Many models can represent different aspects of the same system. An unambiguous model corresponds to only one system.

- **Work division**. This set of use cases describes the work processes of the users that are relevant to the system. The part of the process supported by the system is also described, but the focus is on defining the boundaries between the users and the system.
- **Application-specific system functions**. This set of use cases describes the functions that the system provides that are related to the application domain.
- **Work-specific system functions**. This set of use cases describes the supporting functions of the system that are not directly related with the application domain. These include file management functions, grouping functions, undo functions, and so on. These use cases will be extended during system design, when we are discussing known boundary conditions, such as system initialization, shutdown, and exception handling policies.
- **Dialog**. This set of use cases describes the interactions between the users and the user interface of the system. The focus is on designing resolving control flow and layout issues.

3.2.4. Correctness, Completeness, Consistency, Clarity, and Realism

Requirements are continuously validated with the client and the user. Validation is a critical step in the development process, given that both the client and the developer are dependent on the system specification. Requirement validation involves checking if the specification is correct, complete, consistent, unambiguous, and realistic. A specification is **correct** if it represents the client's view of the system (i.e., everything in the requirements model accurately represents an aspect of the system). It is **complete** if all possible scenarios through the system are described, including exceptional behavior (i.e., all aspects of the system are represented in the requirements model). The system specification is **consistent** if it does not contradict itself. The system specification is **unambiguous** if exactly one system is defined (i.e., it is not possible to interpret the specification two or more different ways). Finally, it is **realistic** if the system can be implemented within constraints. These properties are illustrated with UML instance diagrams in Table 3.1.

Table 3.1 Specification properties checked during validation



The correctness and completeness of a system specification are often difficult to establish, especially before the system exists. Given that the system specification serves as a contractual basis between the client and the developers, the system specification must be carefully reviewed by both parties. Additionally, parts of the system that present a high risk should be prototyped or simulated to demonstrate their feasibility or to obtain feedback from the user. In the case of SatWatch described above, a mock-up of the watch would be built using a traditional watch and users surveyed to gather their initial impressions. A user may remark that she wants the watch to be able to display both American and European date formats.

3.2.5. Verifiability and Traceability

Two more desirable properties of a system specification are that it be verifiable and traceable. The specification is **verifiable** if, once the system is built; a repeatable test can be designed to demonstrate that the system fulfills the requirement. For example, a mean time to failure of a hundred years for SatWatch would be difficult to achieve (assuming it is realistic in the first place). The following requirements are additional examples of non-verifiable requirements:

- *The product shall have a good user interface* (good is not defined).
- *The product shall be error free* (requires large amount of resources to establish).
- *The product shall respond to the user with 1 second for most cases* ("most cases" is not defined).

A system specification is **traceable** if each system function can be traced to its corresponding set of requirements. Traceability is not a constraint on the content of the specification, but rather, on its organization. Traceability facilitates the development of tests and the systematic validation of the design against the requirements.

3.3.**Requirements Elicitation Activities**

These map a problem statement into a system specification that we represent as a set of actors, scenarios, and use cases. We discuss heuristics and methods for extracting requirements from users and modeling the system in terms of these concepts.

Requirement's elicitation activities include:

- identifying actors
- identifying scenarios
- identifying use cases
- refining use cases
- identifying relationships among use cases
- identifying participating objects
- identifying nonfunctional requirements

3.3.1. Identifying Actors

Actors represent external entities that interact with the system. An actor can be human or an external system. In the SatWatch example, the watch owner, the GPS satellites, and the WebifyWatch serial device are actors (see Figure 4-3). They all interact and exchange information with the SatWatch. Note, however, they all have specific interactions with the SatWatch: the watch

owner wears and looks at her watch; the watch monitors the signal from the GPS satellites; the WebifyWatch downloads new data into the watch. Actors define classes of functionality.

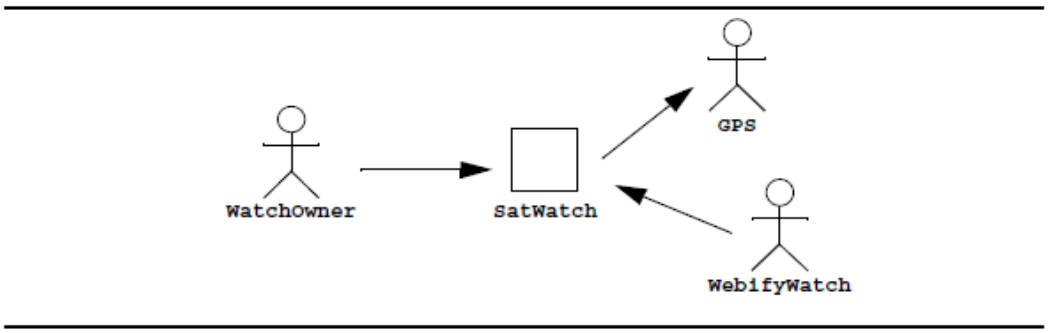


Figure 3.3 Actors for the SatWatch system.

WatchOwner moves the watch (possibly across time zones) and consults it to know what time it is. SatWatch interacts with GPS to compute its position. WebifyWatch upgrades the data contained in the watch to reflect changes in time policy (e.g., changes in daylight savings time start and end dates).

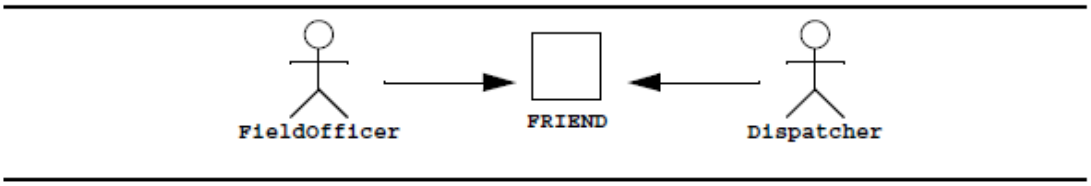


Figure 3.4. Actors of the FRIEND system.

FieldOfficers not only have access to different functionality, they use different computers to access the system.

When identifying actors, developers can ask the following questions:

Questions for identifying actors

• Which user groups are supported by the system to perform their work?

• Which user groups execute the system's main functions?

• Which user groups perform secondary functions, such as maintenance and administration?

• Will the system interact with any external hardware or software system?

In the FRIEND example, these questions lead to a long list of potential actors: Fire Fighter, Police Officer, Dispatcher, Investigator, Mayor, Governor, an EPA hazardous material database, System Administrator, and so on. We then need to consolidate this list into a small number of actors, who

are different from the point of view of the usage of the system. For example, a Fire Fighter and a Police Officer may share the same interface to the system as they are both involved with a single incident in the field. A Dispatcher, on the other hand, manages multiple concurrent incidents and requires access to a larger amount of information. The Mayor and the Governor will not likely interact directly with the system and will use the services of a trained Operator instead.

3.3.2. **Identifying Scenarios**

A scenario is "a narrative description of what people do and experience as they try to make use of computer systems and applications". A scenario is a concrete, focused, informal description of a single feature of the system from the viewpoint of a single actor. The use of scenarios in requirements elicitation is a conceptual departure from the traditional representations, which are generic and abstract. Traditional representations are centered around the system as opposed to the work that the system supports. Finally, their focus is on completeness, consistency, and accuracy, whereas scenarios are open ended and informal. A scenario-based approach cannot (and is not intended to) completely replace traditional approaches. It does, however, enhance requirements elicitation by providing a tool that is readily understandable to users and clients.

Figure 3.5 is an example of scenario for the FRIEND system, an information system for incident response. In this scenario, a police officer reports a fire and a Dispatcher initiates the incident response. Note that this scenario is concrete, in the sense that it describes a single instance. It does not attempt to describe all possible situations in which a fire incident is reported.

Scenario name	`warehouseOnFire`
Participating actor instances	`bob, alice: FieldOfficer` `john: Dispatcher`
Flow of events	1. Bob, driving down main street in his patrol car notices smoke coming out of a warehouse. His partner, Alice, activates the "Report Emergency" function from her FRIEND laptop. 2. Alice enters the address of the building, a brief description of its location (i.e., northwest corner), and an emergency level. In addition to a fire unit, she requests several paramedic units on the scene, given that the area appears to be relatively busy. She confirms her input and waits for an acknowledgment. 3. John, the `Dispatcher`, is alerted to the emergency by a beep of his workstation. He reviews the information submitted by Alice and acknowledges the report. He creates allocates a fire unit and two paramedic units to the `Incident` site and sends their estimated arrival time (ETA) to Alice. 4. Alice receives the acknowledgment and the ETA.

Figure 3.5 warehouseOnFire scenarios for the ReportEmergency use case.

Questions for identifying scenarios

• What are the tasks that the actor wants the system to perform?

• What information does the actor access? Who creates that data? Can it be modified or removed? By whom?

• Which external changes does the actor need to inform the system about? How often? When?

• Which events does the actor need to be informed by the system about? With what latency?

In the FRIEND example, we may identify four scenarios that span the type of tasks the system is expected to support:

- WarehouseOnFire (Figure 3.5): A fire is detected in a warehouse; two field officers arrive at the scene and request resources.
- FenderBender. A car accident without casualties occurs on the highway. Police officers document the incident and manage traffic while the damaged vehicles are towed away.
- CatInATree. A cat is stuck in a tree. A fire truck is called to retrieve the cat. Because the incident is low priority, the fire truck takes time to arrive at the scene. In the mean-time, the impatient cat owner climbs the tree, falls, and breaks a leg, requiring an ambulance to be dispatched.

- EarthQuake. An unprecedented earthquake seriously damages buildings and roads, spanning multiple incidents and triggering the activation of a statewide emergency operations plan. The governor is notified. Road damage hampers incident response.

3.3.3. Identifying Use Cases

A scenario is an instance of a use case, that is, a use case specifies all possible scenarios for a given piece of functionality. A use case is initiated by an actor. After its initiation, a use case may interact with other actors as well. A use case represents a complete flow of events through the system in the sense that it describes a series of related interactions that result from the initiation of the use case.

Use case name	`ReportEmergency`
Participating actor	Initiated by `FieldOfficer` Communicates with `Dispatcher`
Entry condition	1. The `FieldOfficer` activates the "Report Emergency" function of her terminal.
Flow of events	2. FRIEND responds by presenting a form to the officer. 3. The `FieldOfficer` fills the form, by selecting the emergency level, type, location, and brief description of the situation. The `FieldOfficer` also describes possible responses to the emergency situation. Once the form is completed, the `FieldOfficer` submits the form, at which point the `Dispatcher` is notified. 4. The `Dispatcher` reviews the submitted information and creates an `Incident` in the database by invoking the `OpenIncident` use case. The `Dispatcher` selects a response and acknowledges the emergency report.
Exit condition	5. The `FieldOfficer` receives the acknowledgment and the selected response.
Special requirements	The `FieldOfficer`'s report is acknowledged within 30 seconds. The selected response arrives no later than 30 seconds after it is sent by the `Dispatcher`.

Figure 3.6 An example of use case: ReportEmergency.

3.3.4. Refining Use Cases

Figure 3.7 is a refined version of the ReportEmergency use case. It has been extended to include details about the type of incidents that are known to FRIEND, detailed interactions indicating how the Dispatcher acknowledges the FieldOfficer. The use of scenarios and use cases to define the functionality of the system aims at creating requirements that are validated by the user early in the development. As the design and implementation of the system starts, the cost of changing the system specification and adding new unforeseen functionality increases. Although requirements change until late in the development, developers and users should strive to address most requirements issues early. This entails lots of changes and experimentation during requirements elicitation. Note that many use cases are rewritten several times, others substantially refined, and yet others completely dropped. In order to save time, a lot of the exploration work can be done

using scenarios and user interface mock-ups. The following heuristics can be used for writing scenarios and use cases.

Location	*Use case description*
FieldOfficer station	1. The `FieldOfficer` activates the "Report Emergency" function of her terminal. 2. FRIEND responds by presenting a form to the officer. The form includes an emergency type menu (General emergency, fire, transportation), a location, incident description, resource request, and hazardous material fields. 3. The `FieldOfficer` fills the form by specifying minimally the emergency type and description fields. The `FieldOfficer` may also describes possible responses to the emergency situation and request specific resources. Once the form is completed, the `FieldOfficer` submits the form by pressing the "Send Report" button, at which point the `Dispatcher` is notified.
Dispatcher station	4. The `Dispatcher` is notified of a new incident report by a popup dialog. The `Dispatcher` reviews the submitted information and creates an `Incident` in the database by invoking the `OpenIncident` use case. All the information contained in the `FieldOfficer`'s form is automatically included in the `Incident`. The `Dispatcher` selects a response by allocating resources to the `Incident` (with the `AllocateResources` use case) and acknowledges the emergency report by sending a short message to the `FieldOfficer`.
FieldOfficer station	5. The `FieldOfficer` receives the acknowledgment and the selected response.

Figure 3.7 Refined description for the ReportEmergency use case.

3.3.5. Identifying Relationships among Actors and Use Cases

Even medium-sized systems have many use cases. Relationships among actors and use cases enable the developers and users to reduce the complexity of the model and increase its understandability. We use communication relationships between actors and use cases to describe the system in layers of functionality. We use extend relationships to separate exceptional and common flows of events. We use include relationships to reduce redundancy among use cases.

Communication relationships between actors and use cases

Communication relationships between actors and use cases represent the flow of information during the use case. The actor who initiates the use case should be distinguished from the other actors with whom the use case communicates. Thus, access control (i.e., which actor has access to which class functionality) can be represented at this level. The relationships between actors and use cases are identified when use cases are identified. Figure 3.8 depicts an example of communication relationships in the case of the FRIEND system.

Extend relationships between use cases

A use case extends another use case if the extended use case may include the behavior of the extension under certain conditions. In the FRIEND example, assume that the connection between

the FieldOfficer station and the Dispatcher station is broken while the FieldOfficer is filling the form (e.g., the FieldOfficer's car enters a tunnel). The FieldOfficer station needs to notify the FieldOfficer that his form was not delivered and what measures he should take. The ConnectionDown use case is modeled as an extension of ReportEmergency (see Figure 3.9). The conditions under which the ConnectionDown use case is initiated are described in ConnectionDown as opposed to ReportEmergency. Separating exceptional and optional flow of events from the base use case has two advantages. First, the base use case becomes shorter and easier to understand. Second, the common case is distinguished from the exceptional case, which enables the developers to treat each type of functionality differently (e.g., optimize the common case for response time, optimize the exceptional case for clarity). Both the extended use case and the extensions are complete use cases of their own. They must have an entry and an end condition and be understandable by the user as an independent whole.

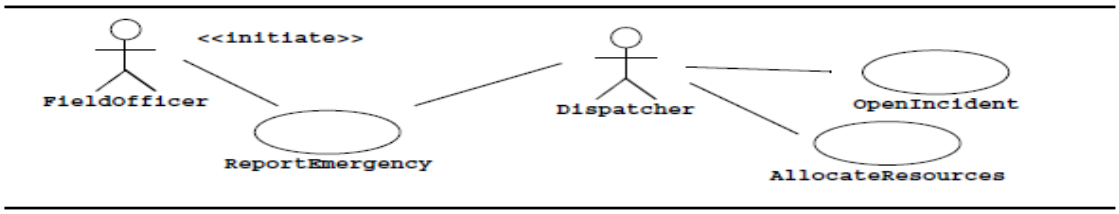


Figure 3.8 Example of communication relationships among actors and use cases in FRIEND (UML use case diagram). The FieldOfficer initiates the ReportEmergency use case and the Dispatcher initiates the OpenIncident and AllocateResources use cases. FieldOfficers cannot directly open an incident or allocate resources on their own.

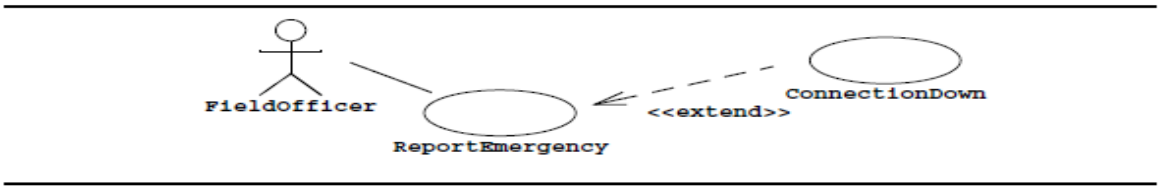


Figure 3.9 Example of use of extend relationship (UML use case diagram).

Connection Down extends the Report Emergency use case. The Report Emergency use case becomes shorter and solely focused on emergency reporting.

Include relationships between use cases

Redundancies among use cases can be factored out using include relationships. Assume, for example, that a Dispatcher needs to consult the city map when opening an incident (e.g, in order to assess which areas are at risk during a fire) and when allocating resources (e.g., to find which resources are closer to the incident). In this case, the ViewMap use case describes the flow of events required when viewing the city map and is used by both the OpenIncident and the AllocateResourcesuse cases (Figure 3.10).

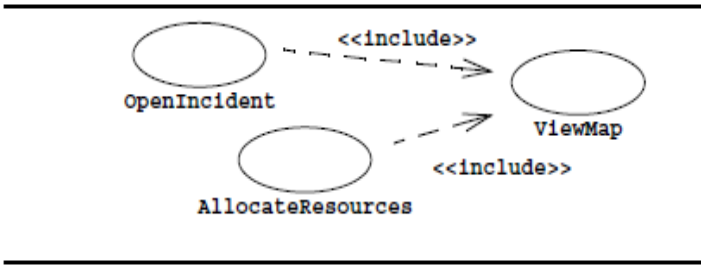


Figure 3.10 Example of include relationships among use cases.

ViewMap describes the flow of events for viewing a city map (e.g., scrolling, zooming, query by street name) and is used by both OpenIncident and AllocateResources use cases.

Extend versus include relationships

Include and extend are similar constructs, and initially it may not be clear to the developer when to use each construct [Jacobson et al., 1992]. The main distinction between these constructs is the direction of the relationship. In the case of an include relationship, the conditions under which the target use case is initiated are described in the initiating use case, as an event in the flow of events. In the case of an extend relationship, the conditions under which the extension is initiated are described in the extension as an entry condition. Figure 3.11 shows the ConnectionDown example described with an include relationship (left column) and with an extend relationship (right column). In the left column, we need to insert text in two places in the event flow where the ConnectionDown use case can be invoked. Also, if additional exceptional situations are described (e.g., a Help function on the FieldOfficer station), the ReportEmergency use case will have to be modified and will become cluttered with conditions.

3.3.6. Identifying Initial Analysis Objects

The identification of participating objects results in the initial analysis model. The identification of participating objects during requirements elicitation only constitutes a first step toward the complete analysis model. The complete analysis model is usually not used as means of

communication between users and developers, as users are often unfamiliar with objectoriented concepts. However, the description of the objects (i.e., the definitions of the terms in the glossary) and their attributes are visible to the users and reviewed.

During requirements elicitation, participating objects are generated for each use case. If two use cases refer to the same concept, the corresponding object should be the same. If two objects share the same name and do not correspond to the same concept, one or both concepts are renamed to acknowledge and emphasize their difference. This consolidation eliminates any ambiguity in the terminology used. For example, Table 3.2 depicts the initial participating objects we identified for the ReportEmergency use case.

3.3.7. Identifying Non-Functional Requirements

Nonfunctional requirements describe user-visible aspects of the system that are not directly related to the functional behavior of the system. Nonfunctional requirements span a number of issues, from user interface look and feel to response time requirements to security issues. Nonfunctional requirements are defined at the same time as functional requirements are, because they have as much impact on the development and cost of the system.

For example, consider a mosaic display that an air traffic controller uses to track planes. A mosaic display system complies data from a series of radar and databases (hence the term “mosaic”) into a summary display indicating all aircraft in a certain area, including their identification, speed, and altitude. The number of aircraft such a system can display constrains the performance of the air traffic controller and the cost of the system. If the system can only handle a few aircraft simultaneously, the system cannot be used at busy airports. On the other hand, a system able to handle a large number of aircraft is more costly and more complex to build.

Nonfunctional requirements can be elicited by investigating the following issues.

- **User interface and human factors**. What kind of interface should the system provide? What is the level of expertise of the users?
- **Documentation**. What level of document is required? Should only user documentation be provided? Should there be technical documentation for maintainers? Should the development process be documented?
- **Hardware considerations**. Are there hardware compatibility requirements? Will the system interact with other hardware systems?

- **Performance characteristics**. How responsive should the system be? How many concurrent users should it support? What is a typical or extreme load?
- **Error handling and extreme conditions**. How should the system handle exceptions? Which exceptions should the system handle? What is the worse environment in which the system is expected to perform? Are there safety requirements on the system?
- **Quality issues**. How reliable/available/robust should the system be? What is the client's involvement in assessing the quality of the system or the development process?
- **System modifications**. What is the anticipated scope of future changes? Who will perform the changes?
- **Physical environment**. Where will the system be deployed? Are there external factors such as weather conditions that the system should withstand?
- **Security issues**. Should the system be protected against external intrusions or against malicious users? To what level?
- **Resource issues**. What are the constraints on the resources consumed by the system?

Once all nonfunctional requirements are identified and described, they are prioritized by importance. Although most nonfunctional requirements are highly desirable, some of them need to be met in order for the system to operate correctly.

3.4. **Managing Requirements Elicitation**

In the previous section, we described the technical issues of modeling a system in terms of use cases. Use case modeling by itself, however, does not constitute requirements elicitation. Even after they become expert use case modelers, developers still need to elicit requirements from the users and converge onto an agreement with the client. In this section, we describe methods for eliciting information from the users and negotiating an agreement with a client. In particular, we describe:

3.4.1. Eliciting Information from Users: Knowledge Analysis of Tasks

Initially, task analysis was not concerned with requirements or system design. Task analysis was used to identify how people should be trained. In the United States, the military was primarily interested in task analysis to decrease the cost of training. In the United Kingdom, the Department of Trade and Industry was interested in task analysis for developing methods to enable people to move across industries. More recently, task analysis has become important in the field of Human

Computer Interaction (HCI) for identifying and describing the user tasks that a system should support.

Task analysis is based on the assumption that it is inefficient to ask users to describe what they do and how they do it. Users usually do not think explicitly about the sequence of tasks that are required to accomplish their work as they have often repeated these tasks many times. Users, when ask how they accomplish their work, would describe, at best, how they are supposed to accomplish it, which may be far from reality. Consequently, task analysis uses observation as an alternative to build an initial task model. This initial task model is then refined by asking the users *why* they accomplish a task a certain way.

KAT can be summarized by the five following steps:

1. **Identifying objects and actions.** Object and actions associated with objects are identified using similar techniques as object identification in object-oriented analysis, such as analyzing textbooks, manuals, rule books, reports, interviewing the task performer, observing the task performer.
2. **Identifying procedures.** A procedure is a set of actions, a precondition necessary to triggering the procedure, and a postcondition. Actions may be partially ordered. Procedures are identified by writing scenarios, observing the task performer, asking the task performer to select and order cards on which individual actions are written.
3. **Identifying goals and sub-goals**. A goal is a state to be achieved for the task to be successful. Goals are identified through interview during the performance of a task or afterward. Sub-goals are identified by decomposing goals.
4. **Identifying typicality and importance.** Each identified element is rated according to how frequently it is encountered and whether it is necessary for accomplishing a goal.
5. **Constructing a model of the task.** The information gathered above is generalized to account for common features across tasks. Corresponding goals, procedures, and objects are related using a textual notation or a graph. Finally, the model is validated with the task performer.

Although task analysis and KAT are not requirements elicitation methods per se (they do not produce a description of the future software system), they can greatly benefit the requirements elicitation activity in several ways:

- During elicitation, they provide techniques for eliciting and describing application domain knowledge, including information such as typicality and importance of specific actions; the end result is understandable by the task performer.
- When defining the boundaries of a system, task models assist in determining which parts of the task should remain manual and which parts should be automated; moreover, the task model may reveal problem areas in the current system.
- When designing the interface of the system, task models serve as a source of inspiration for metaphors understandable by the user.

3.4.2. Negotiating specifications with clients: Joint Application Design

Joint Application Design (JAD) is effectiveness lies in that the requirements elicitation work is done in one single workshop session in which all stakeholders participate. Users, clients, developers, and a trained session leader sit together in one room to present their viewpoint, listen to other viewpoints, negotiate, and agree on a mutually acceptable solution. The outcome of the workshop, the final JAD document, is a complete system specification document that includes definitions of data elements, work flows, and interface screens. Because the final document is jointly developed by the stakeholders (that is, the participants who not only have an interest in the success of the project, but also can make substantial decisions) the final JAD document represents an agreement between users, clients, and developers, and thus minimizes requirements changes later in the development process.

3.4.3. **Validating Requirements: Usability Testing**

Usability testing tests the user understands of the use case model. Usability testing finds problems with the system specification by letting the user explore the system or only part of the system (e.g., the user interface). Usability tests are also concerned with user interface details, such as the look and feel of the user interface, the geometrical layout of the screens, and the hardware. For example, in case of a wearable computer, a usability test might test the ability of the user to issue commands to the system while lying in an awkward position, as in the case of a mechanic looking at a screen under a car while checking a muffler.

The technique for conducting usability tests is based on the classical approach for conducting a controlled experiment. Developers formulate a test objective, which they then test by manipulating selected experimental parameters under controlled conditions. Developers carefully study the values of these parameters to identify statistically significant cause-and-effect relationships. Even

though this type of approach could be used for any parameter, usability tests focus on usability parameters, such as the ease to learn the system, the time to accomplish a task, or the rate of errors a user makes when accomplishing a task.

There are two important differences between classical experiments and usability tests. Whereas the classical experimental method is designed to confirm or refute a hypothesis, the goal of usability tests is to obtain qualitative information on how to fix usability problems and how to improve the system. Another difference is the rigor with which the experiments need to be performed. It has been shown that even a series of quick focused tests starting as early as requirements elicitation is extremely helpful. Nielsen uses the term "discount usability engineering" to indicate that a few usability tests are better than none at all.

There are three types of usability tests:

- **Scenario test**. During this test, one or more users are presented with a visionary scenario of the system. Developers identify how quickly users are able to understand the scenario, how accurately it represents their model of work, and how positively they react to the description of the new system. The selected scenarios should be as realistic and detailed as possible. A scenario test allows rapid and frequent feedback from the user. Scenario tests can be realized as paper mockups2 or with a simple prototyping environment, which is often easier to learn than the programming environment used for development. The advantage of scenario tests is that they are cheap to realize and to repeat. The disadvantages are that the user cannot interact directly with the system and that the data are fixed.
- **Prototype test**. During this type of test, one or more users are presented with a piece of software that practically implements the system. A vertical prototype implements a complete use case through the system, and a horizontal prototype presents an interface for most use cases (without providing much or any functionality). The advantages of prototype tests are that they provide a realistic view of the system to the user and that prototypes can be instrumented to collect detailed data. The disadvantages of prototypes are that they are expensive to build and to modify.
- **Product test.** This test is similar to the prototype test except that a functional version of the system is used in place of the prototype. A product test can only be conducted once most of the system is developed. It also requires that the system be easily modifiable such that the results of the usability test can be taken into account.

In all three types of tests, the basic elements of usability testing include:

- Development of test objectives
- Use of a representative sample of end users
- **U**se of the system in the actual or simulated work environment
- Involvement of end users
- **C**ontrolled, extensive interrogation, and probing of the users by the person performing the usability test
- Collection and analysis of quantitative and qualitative results
- Recommendations on how to improve the system

3.4.4. **Documenting Requirements Elicitation**

The results of the requirements elicitation activity and the analysis activity are documented in the Requirements Analysis Document (RAD). This document completely describes the system in terms of functional and nonfunctional requirements and serves as a contractual basis between the client and the developers. The audience for the RAD includes the client, the users, the project management, the system analysts (i.e., the developers who participate in the requirements), and the system designers (i.e., the developers who participate in the system design). The first part of the document, including use cases and nonfunctional requirements, is written during requirements elicitation. The formalization of the specification in terms of object models is written during analysis.

The first section of the RAD is an *Introduction*. Its purpose is to provide a brief overview of the function of the system and the reasons for its development, its scope, and references to the development context (e.g., related problem statement, references to existing systems, feasibility studies). The introduction also includes the objectives and success criteria of the project.

The second section, *Current system,* describes the current state of affairs. If the new system will replace an existing system, this section describes the functionality and the problems of the current system. Otherwise, this section describes how the tasks supported by the new system are accomplished now. For example, in the case of SatWatch, the user currently resets her watch whenever she travels across a time zone. Because of the manual nature of this operation, the user occasionally sets the wrong time. In contrast, the SatWatch will continually ensure accurate time within its lifetime. In the case of FRIEND, the current system is paper based: Dispatchers keep track of resource assignments by filling forms. Communication between dispatchers and field

officers is radio based. The current system requires a high documentation and management cost that the FRIEND system aims to reduce.

The third section, *proposed system,* documents the requirements elicitation and the analysis model of the new system. It is divided into five subsections:

- *Overview* presents a functional overview of the system.
- *Functional requirements* describe in natural language the high-level functionality of the system.
- *A nonfunctional requirement describes* user-level requirements that are not directly related to functionality. This includes performance, security, modifiability, error handling, hardware platform, and physical environment.
- *Pseudo-requirements* describe design and implementation constraints imposed by the client. This includes the specification of the deployment platform, implementation language, or database management system.
- *System models* describe the scenarios, use cases, object model, and dynamic models describing the system. This section contains the complete functional specification of the system, including mock-ups and navigational charts illustrating the user interface of the system.

The following is an example template for a RAD:

Requirements Analysis Document
1. Introduction
   - 1.1 Purpose of the system
   - 1.2 Scope of the system
   - 1.3 Objectives and success criteria of the project
   - 1.4 Definitions, acronyms, and abbreviations
   - 1.5 References
   - 1.6 Overview
2. Current system
3. Proposed system
   - 3.1 Overview
   - 3.2 Functional requirements
   - 3.3 Nonfunctional requirements
     - 3.3.1 User interface and human factors
     - 3.3.2 Documentation
     - 3.3.3 Hardware consideration
     - 3.3.4 Performance characteristics
     - 3.3.5 Error handling and extreme conditions
     - 3.3.6 Quality issues
     - 3.3.7 System modifications
     - 3.3.8 Physical environment
     - 3.3.9 Security issues
     - 3.3.10Resource issues
   - 3.4 Pseudorequirements
   - 3.5 System models
     - 3.5.1 Scenarios
     - 3.5.2 Use case model
     - 3.5.3 Object model
       - 3.5.3.1 Data dictionary
       - 3.5.3.2 Class diagrams
     - 3.5.4 Dynamic models
     - 3.5.5 User interface—navigational paths and screen mock-ups
4. Glossary

Summary

- Requirement elicitation focuses on describing the purpose of the system. **Identifying actors** ,during this activity developers identify the different types of users the future system will provide. In **identifying scenarios** developers observe users and develop a set of detailed scenarios for typical functionality provide by the future system. Once developers and users agree on a set of scenarios, developers derive from the scenarios a set of use cases that completely represent the future system. This is performed in **identifying use case** activity. When describing use case, developers determine the scope of the system. During the activity **refining a use case** the developers ensure that the system specification is complete. The activity where developers consolidate the use case model by eliminating redundancies and ensure the system specification is consistent is **identifying relationship among use case.** During **identifying nonfunctional requirements,** clients and developers agree on aspects that are visible to the user but not directly related functionality. Documents, Users and Legacy systems are the different source of information in requirement elicitation. Joint Application

Design(JAD) is a means of bringing together the key users, managers, and system analysts involved in the analyst of a current system. Knowledge Analysis of Tasks (KAT) focuses on eliciting information from users through observation. Usability testing focuses on validation the requirements elicitation model with the user through a variety of methods. The Reason for conducting JAD session in a location away from the people normally work is to keep them away from all distractions, so that they can concentrate full on system analysis.

- **Functional requirements** describe the interactions between the system and its environment independent of its implementation. **Nonfunctional requirements** describe user-visible aspects of the system that are not directly related with the functional behavior of the system. **Pseudo requirements** are requirements imposed by the client that restrict the implementation of the system.
- Work division describes the work process of the users that are relevant to the system. Application-specific system functions describe functions that the system provides that are related to the application domain. Work-specific function describes the supporting functions of the system that are not directly related with the application domain. Dialog describes the interaction between the users and the user interface of the system. System specification should be correct, complete, consistent and realistic. The result of the requirements elicitation activity and the analysis activity are documented in the Requirement Analysis Document (RAD)

Review Questions & Problems

1. From the Activities of Requirement Elicitation which one focuses on identifying the functionality that the system provides to its user?
   A. Identifying actors
   B. Identifying Scenarios
   C. Identifying Use Cases
   D. None
2. What are the different sources of information during requirement elicitation?
   A. Different documents provided by the users
   B. Manuals and technical documentation of legacy system
   C. Users
   D. All
3. _____ describes requirements that are imposed by the user.
   A. Functional requirement
   B. Non Functional requirement

C. Pseudo requirement
D. None

4. If a system specification does not contradict itself. It is _____.
   A. Complete
   B. Consistent
   C. Clear
   D. Realistic
5. Debre Tabor University has developed a prototype for Student Information System. The university before deploying the functional system wants to test the visionary scenarios of the SIS prototype.
   What kind of usability testing is the University is conducting?
   A. Scenario test
   B. Prototype test
   C. Project test
6. From the following questions which one can be used to identify use case
   A. Will the system interact with any external hardware or software system?
   B. What are the tasks that the actor wants the system to perform
   C. Which user groups execute the system's main functions?
   D. None
7. _____ is the study of a business problem for the purpose of recommending improvements and specifying the business requirements for the solution.(Ans. System Analysis)

References:

- *Brahmin, Ali(1999), Object oriented System development , McGraw Hill, USA*
- *OOSE Practical software development using UML and Java*
- *OOSE 8th edition Stephen R. Schach*

Chapter 4: Software Project management

Unit Description

This unit deals with the roles of software project managers, Organization & team Structure project planning, and the organization of SPMP document. To address these contents Brainstorming, peer & group discussion and interactive lecture will be used more. Question & answer, group work assessments are among the methods to be used.

Brainstorming

What do you think about software project managers?

Project management is concerned with planning and allocating resources to ensure the delivery of **quality software system on time and within budget.** Complex products require a large number of participants with diverse skills and backgrounds. Competitive markets and evolving requirements introduce change in the development, triggering frequent resource reallocation and making it difficult to track the status of the project.

4.1. Responsibility of Software Project Managers

Project management is concerned with planning and allocating resources to ensure the delivery of quality software system on time and within budget. Project management is subject to the same barriers as technical activities: complexity and change. Complex products require a large number of participants with diverse skills and backgrounds. Competitive markets and evolving requirements introduce change in the development, triggering frequent resource reallocation and making it difficult to track the status of the project. Manager's deal with complexity and change the same way developers do: through

- Modeling
- Communication
- Rationale
- And configuration management

Management models allow representing the resources available to the project, the constraints resources are subjected to, and the relationships among resources

Management Concept

Teams

A team is a small group of people who work on the same sub problem. Each team is responsible for the analysis, design, and implementation of its subsystem. Each team negotiates with other teams the interfaces of the subsystems it needs. Such teams are called subsystem teams

Role

Once the project is organized into teams, we need to define how team members are assigned individual responsibilities. A role defines the types of technical and managerial tasks that are expected from a person. We distinguish between five types of roles:

- **Management roles** are concerned with the organization and execution of the project within constraints. These roles include the project manager and the team leader. Development roles are concerned with specifying, designing, and constructing subsystems. These roles include the analyst, the system architect, the object designer, and the implementer.
- **Cross-functional roles** are concerned with coordinating more than one team.
- **Consultant roles** are concerned with providing temporary support in areas where the project participants lack expertise. The users and the client act in most projects as consultants on the problem domain. Technical consultants may also bring expertise on new technologies or methods.
- **Promoter roles** are concerned with promoting changes through an organization. Once the organization is set up, the task plan in place, and the project is running in steady state, it becomes increasingly difficult to introduce changes in the project. Promoters are roles for overcoming barriers to change.

Work Product

Assigning a role to a participant usually translates into assigning the responsibility of a specific work product. A **work product** is an artifact that is produced during the development, such as an internal document for other project participants or a deliverable to the customer.

An analyst is responsible for a section of the RAD describing the functionality of the system. A system architect is responsible for the SDD describing the system's high-level architecture. A project manager is responsible for the SPMP describing the development processes. Finally, the system and its accompanying documentation also constitute a set of work products that are delivered to the client.

Task

A role describes work in terms of a set of tasks. A task includes a description, duration, and is assigned to a role. A task represents an atomic unit of work that can be managed: A manager assigns it to a developer, the developer carries it out, and the manager monitors the progress and completion of the task. Tasks consume resources, result in work products, and depend on work products produced by other tasks.

Schedule

A schedule is the mapping of tasks onto time: Each task is assigned planned start and end times. This allows us to plan the deadlines for individual deliverables. The two most often used diagrammatic notations for schedules are PERT and Gantt charts. A **Gantt chart** is a compact way to present the schedule of a software project along the time axis.

Project Management Activities

We focus on the following management activities:

- **Project initiation** including defining the problem, identifying the initial task plan, and allocating resources to tasks

- **Steady state**, including project monitoring, risk management, and the project agreement
- **Project termination**, including the client acceptance test and installation

Group Discussion

Discuss on Project Planning

4.2. Project Planning

- Probably the most time-consuming project management activity.
- Continuous activity from initial concept through to system delivery. Plans must be regularly revised as new information becomes available.
- Various different types of plans may be developed to support the main software project plan that is concerned with schedule and budget.

Plan	Description
Quality plan	Describes the quality procedures and standards that will be used in a project.
Validation plan	Describes the approach, resources and schedule used for system validation.
Configuration management plan	Describes the configuration management procedures and structures to be used.
Maintenance plan	Predicts the maintenance requirements of the system, maintenance costs and effort required.
Staff development plan.	Describes how the skills and experience of the project team members will be developed.

The project plan sets out:

- The resources available to the project;
- The work breakdown;
- A schedule for the work.

Project Plan Structure

- Introduction.
- Project organisation.
- Risk analysis.
- Hardware and software resource requirements.
- Work breakdown.
- Project schedule.
- Monitoring and reporting mechanisms.

4.3. The organization of SPMP document

The management models are documented in the SPMP [IEEE Std. 1058.1-1993]. The audience of the SPMP includes the management and the developers. The SPMP documents all issues related to client requirements (such as deliverables and acceptance criteria), the project goals, the

project organization, and the division of labor into tasks, and the allocation resources and responsibilities. An example of SPMP template:

Software Project Management Plan (SPMP)

1. Introduction
    - 1.1 Project overview
    - 1.2 Project deliverables
    - 1.3 Evolution of this document
    - 1.4 References
    - 1.5 Definitions and acronyms
2. Project organization
    - 2.1 Process model
    - 2.2 Organizational structure
    - 2.3 Organizational boundaries and interfaces
    - 2.4 Project responsibilities
3. Managerial process
    - 3.1 Management objectives and priorities
    - 3.2 Assumptions, dependencies and constraints
    - 3.3 Risk management
    - 3.4 Monitoring and controlling mechanisms
4. Technical process
    - 4.1 Methods, tools, and techniques
    - 4.2 Software documentation
    - 4.3 Project support functions
5. Work elements, schedule, and budget

4.4. Project Size Estimation Metrics

Software Metrics

- Measure -a quantitative unit of the object in which you are interested / an attribute of an entity of interest.
- Measurement- the activity of determining the measure.
- Metrics- a quantitative measure built out of the different measures that represent the entity of interest.
- Importance
    - Enable to measure in quantitative terms the different aspects of software that need evaluation on an ongoing basis for estimation.
- Software Metrics can be used to indicate the basic attributes of the software. The indicative metrics, not exhaustive, are
- Size - Line of code ,Function point count
- Quality- Reliability, Accuracy, Dependability, and these are measured through some measures like errors, failures and number of runs between failures ,etc.
- Execution Time –process time of execution of a critical process and many more.
- The IEEE Standard Glossary of Software Engineering Terms Defines metrics as

"Quantitative measure of the degree to which a system, component or process possesses a given attribute."

Software Metrics have two categories,

1. Direct Measure - cost, effort, Loc, response time, resource usage
2. Indirect Measure - functionality, quality, complexity, efficiency, dependability, maintainability, ease of use, quality of documentation, etc.

   Metrics based on direct measure are easy to establish, as they are more tangible, and quantifiable, whereas metrics based on indirect measures are difficult to establish.

Attributes of Effective Software Metrics

- Simple to learn
- Easy to compute
- Unambiguous
- Clear in objective terms of application
- Independent of technology, architecture or programming languages.

Software Size Estimation

- The first step in software estimation is assessing the size of the software proposed for development.
- The size estimate is then used to compute development efforts and resource estimate, cost, and development time.
- The size of the software is directly linked to requirement specifications.

Requirement specifications should be broken into the following structure :

- Functional Requirements
- Non Functional Requirements
- Features
- Facilities
- Interfaces with other Systems

- The software estimation is best handled taking the views of two key participants in the development of requirement specifications: the user and the software engineer as a developer.
- The user view is functional and developer view is technical.
- The technical view in the early stages of the development cycle is difficult but a functional view is possible with a fair accuracy.
- A fair assumption would be the more the functions, features and facilities, the larger the size of the software.
- To generate a function-dependent size estimate, researchers suggested a method based on function points. Function point is a measure of the functionality.

Group Discussion

What are the techniques to estimate projects?

4.5. Project Estimation Technique

Function Point Analysis

- Function points allow the measurement of software size in standard units, independent of the underlying language in which the software is developed.
- Counting the number of externals (inputs, outputs, inquiries, and interfaces) that make up the system.
- There are five types of externals to count:

1. External inputs - data or control inputs (input files, tables, forms, screens, messages, etc.) to the system

2. External outputs - data or control outputs from the system

3. External inquiries - I/O queries which require a response (prompts, interrupts, calls, etc.)

4. External interfaces - libraries or programs which are passed into and out of the system (I/O routines, sorting procedures, math libraries, run-time libraries, etc.)

5. Internal data files - groupings of data stored internally in the system (entities, internal control files, directories)

Apply these steps to calculate the size of a project:

1. Count or estimate all the occurrences of each type of external.

2. Assign each occurrence a complexity weight.

3. Multiply each occurrence by its complexity weight, and total the results to obtain a function count. Complexity weights are as listed below.

	Complexity		
Description	Low	Medium	High
External inputs	3	4	6
External Outputs	4	5	7
External Inquiries	3	4	6
External Interfaces	5	7	10
Internal Files	7	10	15

Function Total	Count	Weight	Inputs
Outputs	**8**	**4**	**32**
Inquiries	**12**	5	60
Data files	4	4	16
Interfaces	2	10	20
	1	7	7
		Total	135

4. Multiply the function count by a value adjustment multiplier (VAM) to obtain the function point count.

- The VAM is calculated as follows:
- Where Vi is a rating of 0 to 5 for each of the following fourteen factors (i). The rating reflects how each factor affects the software size.

1. Data communications
2. Distributed functions
3. Performance
4. Heavily used operational configuration
5. Transaction rate
6. On-line data entry
7. Design for end user efficiency
8. On-line update of logical internal files
9. Complex processing
10. Reusability of system code
11. Installation ease
12. Operational ease
13. Multiple sites
14. Ease of change

- Assign the rating of 0 to 5 according to these values:

0 - factor not present or has no influence
1 - Insignificant influence
2 - Moderate influence
3 - Average influence
4 - Significant influence
5- Strong influence

	Complexity		
Description	Low	Medium	High
External inputs	3	4	6
External Outputs	4	5	7
External Inquiries	3	4	6
External Interfaces	5	7	10
Internal Files	7	10	15

4.6. Scheduling, Organization and Team Structures

Project-task scheduling is an important project planning activity. It involves deciding which tasks would be taken up when. In order to schedule the project activities, a software project manager needs to do the following:

- Identify all the tasks needed to complete the project and Break down large tasks into small activities.
- Determine the dependency among different activities and Establish the most likely estimates for the time durations necessary to complete the activities and Allocate resources to activities.
- Plan the starting and ending dates for various activities and Determine the critical path. A critical path is the chain of activities that determines the duration of the project.

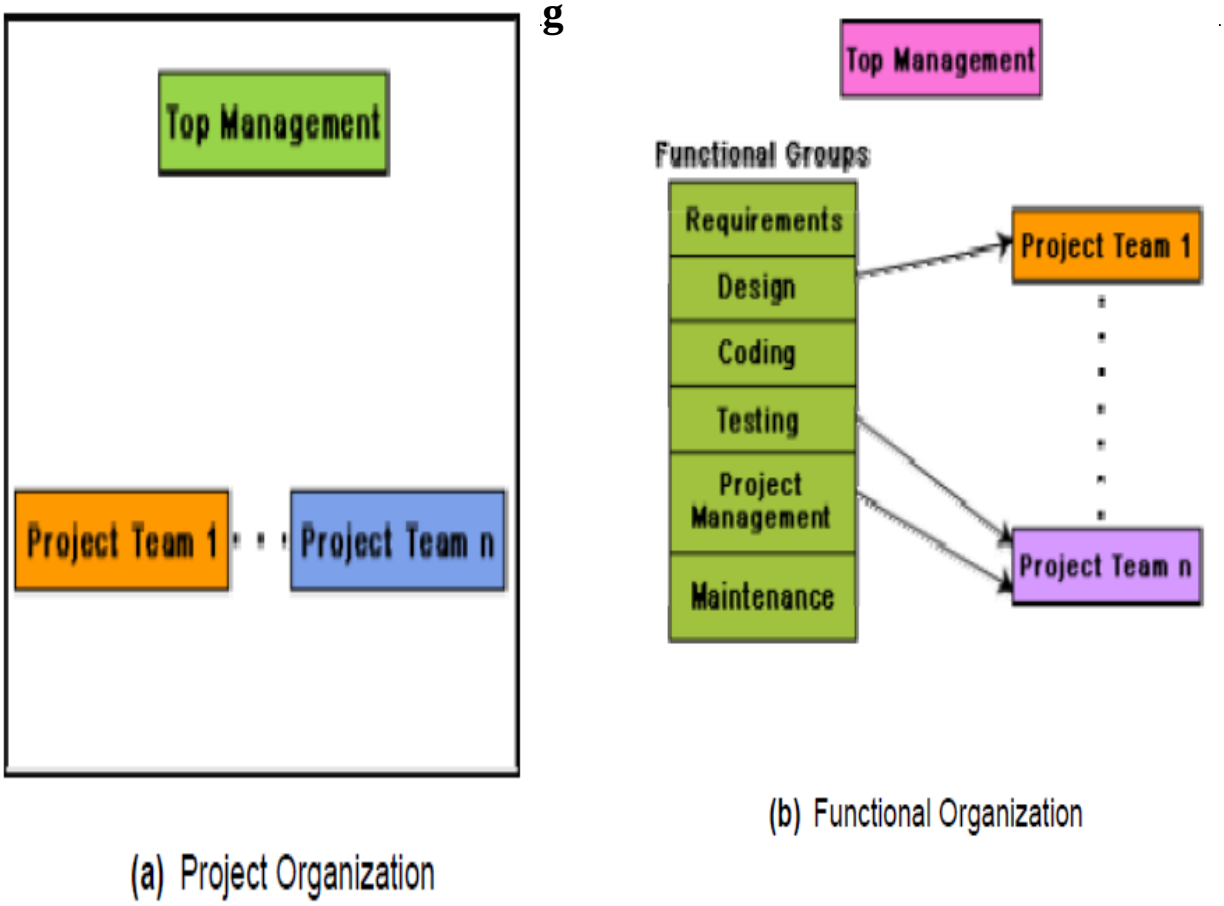
Organization structure

- Usually every software development organization handles several projects at any time.
- Software organizations assign different teams of engineers to handle different software projects.
- Each type of organization structure has its own advantages and disadvantages so the issue "how is the organization as a whole structured?" must be taken into consideration so that each software project can be finished before its deadline.

Functional format vs. project format

- There are essentially two broad ways in which a software development organization can be structured:
- Functional format and project format. In the project format, the project development staffs are divided based on the project for which they work (as shown in next fig).
- In the functional format, the development staffs are divided based on the functional group to which they belong.

g



(a) Project Organization

(b) Functional Organization

Team Structure

- Team structure addresses the issue of organization of the individual project teams. There are some possible ways in which the individual project teams can be organized.
- Problems of different complexities and sizes often require different team structures for chief solution.
- The most common types of team structure are:
  - Chief Programmer Team
  - Democratic Team
  - Mixed Control Team Organization

4.7. Risk Management

- Risk management is concerned with identifying risks and drawing up plans to minimise their effect on a project.
- A risk is a probability that some adverse circumstance will occur
  - Project risks affect schedule or resources;
  - Product risks affect the quality or performance of the software being developed;
  - Business risks affect the organisation developing or procuring the software.

Risk Management Process

- Risk identification- Identify project, product and business risks;
- Risk analysis -Assess the likelihood and consequences of these risks;

- Risk planning- Draw up plans to avoid or minimise the effects of the risk;
- Risk monitoring- Monitor the risks throughout the project.

Software Risks

Risk	Affects	Description
Staff turnover	Project	Experienced staff will leave the project before it is finished.
Management change	Project	There will be a change of organisational management with different priorities.
Hardware unavailability	Project	Hardware that is essential for the project will not be delivered on schedule.
Requirements change	Project and product	There will be a larger number of changes to the requirements than anticipated.
Specification delays	Project and product	Specifications of essential interfaces are not available on schedule
Size underestimate	Project and product	The size of the system has been underestimated.
CASE tool under-performance	Product	CASE tools which support the project do not perform as anticipated
Technology change	Business	The underlying technology on which the system is built is superseded by new technology.
Product competition	Business	A competitive product is marketed before the system is completed.

Risk type	Possible risks
Technology	The database used in the system cannot process as many transactions per second as expected. Software components that should be reused contain defects that limit their functionality.
People	It is impossible to recruit staff with the skills required. Key staff are ill and unavailable at critical times. Required training for staff is not available.

Organizational	The organization is restructured so that different management are responsible for the project. Organizational financial problems force reductions in the project budget.
Tools	The code generated by CASE tools is inefficient. CASE tools cannot be integrated.
Requirements	Changes to requirements that require major design rework are proposed. Customers fail to understand the impact of requirements changes.
Estimation	The time required to develop the software is underestimated. The rate of defect repair is underestimated. The size of the software is underestimated.

Risk Analysis.

Risk	Probability	Effects
Organizational financial problems force reductions in the project budget.	Low	Catastrophic
It is impossible to recruit staff with the skills required for the project.	High	Catastrophic
Key staff is ill at critical times in the project.	Moderate	Serious
Software components that should be reused contain defects which limit their functionality.	Moderate	Serious
Changes to requirements that require major design rework are proposed.	Moderate	Serious
The organization is restructured so that different management is responsible for the project.	High	Serious

Risk Management Strategy

Risk	Strategy
Organizational financial problems	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business.

Recruitment problems	Alert customer of potential difficulties and the possibility of delays, investigate buying-in components.
Staff illness	Reorganize team so that there is more overlap of work and people therefore understand each other's jobs.
Defective components	Replace potentially defective components with bought-in components of known reliability.

4.8. Quality Assurance Monitoring Plans

Quality assurance consists of the auditing and reporting functions of management. The goal of quality assurance is to provide management with the data necessary to be informed about product quality, thereby gaining insight and confidence that product quality is meeting its goals. Of course, if the data provided through quality assurance identify problems, it is management's responsibility to address the problems and apply the necessary resources to resolve quality issues.

Cost Quality

The cost of quality includes all costs incurred in the pursuit of quality or in performing quality-related activities. Cost of quality studies are conducted to provide a base-
Quality costs may be divided into costs associated with prevention, appraisal, and failure. Prevention costs include

- quality planning, formal technical reviews, test equipment and training

Appraisal costs include activities to gain insight into product condition the "first time through" each process. Examples of appraisal costs include

- in-process and inter process inspection, equipment calibration and maintenance and testing

Failure costs are those that would disappear if no defects appeared before shipping a product to customers. Failure costs may be subdivided into internal failure costs and external failure costs. Internal failure costs are incurred when we detect a defect in our product prior to shipment. Internal failure costs include

- rework, repair, and failure mode analysis

External failure costs are associated with defects found after the product has been shipped to the customer.

Sample questions

- why quality assurance
- write down at two risk types and explain them briefly

Chapter 5: Analysis

OBJECTIVES

After the completion of this chapter students will be able to:

- ⇨ Understand OO analysis process.
- ⇨ Perform the analysis workflow
- ⇨ Extract the boundary, control and entity classes
- ⇨ Perform functional modeling and class modeling

INTRODUCTION

Analysis focuses on producing a model of the system, called the analysis model, which is correct, complete, consistent, and verifiable. Analysis is different from requirements elicitation in that developers focus on structuring and formalizing the requirements elicited from users

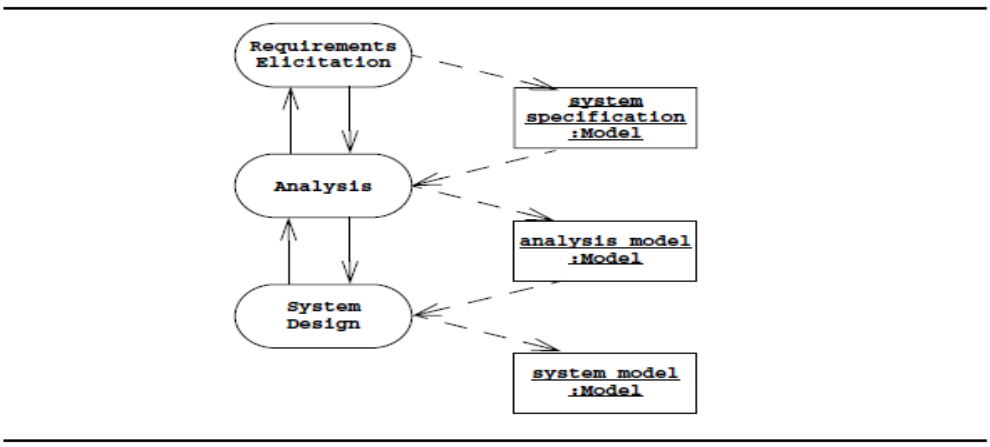


Figure 4.1 Products of requirements elicitation and analysis (UML activity diagram).

Althoughtheanalysismodelmaynotbeunderstandabletotheusersandtheclient, it helps developers verify the system specification produced during requirements elicitation. There is a natural tendency for users and developers to postpone difficult decisions until later in the project. A decision may be difficult because of lack of domain knowledge, lack of technological knowledge, or simply because of disagreements among users and developers. Postponing decisions enables the project to move on smoothly and avoid confrontation with reality or peers. Unfortunately, difficult decisions will eventually need to be made, often at higher cost when intrinsic problems are discovered during testing, or worse, during user evaluation. Translating a system specification in to a formal or semi-formal model forces developers to identify and resolve difficult issues early in the development.

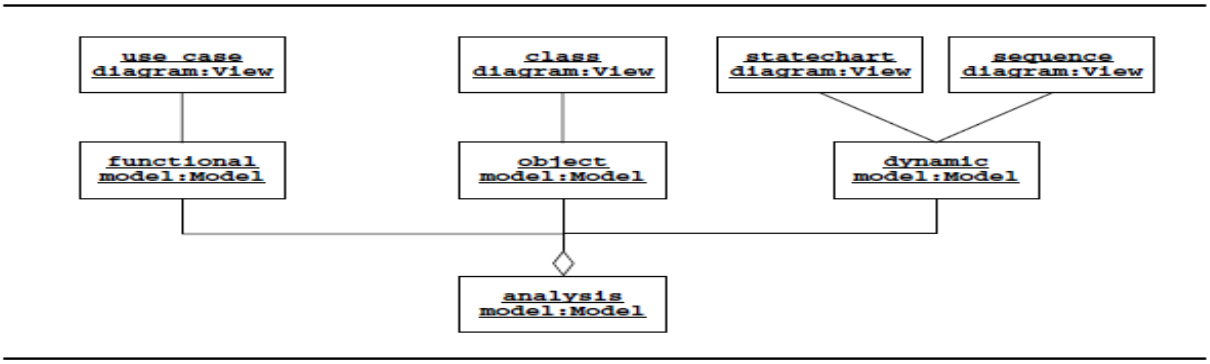


Figure 4.2 Theanalysismodeliscomposedofthefunctionalmodel,theobjectmodel,andthedynamic model.

In UML, the functional model is represented with use case diagrams, the object model with class diagrams, and the dynamic model with statechart and sequence diagrams.

5.1. ANALYSIS CONCEPTS

5.1.1. Entity, Boundary, and Control Objects

The analysis object model consists of entity, boundary, and control objects. **Entity objects** represent the persistent information tracked by the system. **Boundary objects** represent the interactions between the actors and the system. **Control objects** represent the tasks that a formed by the user and supported by the system. In the 2B watch example, Year, Month, Day are entity objects; Button Boundary and LCD Display Boundary are boundary objects; Change Date Control is a control object that represents the activity of changing the date by pressing combinations of buttons.

Modeling the system with entity, boundary, and control objects has several advantages. First, it provides developers with simple heuristics to distinguish different, but related concepts. For example, the time that is tracked by a watch has different properties than the display that depictsthetime.Differentiatingbetweenboundaryandentityobjectsforcesthatdistinction:Thetimethat istrackedbythewatchisrepresentedbytheTimeobject.Thedisplayisrepresentedby the LCD Display Boundary. Second, the three object type approach results in smaller and more specializedobjects.Third,thethreeobjecttypeapproachleadstomodelsthataremoreresilient to change: Theinterfacetothesystem(representedbytheboundaryobjects)ismorelikelyto change than its basic functionality (represented by the entity and control objects).

To distinguish between different types of objects, UML provides the stereotype mechanism to enable the developer to attach such metainformation to modeling elements. For example, in Figure5-4,weattachthe<<control>>stereotypetotheChangeDateControlobject.Inaddition tostereotypes,wemayalsousenamingconventionsforclarityandrecommenddistinguishingthe three different types of objects on a syntactical basis: Boundary objects may have the suffix Boundary appended to their name; control objects may have the suffix Control appended to their name;entityobjectsusuallydonothaveanysuffixappendedtotheirname.

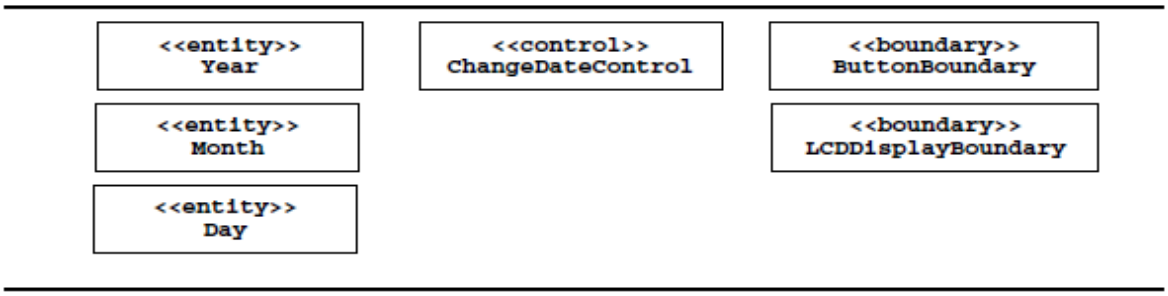


Figure 4.3 Analysis classes for the2Bwatchexample.

5.1.2. Association Multiplicity Revisited

The end of an association can be labeled by a set of integers called **multiplicity**. The multiplicity indicates the number of links that can legitimately originate from an instance of the class connected to the association end. For example, in Figure 4.4, a 2Bwatchhas exactly two Buttons and one LCD Display.

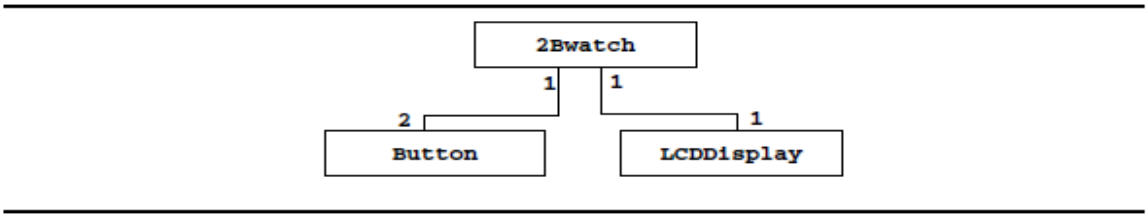


Figure 4.4 An example of multiplicity of associations (UML class diagram).

A2Bwatchhastwobuttons and one LCD Display.

In UML, an association end can have an arbitrary set of integers as a multiplicity. For example, an association could allow only a prime number of links and, thus, would have a multiplicity 1, 2, 3, 5, 7, 11, 13, ... In practice, however, most of the associations we encounter belong to one of the following three types (see Figure 4.5).

- A**one-to-oneassociation**hasamultiplicity1oneachend.Aone-to-oneassociation betweentwoclasses(e.g.,PoliceOfficerandBadgeNumber),meansthatexactlyone linkexistsbetweeninstancesofeachclass(e.g.,aPoliceOfficerhasexactlyone BadgeNumber, and a BadgeNumberdenotes exactly one PoliceOfficer).
- A**one-to-manyassociation**hasamultiplicity1ononeendand0….n(alsorepresentedby astar)or1...nontheother.Aone-to-manyassociationbetweentwoclasses(e.g.,Person and Car) denotes composition (e.g., a FireUnitowns one or more FireTrucks, a FireTruckis owned exactly by one FireUnit).
- A**many-to-manyassociation**hasamultiplicity0...nor1...nonbothends.Amany-to- many associationbetween two classes (e.g., FieldOfficer and IncidentReport) denotes that an arbitrary number of links can exist between instances of the two classes (e.g.,aFieldOfficercanwriteformanyIncidentReport,anIncidentReportcanbe written by many FieldOfficers).This is the most complex type of association.

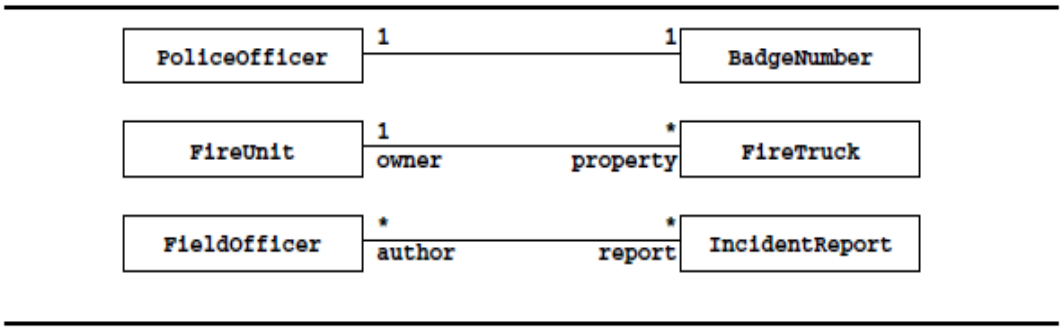


Figure 4.5 Examples ofmultiplicity (UML class diagram).

5.1.3. Qualified Associations

Qualificationisatechniqueforreducingmultiplicitybyusingkeys.Associationswitha 0…1 or 1 multiplicity are easier to understand than associations with a 0...n or 1…n multiplicity. Often, in the case of a one-to-many association, objects on the "many" side can be distinguishedfromoneanotherusinganame.For-example,inahierarchicalfile system, each filebelongstoexactlyonedirectory.Eachfileisuniquelyidentifiedbyanameinthecontextof adirectory.Manyfilescanhavethesamenameinthecontextofthefilesystem;however,two files cannotsharethesamenamewithinthesamedirectory.Withoutqualification(seetopof Figure 4.8), the association between Directory and File has a one multiplicity on the Directorysideandazero-to-

manymultiplicityontheFileside.Wereducethemultiplicityon theFilesidebyusingthefilenameattributeasakey,alsocalleda**qualifier**(seetopof Figure 4.8).The relationship betweenDirectoryand Fileis called a **qualifiedassociation**.

5.1.4. Generalization

Generalization enables us to organize concepts in to hierarchies. There may be any number of intermediate levels in between, covering more-or-less generalized concepts (e.g., Low Priority Incident, Emergency, Disaster). Such hierarchies allow us to refer to many concepts precisely. When we use the term Incident, we mean all instances of all types of Incidents. When we use the term Emergency, we only refer to an Incident that requires an immediate response. This view of generalization stems from modeling.

In an object-oriented programming language, inheritance is a reusability technique. If a class Child inherits from a class Parent, all the attributes and methods available on Parent are automatically available on Child. The class Child may add additional methods or override inherited methods, thus refining the class Parent. As in the case of generalization, the class at the top of the hierarchy tends to be the most general one, whereas the leaves tend to be the most specialized.

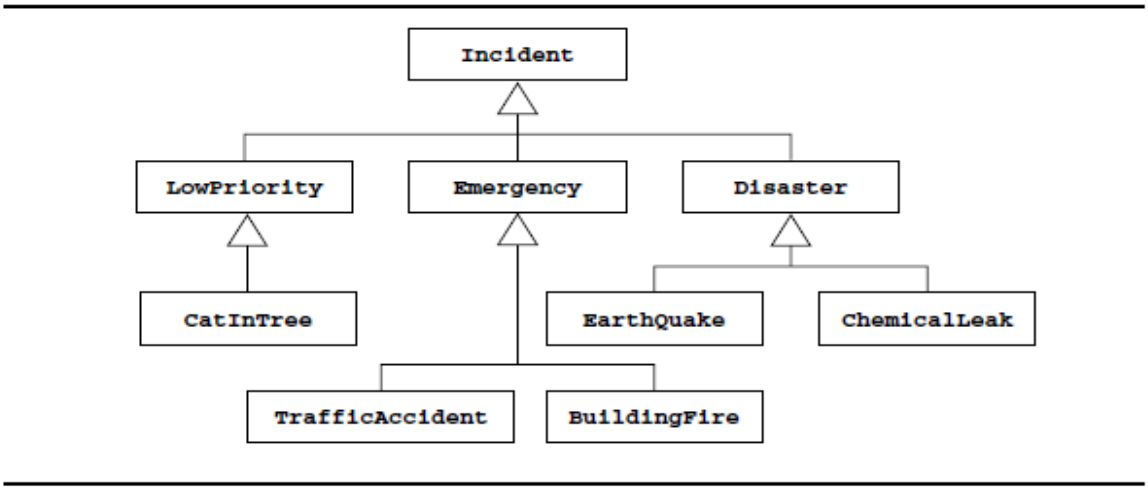


Figure 4.6 An example of a generalization hierarchy (UML class diagram).

5.2. ANALYSIS ACTIVITIES: FROM USE CASES TO OBJECTS

We illustrate each activity by focusing on the ReportEmergency use case of FRIEND . These activities are guided by heuristics. The quality of their outcome depends on the experience of the developer in applying these heuristics and methods.The methods and heuristics presented in this section area.

5.2.1. Identifying Entity Objects

Requirements Elicitation, participating objects are found by examining each use case and identifying candidate objects. Natural language analysis is an intuitive set of heuristics for identifying objects, attributes, and associations from a system specification. Abbott's heuristics maps parts of speech (e.g., nouns, having verbs, being verbs, adjectives) to model components (e.g., objects, operations, inheritance relationships, classes). Table 4-1 provides examples of such mappings by examining the Report Emergency use case.

Table 4-1 *Abbott's heuristics for mapping parts of speech to model components [Abbott, 1983]*

Part of speech	Model component	Examples
Proper noun	Object	Alice
Common noun	Class	FieldOfficer
Doing verb	Operation	Creates, submits, selects
Being verb	Inheritance	Is a kind of, is one of either
Having verb	Aggregation	Has, consists of, includes
Modal verb	Constraints	Must be
Adjective	Attribute	Incident description

Use case name	`ReportEmergency`
Entry condition	1. The `FieldOfficer` activates the "Report Emergency" function of her terminal.
Flow of events	2. FRIEND responds by presenting a form to the officer. The form includes an emergency type menu (General emergency, fire, transportation), a location, incident description, resource request, and hazardous material fields. 3. The `FieldOfficer` fills the form, by specifying minimally the emergency type and description fields. The `FieldOfficer` may also describes possible responses to the emergency situation and request specific resources. Once the form is completed, the `FieldOfficer` submits the form by pressing the "Send Report" button, at which point, the `Dispatcher` is notified. 4. The `Dispatcher` reviews the information submitted by the `FieldOfficer` and creates an `Incident` in the database by invoking the `OpenIncident` use case. All the information contained in the `FieldOfficer`'s form is automatically included in the incident. The `Dispatcher` selects a response by allocating resources to the incident (with the `AllocateResources` use case) and acknowledges the emergency report by sending a FRIENDgram to the `FieldOfficer`.
Exit condition	5. The `FieldOfficer` receives the acknowledgment and the selected response.

5.2.2. Identifying Boundary Objects

Boundary objects represent the system interface with the actors. In each use case, each actor interacts with at least one boundary object. The boundary object collects the information from the actor and translates it into an interface neutral form that can be used by the entity objects and also by the control objects.

Boundary objects model the user interface at a coarse level. They do not describe in detail the visual aspects of the user interface. For example, boundary objects such as "button" or "menu item" are too detailed. First, developers can discuss user interface details more easily with sketches and mock-ups. Second, the design of the user interface design will continue to evolve as a consequence of usability tests, even after the functional specification of the system becomes stable. Updating the analysis model every time a visual change is made to the interface is time consuming and does not yield any substantial benefit.

Table 4-2 *Entity objects for the Report Emergency use case*

Dispatcher	Police officer who manages Incidents. A Dispatcher opens, documents, and closes Incidents in response to Emergency Reports and other communication with FieldOfficers. Dispatchers are identified by badge numbers.
EmergencyReport	Initial report about an Incident from a FieldOfficer to a Dispatcher. An EmergencyReport usually triggers the creation of an Incident by the Dispatcher. An EmergencyReport is composed of a emergency level, a type (fire, road accident, or other), a location, and a description.
FieldOfficer	Police or fire officer on duty. A FieldOfficer can be allocated to, at most, one Incident at a time. FieldOfficers are identified by badge numbers.
Incident	Situation requiring attention from a FieldOfficer. An Incident may be reported in the system by a FieldOfficer or anybody else external to the system. An Incident is composed of a description, a response, a status (open, closed, documented), a location, and a number of FieldOfficers.

We find the following boundary objects by examining the Report Emergency use case (Table 4-3).

Note that the Incident Form is not explicitly mentioned anywhere in the ReportEmergency use case. We identified this object by observing that the Dispatcher needs an interface to view the emergency report submitted by the FieldOfficer and to send back an acknowledgment. The terms used for describing the boundary objects in the analysis model should follow the user terminology, even if it is tempting to use terms from the implementation domain.

Table 4-3 Boundary objects for the ReportEmergency use case

3.4.5. 4.2.3 Identifying Control Objects

AcknowledgmentNotice	Notice used for displaying the Dispatcher's acknowledgment to the FieldOfficer.
DispatcherStation	Computer used by the Dispatcher.
ReportEmergencyButton	Button used by a FieldOfficer to initiate the ReportEmergency use case.
ReportEmergencyForm	Form used for the input of the ReportEmergencyForm. This form is presented to the FieldOfficer on the FieldOfficerStation when the "Report Emergency" function is selected. The EmergencyReportForm contains fields for specifying all attributes of an emergency report and a button (or other control) for submitting the form once it is completed.
FieldOfficerStation	Mobile computer used by the FieldOfficer.
IncidentForm	Form used for the creation of Incidents. This form is presented to the Dispatcher on the DispatcherStation when the EmergencyReport is received. The Dispatcher also uses this form to allocate resources and to acknowledge the FieldOfficer's report.

Table 4-4 *Control objects for the ReportEmergency use case*

`ReportEmergencyControl`	Manages the report emergency reporting function on the `FieldOfficerStation`. This object is created when the `FieldOfficer` selects the "Report Emergency" button. It then creates an `EmergencyReportForm` and presents it to the `FieldOfficer`. After submission of the form, this object then collects the information from the form, creates an `EmergencyReport`, and forwards it to the `Dispatcher`. The control object then waits for an acknowledgment to come back from the `DispatcherStation`. When the acknowledgment is received, the `ReportEmergencyControl` object creates an `AcknowledgmentNotice` and displays it to the `FieldOfficer`.
`ManageEmergencyControl`	Manages the report emergency reporting function on the `DispatcherStation`. This object is created when an `EmergencyReport` is received. It then creates an `IncidentForm` and displays it to the `Dispatcher`. Once the `Dispatcher` has created an `Incident`, allocated `Resources`, and submitted an acknowledgment, `ManageEmergencyControl` forwards the acknowledgment to the `FieldOfficerStation`.

5.2.3. Modeling Interactions between Objects: Sequence Diagrams

A sequence diagram ties use cases with objects. It shows how the behavior of a use case (or scenario) is distributed among its participating objects. Sequence diagrams are usually not a good medium for communication with the user. They represent, however, another shift in perspective and allow the developers to find missing objects or grey areas in the system specification. In this section, we model the sequence of interactions among objects needed to realize the use case. Figures 4.11 to 4.13aresequencediagramsassociatedwiththeReportEmergencyuse case. Thecolumnsofasequencediagramrepresenttheobjectsthatparticipateintheusecase.

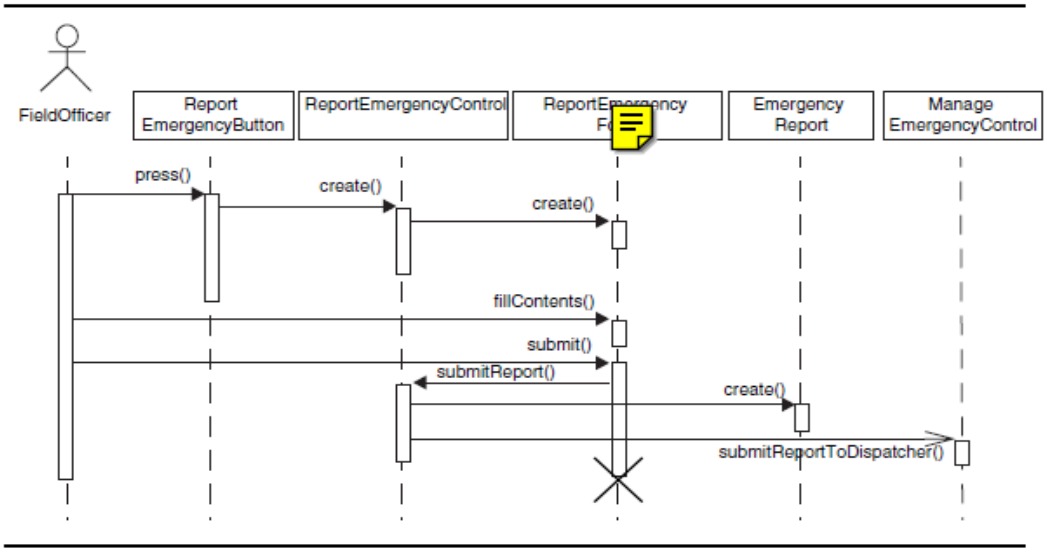


Figure 4.11 Sequence diagram for the ReportEmergency use case (initiation from the Field Officer Station side).

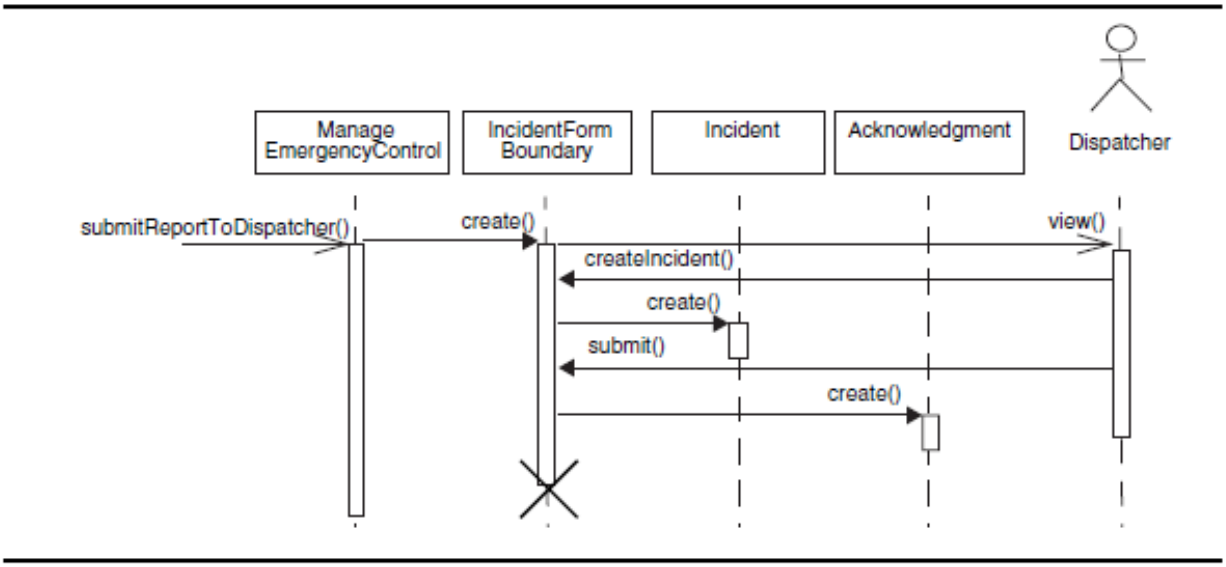


Figure 4.12 Sequence diagram for the ReportEmergency use case (DispatcherStation).

In Figure4.12, we discover the entity object Acknowledgment that we forgot during our initial examination of the ReportEmergency usecase(inTable4-2).The Acknowledgment object is different from an Acknowledgment Notice: It holds the information associated with anAcknowledgmentandiscreatedbeforetheAcknowledgmentNoticeboundaryobject.

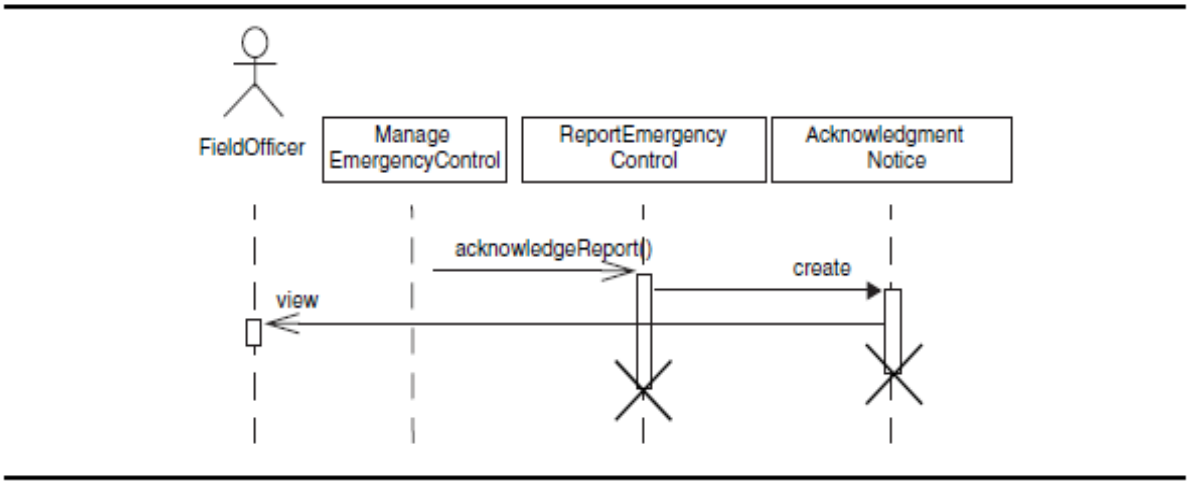


Figure 4.13 Sequence diagram for the ReportEmergency use case (acknowledgment on the FieldOfficerStation).

5.2.4. Identifying Associations

Whereassequencediagramsallowdeveloperstorepresentinteractionsamongobjectsover time, class diagrams allow developers to describe the spatial connectivity of objects.

An association shows a relationship between two or more classes. For example, a FieldOfficer writes an Emergency Report(seeFigure4.15).Identifying associations has two advantages.First,itclarifiestheanalysismodelbymakingrelationshipsbetweenobjectsexplicit (e.g.,anEmergencyReportcanbecreatedbyaFieldOfficeroraDispatcher).Second,it enables the developer to discover boundary cases associated with links. Boundary cases are exceptions that need to be clarified in the model. Forexample, it is intuitive to assume that most EmergencyReports are written by one FieldOfficer. However,should the system support EmergencyReports written by more than one? Should the system allow for anonymous EmergencyReports?

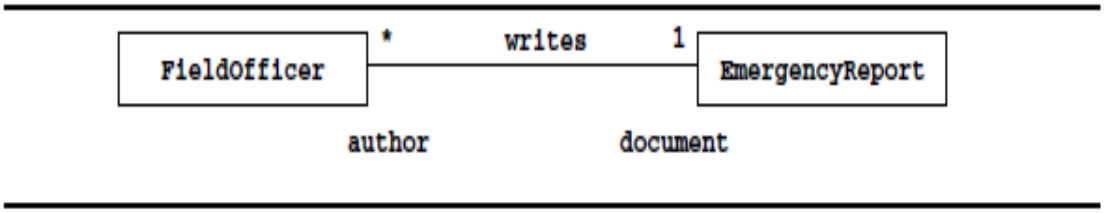


Figure 4.14 An example of association between the EmergencyReport and the FieldOfficer classes.

Associations have several properties:

- A **name**, to describe the association between the two classes (e.g., writes in Figure 4.15). Association names are optional and need not be unique globally.
- A **role** ateachend,identifyingthefunctionofeachclasswithrespecttotheassociations(e.g., authoris the role played by FieldOfficerin the Writesassociation).
- A**multiplicity**ateachend,identifyingthepossiblenumberofinstances(e.g.,*indicates aFieldOfficermaywritezeroormoreEmergencyReports,whereas1indicatesthat each EmergencyReporthas exactly one FieldOfficeras author.

Initially, the associations between entity objects are the most important, as they reveal more information aboutthe application domain. According to Abbott'sheuristics (see Table 4.1),associationscanbeidentifiedbyexaminingverbsandverbphrasesdenotingastate (e.g., *has, is part of, manages, reports to, is triggered by, is contained in, talks to, includes*). Every association should

be named and roles assigned to each end.

5.2.5. Identifying Attributes

Attributes are properties of individual objects. For example, an Emergency Report, as described in Table 4-2,has an emergency type, a location, and a description property(see Figure 4.17).TheseareenteredbyaFieldOfficerwhenshereportsanemergencyandare subsequently tracked by the system. When identifying properties of objects, only the attributes relevant to the system should be considered. For example, each Field Officer has asocial security number that is not relevant to the emergency information system. Instead, Field Officers are identified by badge number, represented by the bad geNumberproperty.

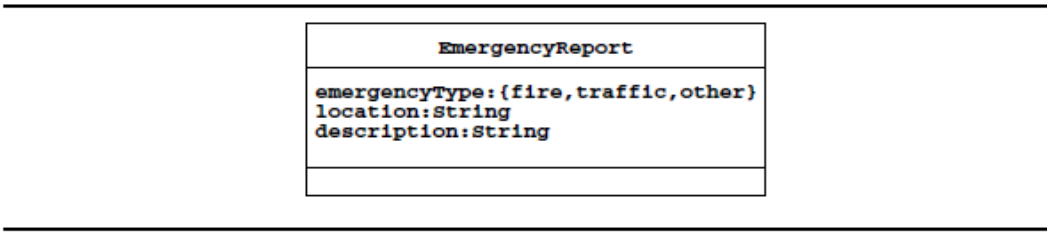


Figure 4.15 Attributes of the EmergencyReportclass.

Properties that are represented by objects are not attributes. Forexample, every EmergencyReporthasanauthorrepresentedbyanassociationtotheFieldOfficerclass. Developers should identify as many associations as possible before identifying attributes to avoid confusing attributes and objects. Attributes have:

- A**name**identifyingthemwithinanobject.Forexample,anEmergencyReportmayhavea reportTypeattributeandanemergencyTypeattribute.ThereportTypedescribesthe kindofreportbeingfiled (e.g.,initialreport,requestforresource,final report).The emergencyTypedescribes thetypeof emergency(e.g., fire,traffic,other).To avoid confusion, these attributes should not be both calledtype.
- A brief **description**.
- A**type**describingthelegalvaluesitcantake.Forexample,thedescriptionattributeof anEmergencyReportisastring.TheemergencyTypeattributeisanenumerationthatcan take one of three values: fire, traffic, other.

5.2.6. Modeling the Mon trivial Behavior of Individual Objects

Sequence diagrams are used to distribute behavior across objects and to identify operations. Sequence diagrams represent the behavior of the system from the perspective of a single use case.

Statechart diagramsrepresentbehaviorfromtheperspectiveofasingleobject. Viewing behavior from the perspective of each object enables the developer, on the one hand, to identify missing use cases, and, on the other hand, to build a more-formal description of the behavior of the object. Note that it is not necessary to build statecharts for every class in the system. Only the statecharts of objects with an extended lifespan and nontrivial behavior are worth constructing.

Figure 4.18displays astatechart for the Incident class. The examinationof this statechart may help the developer check if there are use cases for documenting, closing, and archivingIncidents.NotethatFigure 4.18isahigh-levelstatechartanddoesnotmodelthe statechangesanIncidentgoesthroughwhileitisactive(e.g.,whenresourcesareassignedto it). Such behavior can be modeled by associating a nested statechart with theActivestate.

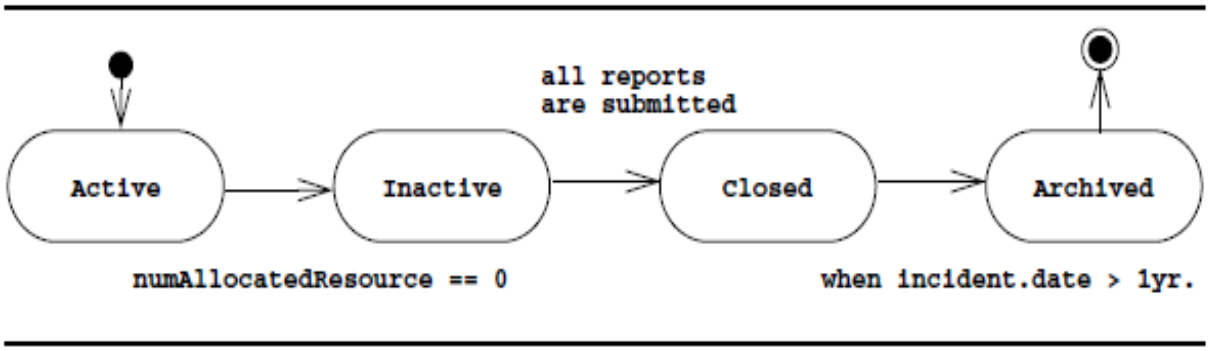


Figure 4.16 UML state chart for Incident.

5.2.7. Modeling Generalization Relationships between Objects

Generalization is used to eliminate redundancy from the analysis model. If two or more classes share attributes or behavior, the similarities are consolidated into a super-class. For example, Dispatchers and Field Officers both have abadgeNumber attribute that serves to identify them with inacity.FieldOfficers and Dispatchers are both PoliceOfficers who are assigned different functions. To model explicitly this similarity, we introduce an abstract *PoliceOfficer*classfromwhichtheFieldOfficerandDispatcherclassesinherit(see Figure 4.19).

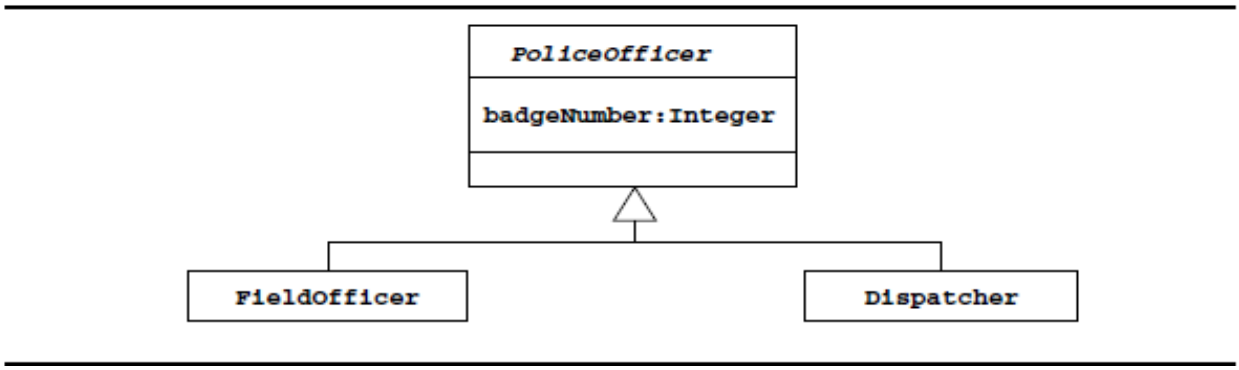


Figure 4.17 An example of inheritance relationship (UML class diagram).

5.2.8. Reviewing the Analysis Model

The analysis model is built incrementally and iteratively. The analysis model is seldom correct or even complete on the first pass. Several iterations with the client and the user are necessarybeforetheanalysismodelconvergestowardacorrectspecification usable by the developers for proceeding to design and implementation. For example, an omission discovered duringanalysiswillleadtoaddingorextendingausecaseinthesystemspecification, which may lead to eliciting more information from the user.

Once the analysis model becomes stable (i.e., when the number of changes to the model are minimal and the scope of the changes localized),the analysis model is reviewed, first by the developers(i.e.,internalreviews),thenjointlybythedevelopersandtheclient. The goal of the review is to make sure that the system specification is correct, complete, consistent, and realistic. Notethatdeveloperssshouldbepreparedtodiscovererrorsdown-streamandmakechangestothe specification. Itis,however,aworthwhileinvestmenttocatchasmanyrequirementserrors upstream. Thereviewcanbefacilitatedbyachecklistoralistofquestions.Belowareexample questions adapted from.

The following questions should be asked to ensure that the model is*correct*:

- Is the glossary of entity objects understandable by the user?
- Do abstract classes correspond to user-level concepts?
- Are all descriptions in accordance with the users'definitions?
- Do all entity and boundary objects have meaningful noun phrases as names?
- Do all use cases and control objects have meaningful verb phrases as names?

- Are all error cases described and handled?
- Are the start-up and the shut-down phases of the system described?
- Are the administration functions of the system described?

The following questions should be asked to ensure that the model is*complete*:

- Foreachobject:Isitneededbyanyusecase?Inwhichusecaseisitcreated?Modified? Destroyed?
- Can it be accessed from an boundary object?
- For each attribute: When is it set? What is its type? Should it be a qualifier?
- Foreachassociation:Whenisittraversed?Whywasthespecificmultiplicitychosen?Can associations with one-to-many and many-to-many multiplicities be qualified?
- Foreachcontrolobject:Doesithavethenecessaryassociationstoaccesstheobjects participating in its corresponding use case?

The following questions should be asked to ensure that the model is *consistent*:

- Are there multiple classes or use cases with the same name?
- Do entities (e.g., use cases, classes, attributes) with similar names denote similar phenomena?
- Are all entities described at the same level of detail?
- Are there objects with similar attributes and associations that are not in the same generalization hierarchy?

Thefollowingquestionsshouldbeaskedtoensurethatthesystemdescribedbytheanalysis model is *realistic*:

- Are there any no features in the system? Were there any studies or prototypes built to ensure their feasibility?
- Can the performance and reliability requirements be met? Were these requirements verified by any prototypes running on the selected hardware?

Summary

Analysis focuses on producing a model of the system, called the analysis model, which is correct, complete, consistent, and verifiable. The analysis model is composed of three individual models: The **functional model,** represent by use case and scenarios; The **analysis**

object model, represented by class and object diagrams; The **dynamic model,** represented by state chart and sequence diagrams

Entity objects represent the persistent information tracked by the system. **Boundary objects** represent the interactions between the actors and the system. **Control objects** represents the tasks that are performed by the user and supported by the system. **Qualification** is a technique for reducing multiplicity by using keys. **Generalization** enables us to organize concepts into hierarchies.

Abbott's heuristics for mapping parts of speech to model components can be used to identify objects, relationships, attributes from system specification documents.

Review Questions & Problems

1. What is the end result of system analysis?
2. What is the purpose of Generalization in System Analysis?
3. The analysis object model is represented by
    A) Use cases & Scenarios
    B) Class and objects
    C) State chart and sequence

References:

- *Brahmin, Ali(1999), Object oriented System development , McGraw Hill, USA*
- *OOSE Practical software development using UML and Java*
- *OOSE 8th edition Stephen R. Schach*

Chapter 6: Object Oriented System Design

OBJECTIVES

After the completion of this chapter students will be able to:

- ⇨ Define OO design process.
- ⇨ Understand OO design management.

INTRODUCTION

Analysis results in the requirements model described by the following products:

- A set of *nonfunctional requirements* and *constraints,* such as maximum response time, minimum throughput, reliability, operating system platform, and so on
- A *use case model,* describing the functionality of the system from the actors' point of view
- An *object model,* describing the entities manipulated by the system
- A *sequence diagram* for each use case, showing the sequence of interactions among objects participating in the use case.

The analysis model describes the system completely from the actors' point of view and serves as the basis of communication between the client and the developers. The analysis model, however, does not contain information about the internal structure of the system, its hardware configuration, or, more generally, how the system should be realized. System design is the first step in this direction. System design results in the following products:

- List of *design goals,* describing the qualities of the system that developers should optimize
- *Software architecture*, describing the subsystem decomposition in terms of subsystem responsibilities, dependencies among subsystems, subsystem mapping to hardware, and major policy decisions such as control flow, access control, and data storage.

The design goals are derived from the nonfunctional requirements. Design goals guide the decisions to be made by the developers especially when trade-offs are needed. The subsystem decomposition constitutes the bulk of system design. Developers divide the system into manageable pieces to deal with complexity: Each subsystem is assigned to a team and realized independently. In order for this to be possible, though, developers need to address system-wide issues when decomposing the system. In particular, they need to address the following issues:

- *Hardware/software mapping:* What is the hardware configuration of the system? Which node is responsible for which functionality? How is communication between nodes realized? Which services are realized using existing software components? How are these components encapsulated? Addressing hardware/software mapping issues often leads to the definition of *additional subsystems* dedicated to moving data from one node to another, dealing with concurrency, and reliability issues. Off-the-shelf components enable developers to realize complex services more economically. User interface packages and database management systems are prime examples of off-the-shelf components. Components, however, should be encapsulated to minimize dependencies on a particular component: A competing vendor may come up with a better component.
- *Data management:* Which data need to be persistent? Where should persistent data be stored? How are they accessed? Persistent data represents a bottleneck in the system on many different fronts: Most functionality in system is concerned with creating or manipulating persistent data. For this reason, access to the data should be fast and reliable. If retrieving data is slow, the whole system will be slow. If data corruption is likely, complete system failure is likely. These issues need to be addressed consistently at the system level. Often, this leads to the selection of a database management system and of an *additional subsystem* dedicated to the management of persistent data.
- *Access control:* Who can access which data? Can access control change dynamically? How is access control specified and realized? Access control and security are system-wide issues. The access control must be consistent across the system; in other words, the policy used to specify who can and cannot access certain data should be the same *across all subsystems*.
- *Control flow:* How does the system sequence operations? Is the system event driven? Can it handle more than one user interaction at a time? The choice of control flow has an impact on the interfaces of subsystems. If an event-driven control flow is selected, subsystems will provide event handlers. If threads are selected, subsystems need to guarantee mutual exclusion in critical sections.
- *Boundary conditions:* How is the system initialized? How is it shut down? How are exceptional cases detected and handled? System initialization and shutdown often represent the larger part of the complexity of a system, especially in a distributed environment. Initialization, shutdown, and exception handling have an impact on the interface of *all subsystem*s.

Figure 6.1 depicts the activities of system design. Each activity addresses one of the issues we described above. Addressing any one of these issues can lead to changes in the subsystem decomposition and to raising new issues. As you will see when we describe each of these activities, system design is a highly iterative activity, constantly leading to the identification of new subsystems, the modification of existing subsystems, and system-wide revisions that impact all subsystems. But first, let us describe the concept of subsystem in more detail.

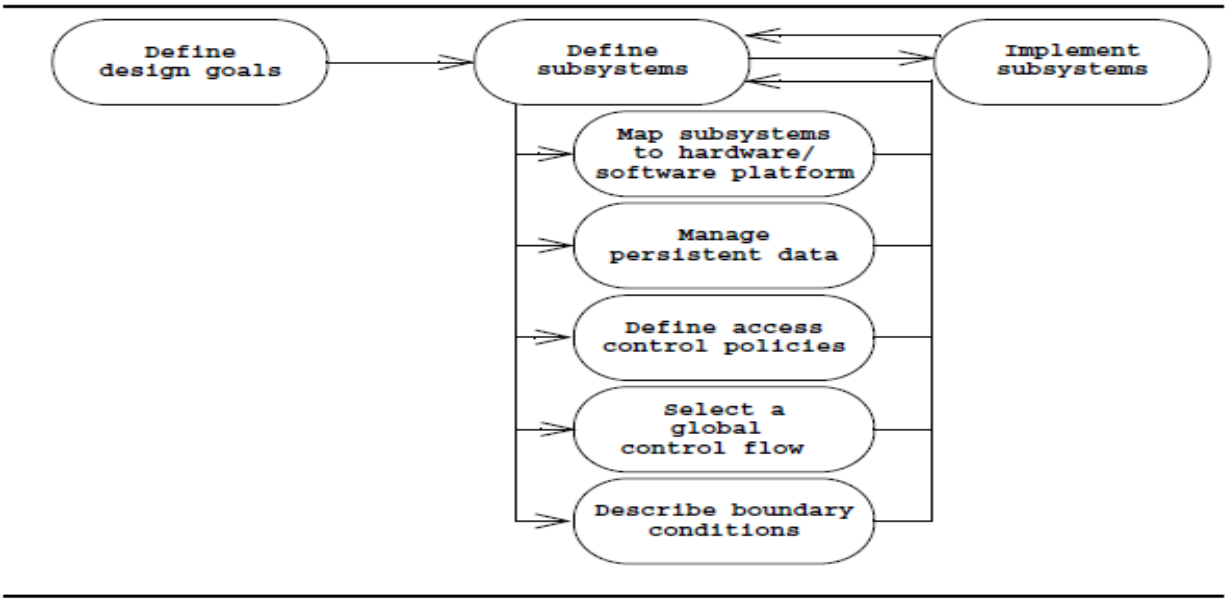


Figure 6.1 The activities of system design (UML activity diagram).

6.1. System design concepts

6.1.1. Subsystems and classes

In order to reduce the complexity of the application domain, we identified smaller parts called classes and organized them into packages. Similarly, to reduce the complexity of the solution domain, we decompose a system into simpler parts, called subsystems, which are made of a number of solution domain classes. In the case of complex subsystems, we recursively apply this principle and decompose a subsystem into simpler subsystems (see Figure 6.2).

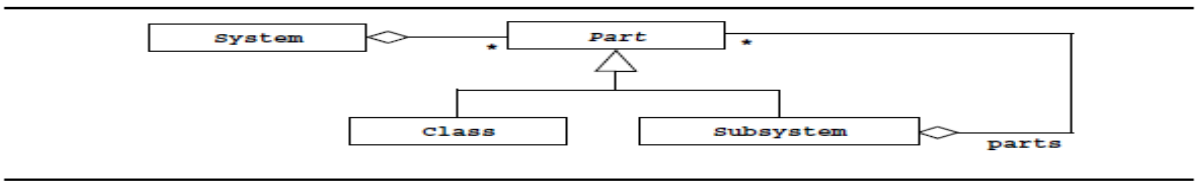


Figure 6.2 Subsystem decomposition (UML class diagram).

For example, the accident management system we previously described can be decomposed into a DispatcherInterface subsystem, implementing the user interface for the Dispatcher; a FieldOfficerInterfacesubsystem, implementing the user interface for the FieldOfficer; an IncidentManagement subsystem, implementing the creation, modification, and storage of Incidents; and a Notification subsystem, implementing the communication between FieldOfficer terminals and Dispatcher stations. This subsystem decomposition is depicted in Figure 6.3 using UML packages.

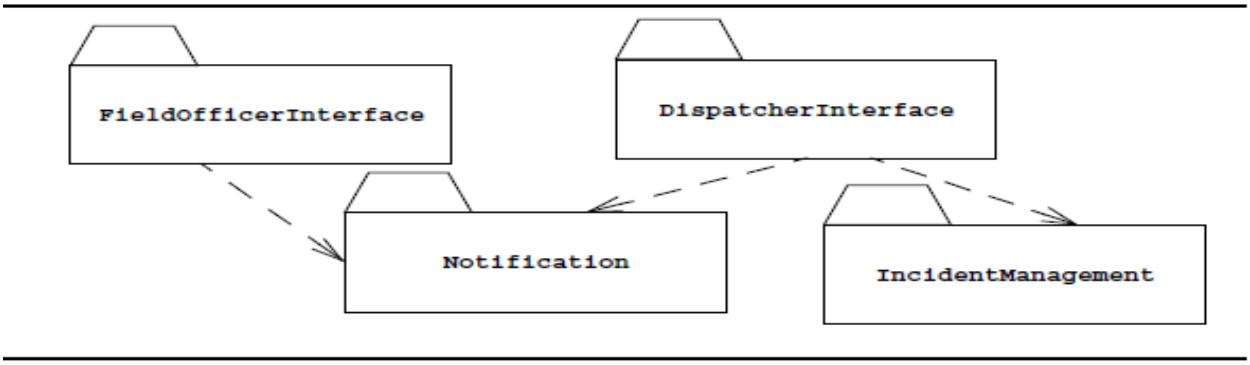


Figure 6.3 Subsystem decomposition for an accident management system (UML class diagram, collapsed view). Subsystems are shown as UML packages. Dashed arrows indicate dependencies between subsystems.

6.1.2. Services and subsystem interfaces

A subsystem is characterized by the **services** it provides to other subsystems. A service is a set of related operations that share a common purpose. A subsystem providing a notification service, for example, defines operations to send notices, look up notification channels, and subscribe and unsubscribe to a channel.

The set of operations of a subsystem that are available to other subsystems form the **subsystem interface**. The subsystem interface, also referred to as the application programmer interface (API), includes the name of the operations, their parameters, their types, and their return values. System design focuses on defining the services provided by each subsystem that is, enumerating the operations, their parameters, and their high-level behavior. Object design will focus on defining the subsystem interfaces, that is, the type of the parameters and the return value of each operation. The definition of a subsystem in terms of the services it provides helps us focus on its interface as opposed to its implementation. A good subsystem interface should provide as little information

about its implementation. This allows us to minimize the impact of change when we revise the implementation of a subsystem. More generally, we want to minimize the impact of change by minimizing the dependencies among subsystems.

6.1.3. Coupling and coherence

Coupling is the strength of dependencies between two subsystems. If two subsystems are loosely coupled, they are relatively independent, and thus, modifications to one of the subsystems will have little impact on the other. If two subsystems are strongly coupled, modifications to one subsystem is likely to have impact on the other. A desirable property of a subsystem decomposition is that subsystems are as loosely coupled as possible. This minimizes the impact that errors or future changes have on the correct operation of the system.

Consider, for example, a compiler in which a parse tree is produced by the syntax analysis subsystem and handed over to the semantic analysis subsystem. Both subsystems access and modify the parse tree. An efficient way for sharing large amounts of data is to allow both subsystems to access the nodes of the tree via attributes. This introduces, however, a tight coupling: Both subsystems need to know the exact structure of the parse tree and its invariants.

Figures 5.4 and 5.5 show the effect of changing the parse tree data structure for two cases: The left columns show the class interfaces when attributes are used for sharing data; the right columns show the class interfaces when operations are used. Because both the syntax analyzer and the semantic analyzer depend on these classes, both subsystems would need to be modified and retested in the case depicted by the left column. In general, sharing data via attributes increases coupling and should be avoided.

Coherence is the strength of dependencies within a subsystem. If a subsystem contains many objects that are related to each other and perform similar tasks, its coherence is high. If a subsystem contains a number of unrelated objects, its coherence is low. A desirable property of a subsystem decomposition is that it leads to subsystems with high coherence.

For example, consider a decision tracking system for recording design problems, discussions, alternative evaluations, decisions, and their implementation in terms of tasks (see Figure 6.4).

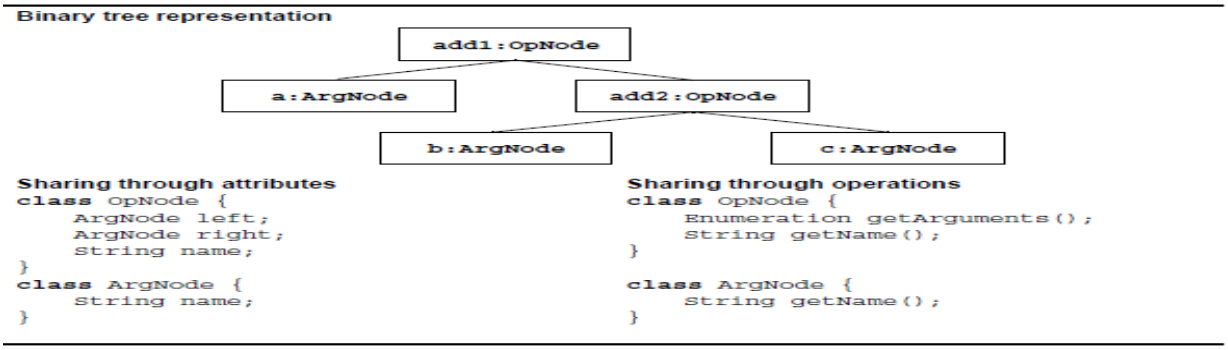


Figure 6.4 Example of coupling reduction (UML object diagram and Java declarations).

6.1.4. Software architecture

As the complexity of systems increases, the specification of the system decomposition is critical. It is difficult to modify or correct a weak decomposition once development has started, as most subsystem interfaces have to change. In recognition of the importance of this problem, the concept of **software architecture** has emerged. A software architecture includes the system decomposition, the global control flow, error-handling policies and inter subsystem communication protocols.

In this section, we describe a few sample architectures that can be used for different systems. This is by no means a systematic or thorough exposition of the subject. Rather, we aim to provide a few representative examples and refer the reader to the literature for more details.

Repository architecture

In the repository architecture (see Figure 6.5), subsystems access and modify data from a single data structure called the central **repository**. Subsystems are relatively independent and interact only through the central data structure. Control flow can be dictated either by the central repository (e.g., triggers on the data invoke peripheral systems) or by the subsystems (e.g., independent flow of control and synchronization through locks in the repository).

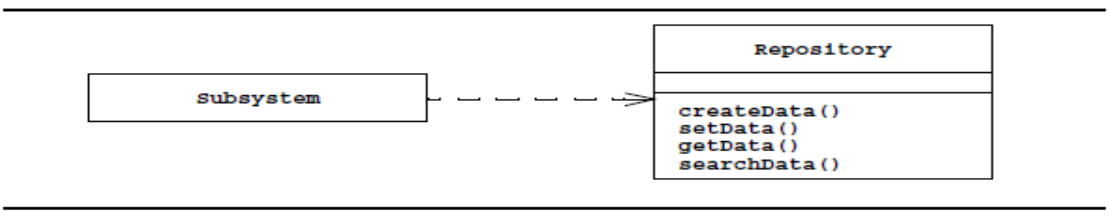
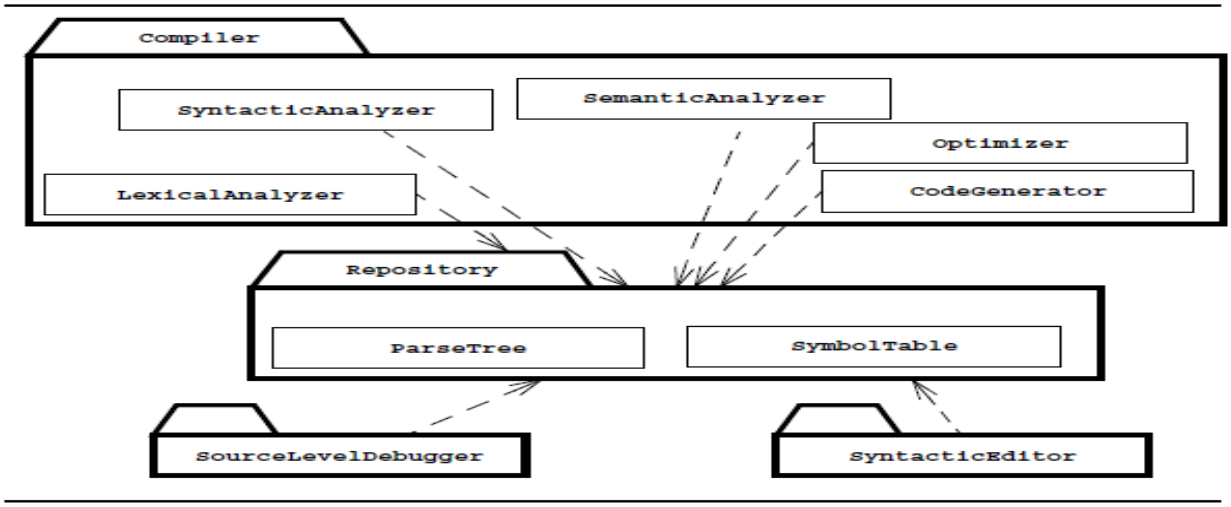


Figure 6.5 Repository architecture (UML class diagram).

Every subsystem depends only on a central data structure called the repository. The repository in turn, has no knowledge of the other subsystems.

The repository architecture is typical for database management systems, such as a payroll system or a bank system. The central location of the data makes it easier to deal with concurrency and integrity issues between subsystems. Modern compilers and software development environments also follow a repository architecture (see Figure 5.9). The different subsystems of a compiler access and update a central parse tree and a symbol table. Debuggers and syntax editors access the symbol table as well.

The repository subsystem can also be used for implementing the global control flow. In the compiler example of Figure 6.6, each individual tool (e.g., the compiler, the debugger, and the editor) is invoked by the user. The repository only ensures that concurrent accesses are serialized. Conversely, the repository can be used to invoke the subsystems based on the state of the central data structure. These systems are called blackboard systems. The HEARSAY II Speech understanding system, one of the first blackboard systems, selected tools to invoke based on the



current state of the blackboard.

Figure 6.6 An instance of the repository architecture (UML Class diagram).

A modern compiler incrementally generates a parse tree and a symbol table that can be later used by debuggers and syntax editors.

Repository architectures are well suited for applications with constantly changing complex data processing tasks. Once a central repository is well defined, we can easily add new services in the

form of additional subsystems. The main disadvantage of repository systems is that the central repository can quickly become a bottleneck, both from a performance aspect and a modifiability aspect. The coupling between each subsystem and the repository is high, thus making it difficult to change the repository without having an impact on all subsystems.

Model/View/Controller

In the Model/View/Controller (MVC) architecture, subsystems are classified into three different types: **model subsystems** are responsible for maintaining domain knowledge, **view subsystems** are responsible for displaying it to the user, and **controller subsystems** are responsible for managing the sequence of interactions with the user. The model subsystems are developed such that they do not depend on any view or controller subsystem. Changes in their state are propagated to the view subsystem via a subscribe/notify protocol. The MVC architecture is a special case of repository architecture where Model implements the central data structure and control objects dictate the control flow.

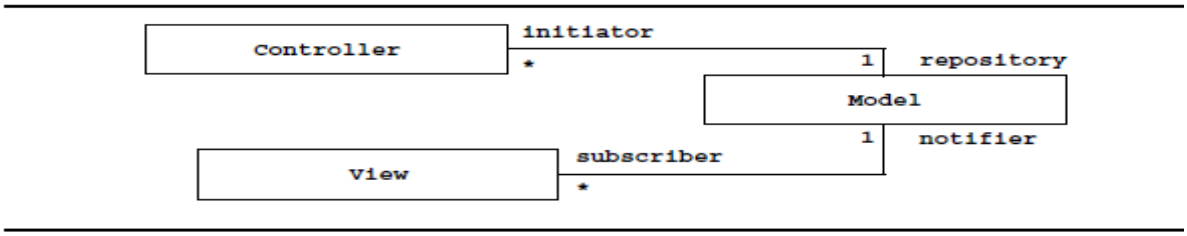


Figure 6.7 Model/View/Controller architecture (UML class diagram).

The Controller gathers input from the user and sends messages to the Model. The Model maintains the central data structure. The View(s) display the Model and is notified (via a subscribe/notify protocol) whenever the Model is changed.

Figure 6.8 shows the sequence of events:

1. The InfoView and the FolderView both subscribe for changes to the File models they display (when they are created).
2. The user types the new name of the file.
3. The Controller, the object responsible for interacting with the user during file name changes, sends a request to the Model.
4. The Model changes the file name and notifies all subscribers of the change.
5. Both InfoView and FolderView are updated, so the user sees a consistent change.

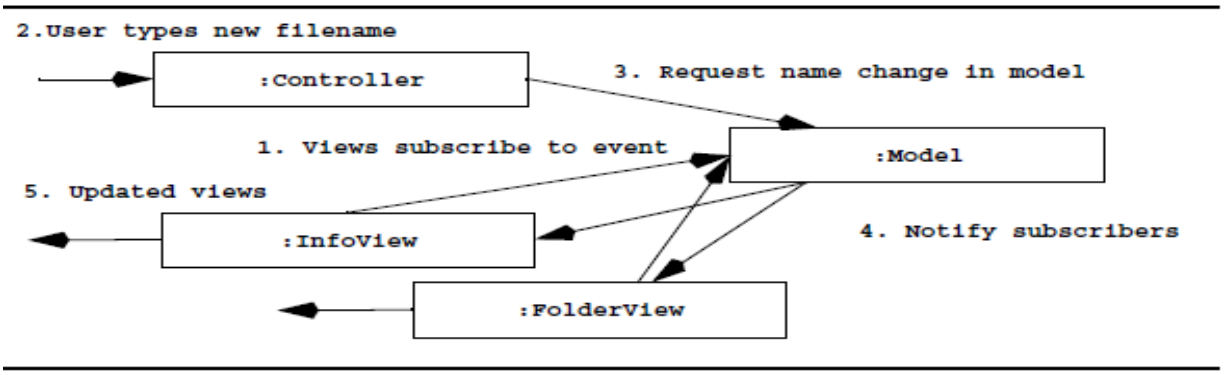


Figure 6.8 Sequence of events in the Model/View/Control architecture (UML collaboration diagram).

MVC architectures are well suited for interactive systems, especially when multiple views of the same model are needed. MVC can be used for maintaining consistency across distributed data; however, it introduces the same performance bottleneck as for other repository architectures.

Client/server architecture

In the client/server architecture (Figure 6.9), a subsystem, the **server**, provides servicesto instances of other subsystems called the **clients**, which are responsible for interacting with theuser. The request for a service is usually done via a remote procedure call mechanism or acommon object broker (e.g., CORBA or Java RMI). Control flow in the clients and the servers isindependent except for synchronization to manage requests or to receive results.

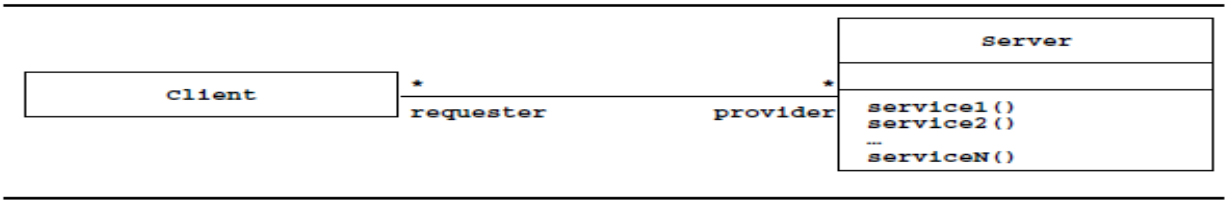


Figure 6.9 Client/server architecture (UML class diagram).

Clients request services from one or more servers. The server has no knowledge of the client. The client/server architecture is a generalization of the repository architecture.

An information system with a central database is an example of a client/server architecture. The clients are responsible for receiving inputs from the user, performing range checks, and initiating database transactions once all necessary data are collected. The server is then responsible for performing the transaction and guaranteeing the integrity of the data. In this case, a client/server

architecture is a special case of the repository architecture, where the central ata structure is managed by a process. Client/server systems, however, are not restricted to a single server. On the World Wide Web, a single client can easily access data from thousands of different servers (Figure 6.10).

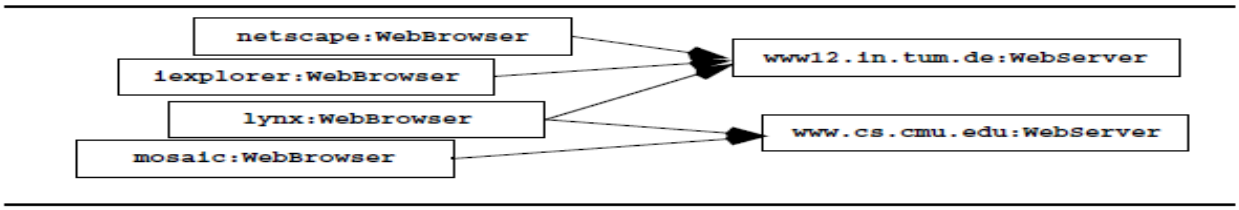


Figure 6.10 The World Wide Web as an instance of the client/server architecture (UML object diagram).

Client/server architectures are well suited for distributed systems that manage large amounts of data.

Peer-to-peer architecture

A peer-to-peer architecture (see Figure 6.11) is a generalization of the client/server architecture in which subsystems can act both as client or as servers, in the sense that each subsystem can request and provide services. The control flow within each subsystem is independent from the others except for synchronizations on requests.

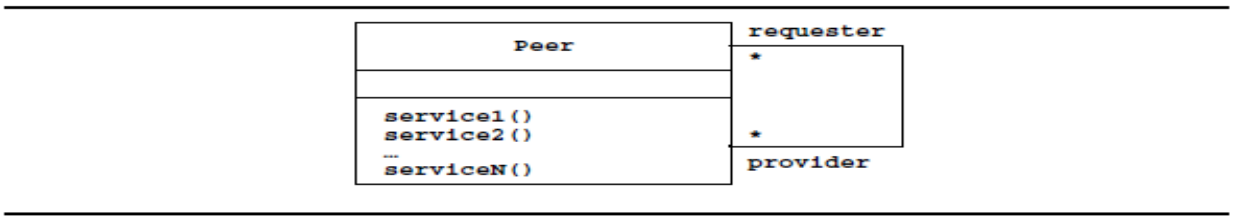


Figure 6.11 Peer-to-peer architecture (UML class diagram). Peers can request services from and provide services to other peers.

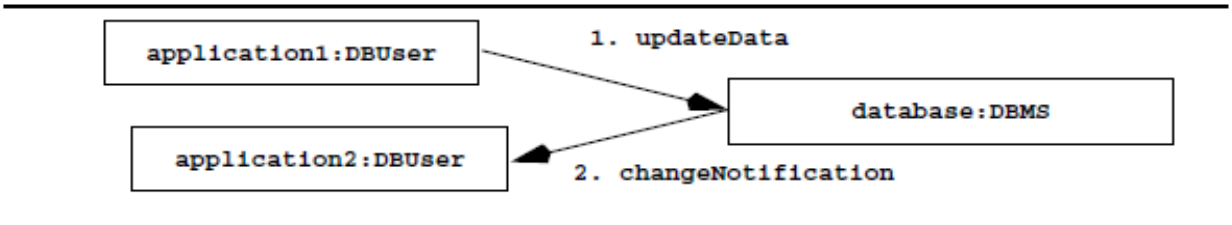


Figure 6.12 An example of peer-to-peer architecture (UML collaboration diagram). The database server can both process requests from and send notifications to applications.

An example of a peer-to-peer architecture is a database which, on the one hand, accepts requests from the application and, on the other hand, sends notifications to the application whenever certain data are changed. Peer-to-peer systems are more difficult to design than client/ server systems are. They introduce the possibility of deadlocks and complicate the control flow.

Pipe and filter architecture

In the pipe and filter architecture (see Figure 6.13), subsystems process data received from a set of inputs and send results to other subsystems via a set of outputs. The subsystems are called **filters**, and the associations between the subsystems are called **pipes**. Each filter only knows the content and the format of the data received on the input pipes, not the filters that produced them. Each filter is executed concurrently and synchronization is done via the pipes. The pipe and filter architecture is modifiable: Filters can be substituted for others or reconfigured to achieve a different purpose.

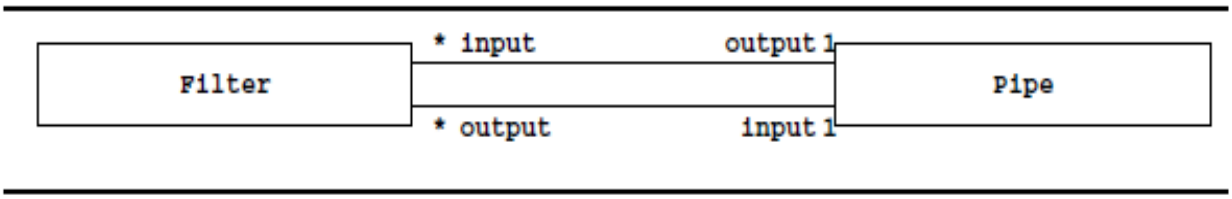


Figure 6.13 Pipe and filter architecture (UML class diagram). A Filter can have many inputs and outputs. A Pipe connects one of the outputs of a Filter to one of the inputs of another Filter.

The best known example of pipe and filter architecture is the Unix shell. Most filters are written such that they read their input and write their results on standard pipes. This enables a Unix user to combine them in many different ways. Figure 5.18 shows an example made out of four filters. The output of ps (process status) is fed into grep (search for a pattern) to get rid of all the processes that are not owned by a specific user. The output of grep (i.e., the processes owned by the user) is then sorted by sort and then sent to more, which is a filter that displays its input to a terminal, one screen at a time.

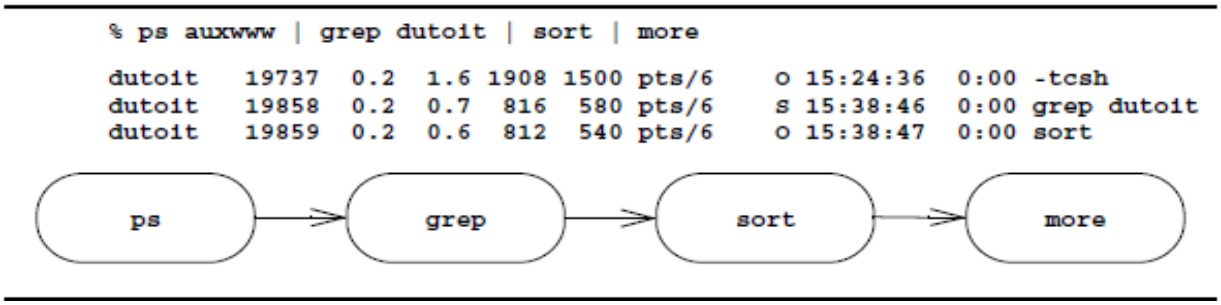


Figure 6.14 An instance of the pipe and filter architecture (Unix command and UML activity diagram).

Pipe and filter architectures are suited for systems that apply transformations to streams of data without intervention by users. They are not suited for systems that require more-complex interactions between components, such as an information management system or an interactive system.

6.2. SYSTEM DESIGN ACTIVITIES: FROM OBJECTS TO SUBSYSTEMS

6.2.1. Identifying design goals

The definition of design goals is the first step of system design. It identifies the qualities that our system should focus on. Many design goals can be inferred from the nonfunctional requirements or from the application domain. Others will have to be elicited from the client. It is, however, necessary to state them explicitly such that every important design decision can be made consistently following the same set of criteria.

For example, in the light of the nonfunctional requirements for MyTrip, we identify *reliability* and *fault tolerance to connectivity loss* as design goals. We then identify *security* as a design goal, as numerous drivers will have access to the same trip planning server. We add *modifiability* as a design goal, as we want to provide the ability for drivers to select a trip planning service of their choice. The following box summarizes the design goals we identified.

In general, we can select design goals from a long list of highly desirable qualities. Tables 5-1–5-5 list a number of possible design criteria. These criteria are organized into five groups: *performance*, *dependability*, *cost*, *maintenance*, and *end user criteria.* Performance, dependability, and end user criteria are usually specified in the requirements or inferred from the application domain. Cost and maintenance criteria are dictated by the customer and the supplier.

Performance criteria (Table 5-1) include the speed and space requirements imposed on the system. Should the system be responsive or should it accomplish a maximum number of tasks? Is memory space available for speed optimizations or should memory be used sparingly?

Table 5-1 *Performance criteria*

Design criterion	Definition
Response time	How soon is a user request acknowledged after the request has been issued?
Throughput	How many tasks can the system accomplish in a fixed period of time?
Memory	How much space is required for the system to run?

Table 5-2 *Dependability criteria*

Design criterion	Definition
Robustness	Ability to survive invalid user input
Reliability	Difference between specified and observed behavior.
Availability	Percentage of time system can be used to accomplish normal tasks.
Fault tolerance	Ability to operate under erroneous conditions.
Security	Ability to withstand malicious attacks
Safety	Ability to not endanger human lives, even in the presence of errors and failures.

Dependability criteria (Table 5-2) determine how much effort should be expended in minimizing system crashes and their consequences. How often can the system crash? How available to the user should the system be? Are there safety issues associated with system crashes? Are there security risks associated with the system environment?

Cost criteria (Table 5-3) include the cost to develop the system, to deploy it, and to administer it. Note that cost criteria not only include design considerations but managerial ones as well. When the system is replacing an older one, the cost of ensuring backward compatibility or transitioning to the new system has to be taken into account. There are also trade-offs between different types of costs such as development cost, end user training cost, transition costs and maintenance costs. Maintaining backward compatibility with a previous system can add to the development cost while reducing the transition cost.

Table 5-3 *Cost criteria*

Design criterion	Definition
Development cost	Cost of developing the initial system
Deployment cost	Cost of installing install the system and training the users.
Upgrade cost	Cost of translating data from the previous system. This criteria results in backward compatibility requirements.
Maintenance cost	Cost required for bug fixes and enhancements to the system
Administration cost	Money required to administer the system.

Maintenance criteria (Table 5-4) determine how difficult it is to change the system after deployment. How easily can new functionality be added? How easily can existing functions be revised? Can the system be adapted to a different application domain? How much effort will be required to port the system to a different platform? These criteria are harder to optimize and plan for, as it is seldom clear how long the system will be operational and how successful the project will be.

Table 5-4 *Maintenance criteria*

Design criterion	Definition
Extensibility	How easy is it to add the functionality or new classes of the system?
Modifiability	How easy is it to change the functionality of the system?
Adaptability	How easy is it to port the system to different application domains?
Portability	How easy is it to port the system to different platforms?
Readability	How easy is it to understand the system from reading the code?
Traceability of requirements	How easy is it to map the code to specific requirements?

End user criteria (Table 5-5) include qualities that are desirable from a users' point of view that have not yet been covered under the performance and dependability criteria. These include usability (How difficult is the software to use and to learn?) and utility (How well does the system support the user's work?). Often these criteria do not receive much attention, especially when the client contracting the system is distinct from the users of the system.

Table 5-5 *End user criteria*

Design criterion	Definition
Utility	How well does the system support the work of the user?
Usability	How easy is it for the user to use the system?

When defining design goals, only a small subset of these criteria can be simultaneously taken into account. It is, for example, unrealistic to develop software that is safe, secure, and cheap. Typically, developers need to prioritize design goals and trade them off against each other as well as against managerial goals as the project runs behind schedule or over budget. Table 5-6 lists several possible trade-offs.

Table 5-6 *Examples of design goal trade-offs*

Trade-off	Rationale
Space vs. speed	If the software does not meet response time or throughput requirements, more memory can be expended to speed up the software (e.g., caching, more redundancy, etc.). If the software does not meet memory space constraints, data can be compressed at the cost of speed.
Delivery time vs. functionality	If the development runs behind schedule, a project manager can deliver less functionality than specified and deliver on time, or deliver the full functionality at a later time. Contract software usually puts more emphasis on functionality, whereas off-the-shelf software projects puts more emphasis on delivery date.
Delivery time vs. quality	If the testing runs behind schedule, a project manager can deliver the software on time with known bugs (and possibly provide a later patch to fix any serious bugs) or to deliver the software later with more bugs fixed.
Delivery time vs. staffing	If development runs behind schedule, a project manager can add resources to the project in order to increase productivity. In most cases, this option is only available early in the project: Adding resources usually decreases productivity while new personnel is being trained or brought up to date. Note that adding resources will also raise the cost of development.

6.3. DOCUMENTING SYSTEM DESIGN

System design is documented in the System Design Document (SDD). It describes design goals set by the project, the subsystem decomposition (with UML class diagrams), the hardware/software mapping (with UML deployment diagrams), the data management, the access control, control flow mechanisms, and boundary conditions. The SDD is used to define interfaces between teams of developers and as reference when architecture-level decisions need to be revisited. The audience for the SDD includes the project management, the system architects (i.e., the developers who participate in the system design), and the developers who design and implement each subsystem. The following is an example template for a SDD:

System Design Document

1. Introduction
   - 1.1 Purpose of the system
   - 1.2 Design goals
   - 1.3 Definitions, acronyms, and abbreviations
   - 1.4 References
   - 1.5 Overview
2. Current software architecture
3. Proposed software architecture
   - 3.1 Overview
   - 3.2 Subsystem decomposition
   - 3.3 Hardware/software mapping
   - 3.4 Persistent data management
   - 3.5 Access control and security
   - 3.6 Global software control
   - 3.7 Boundary conditions
4. Subsystem services

Glossary

6.4.AN OVERVIEW OF OBJECT DESIGN

Conceptually, we think of system development as filling a gap between the problem and the machine. The activities of system development incrementally close this gap by identifying and defining objects that realize part of the system (Figure 6.15).

Analysis reduces the gap between the problem and the machine by identifying objects representing user-visible concepts. During analysis, we describe the system in terms of external behavior, such as its functionality (use case model), the application domain concepts it manipulates (object model), its behavior in terms of interactions (dynamic model), and its nonfunctional requirements.

System design reduces the gap between the problem and the machine by defining a hardware/software platform that provides a higher level of abstraction than the computer hardware. This is done by selecting off-the-shelf components for realizing standard services, such as middleware, user interface toolkits, application frameworks, and class libraries.

During object design, we refine the analysis and system design models, identify new objects, and close the gap between the application objects and off-the-shelf components. This includes the identification of custom objects, the adjustment of off-the-shelf components, and the precise specification of each subsystem interface and class. As a result, the object design model can be partitioned into sets of classes such that they can be implemented by individual developers.

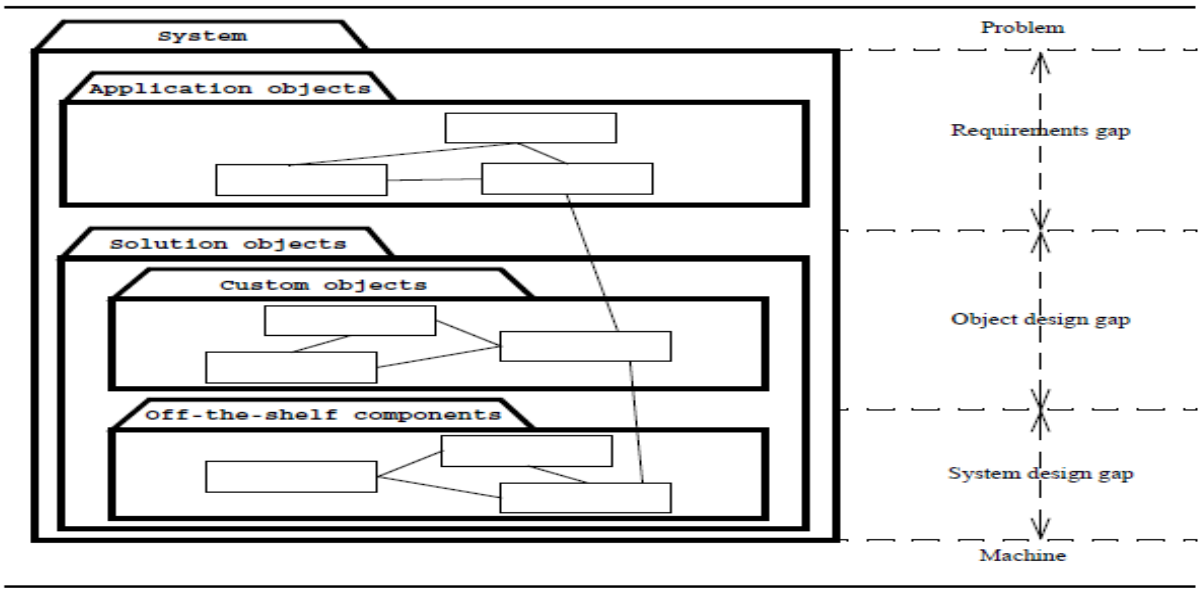


Figure 6.16 Object design closes the gap between application objects identified during requirements and off-the-shelf components selected during system design (stylized UML class diagram).

Object design includes four groups of activities (see Figure 6.16):

- *Service specification.* During object design, we specify the subsystem services (identified during system design) in terms of class interfaces, including operations, arguments, type signatures, and exceptions. During this activity, we also find missing operations and objects needed to transfer data among subsystems. The result of service specification is a complete interface specification for each subsystem. The subsystem service specification is often called subsystem **API** (Application Programmer Interface).
- *Component selection.* During object design, we use and adapt the off-the-shelf components identified during system design to realize each subsystem. We select class libraries and additional components for basic data structures and services. Often, we need to adjust the components we selected before we can use them, by wrapping custom objects around them or by refining them using inheritance. During these activities, we face the same buy versus build trade-off that we faced during system design.
- *Restructuring.* Restructuring activities manipulate the system model to increase code reuse or meet other design goals. Each restructuring activity can be seen as a graph

transformation on subsets of a particular model. Typical activities include transforming n-ary associations into binary associations, implementing binary associations as references, merging two similar classes from two different subsystems into a single class, collapsing classes with no significant behavior into attributes, splitting complex classes into simpler ones, rearranging classes and operations to increase the inheritance and packaging. During restructuring, we address design goals such as maintainability, readability, and understandability of the system model.

- *Optimization.* Optimization activities address performance requirements of the system model. This includes changing algorithms to respond to speed or memory requirements, reducing multiplicities in associations to speed up queries, adding redundant associations for efficiency, rearranging execution orders, adding derived attributes to improve the access time to objects and opening up the architecture, that is, adding access to lower layers because of performance requirements.

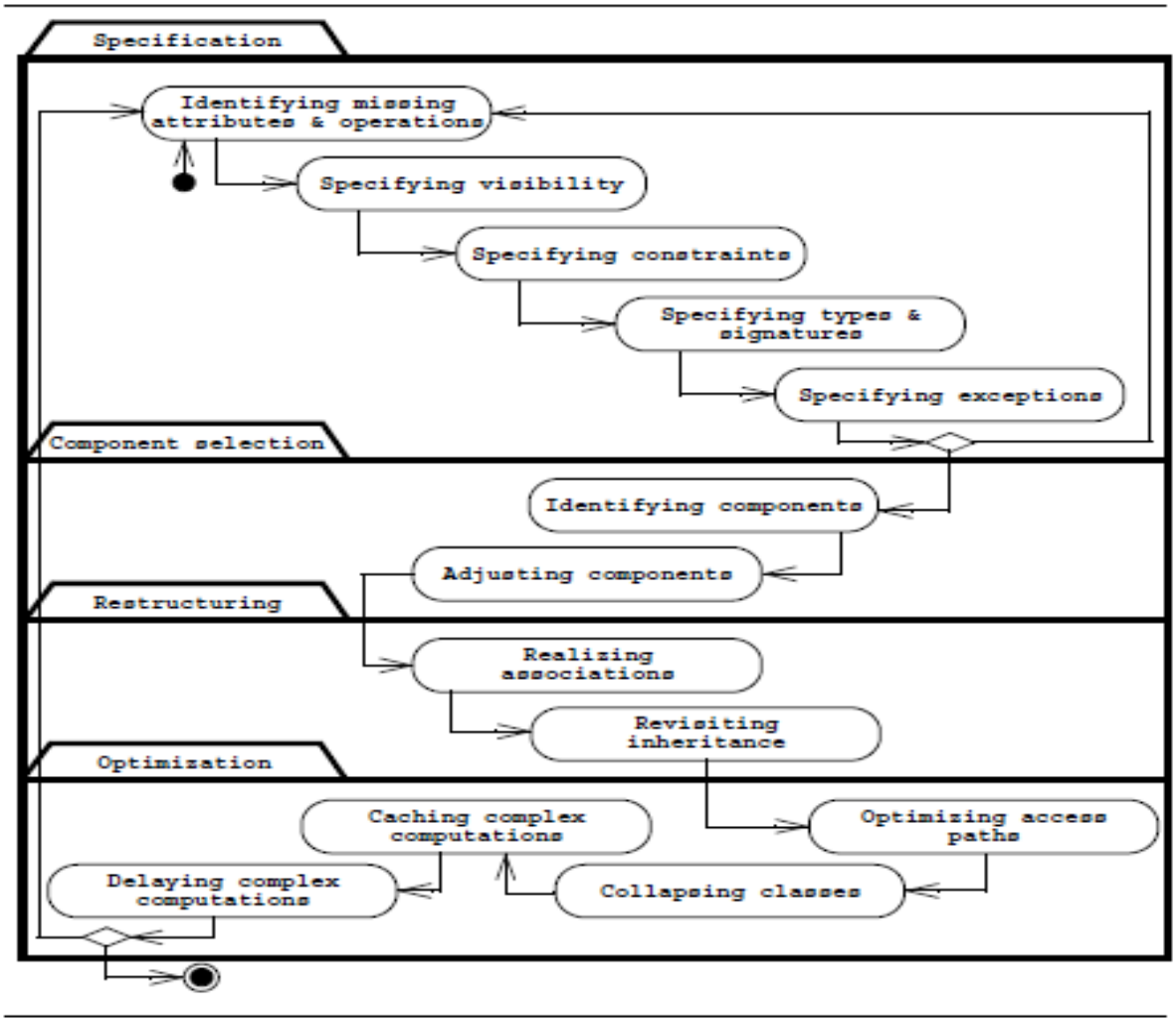


Figure 6.17 Activities of object design (UML activity diagram).

6.5. OBJECT DESIGN CONCEPTS

In this section, we present the principal object design concepts:

6.5.1. Application objects versus solution objects revisited

Application objects, also called domain objects, represent concepts of the domain that the system manipulates. **Solution objects** represent support components that do not have a counterpart in the application domain, such as persistent data stores, user interface objects, or middleware.

During analysis, we identify application objects, their relationships, and attributes and operations. During system design, we identify subsystems and most important solution objects. During object design, we refine and detail both sets of objects and identify any remaining solution objects needed to complete the system.

6.5.2. Types, signatures, and visibility revisited

During analysis, we identified attributes and operations without specifying their types or their parameters. During object design, we refine the analysis and system design models by adding type and visibility information. The **type** of an attribute specifies the range of values the attribute can take and the operations that can be applied to the attribute. For example, consider the attribute numElements of a Hashtable class. numElementsrepresent the current number of entries in a given Hashtable. Its type is int, denoting that it is an integer number. The type of the numElements attribute also denotes the operations that can be applied to this attribute: We can add or subtract other integers to numElements.

Operation parameters and return values are typed in the same way as attributes are. The type constrains the range of values the parameter or the return value can take. Given an operation, the tuple made out of the types of its parameters and the type of the return value is called the **signature** of the operation. For example, the put() operation of Hashtable takes two parameters of type Object and does not have a return value. The type signature for put() is then (Object, Object):voi. dSimilarly, the get() operation of Hashtable takes one Object parameter and returns an Object. The type signature of get() is then (Object):Object.

The **visibility** of an attribute or an operation specifies whether it can be used by other classes or not. UML defines three levels of visibility:

- **Private**. A private attribute can be accessed only by the class in which it is defined. Similarly, a private operation can be invoked only by the class in which it is defined. Private attributes and operations cannot be accessed by subclasses or other classes.
- **Protected**. A protected attribute or operation can be accessed by the class in which it is defined and on any descendant of the class.
- **Public**. A public attribute or operation can be accessed by any class.

6.5.3. Contracts: Invariants, preconditions, and post-conditions

Contracts are constraints on a class that enable caller and callee to share the same assumptions about the class. A contract specifies constraints that the caller must meet before using the class as well as constraints that are ensured by the callee when used.

Contracts include three types of constraints:

- An **invariant** is a predicate that is always true for all instances of a class. Invariants are constraints associated with classes or interfaces. Invariants are used to specify consistency constraints among class attributes.
- A **precondition** is a predicate that must be true before an operation is invoked. Preconditions are associated with a specific operation. Preconditions are used to specify constraints that a caller must meet before calling an operation.
- A **post-condition** is a predicate that must be true after an operation is invoked. Post-conditions are associated with a specific operation. Post-conditions are used to specify constraints that the object must ensure after the invocation of the operation.

6.5.4. UML's Object Constraint Language

In UML, constraints are expressed using OCL. OCL is a language that allows constraints to be formally specified on single model elements (e.g., attributes, operations, classes) or groups of model elements (e.g., associations and participating classes). A constraint is expressed as an OCL expression returning the value True or False. OCL is not a procedural language and thus cannot be used to denote control flow. In this chapter, we focus exclusively on the aspects of OCL related to invariants, preconditions, and post-conditions.

A constraint can be depicted as a note attached to the constrained UML element by a dependency relationship. A constraint can be expressed in natural language or in a formal language such as OCL.

Summary

- The following are the products of system analysis: A set of nonfunctional requirements; A use case model; An object model; A sequence diagram
- The analysis model describes the system completely from the actors point of view and serves as the basis of communication between the client and the developers. The following are the products of the system design; List of design goals; Software architecture
- The design goals are derived from the nonfunctional requirements. The issues has to be addressed when decomposing a system; Hardware/software mapping; Data Management; Control flow; Access control; Boundary conditions;
- The following are lists of system design concepts: Subsystems and classes: - To reduce the complexity of the solution domain, we decompose a system into simple parts, called

subsystems. Decomposition is a process to break down a system in to components. The direct result of decomposition is component. Service and subsystem interface: - a service is a set of related operations that share a common purpose. The set of operations of a subsystem are available to other subsystems form the **subsystem interface.** Coupling and coherence: - Coupling is the strength of dependencies between two subsystems. Coherence is the strength of dependencies within a subsystem. A desirable property of subsystem decomposition is that subsystems should be as loosely as possible with high coherence.

- Software architecture: - Repository architecture: - here subsystems access and modify data from a single data structure called the central repository.Model|View|Control(MVC): - subsystems are classified into three different types:Model subsystems: - are responsible for maintaining domain knowledge.
- View subsystems: - are responsible for displaying to the user: Control subsystems: - are responsible for managing the sequence of interactions with the user.Pipe and filter architecture: -subsystems process data received from a set of inputs and send results to other subsystems via a set of outputs.

System design activities: Identifying design goals: - criteria that are used to select the design goals; Performance criteria; Dependability criteria; Cost criteria; Maintenance criteria; End-User criteria

These are the section that are included in System Design Document(SDD): Introduction; Current software architecture; Proposed software architecture; Subsystem interface; During object design, we refine the analysis and system design models, identify new objects, and close the gap between the application objects and off-the-shelf components: Object design includes four groups of activities; Service specification; Component selection; Restructuring activities manipulate the system model to increase code reuse or meet other design goals.Optimization activities address performance requirements of the system model.Following are the list of Object design concepts: Application objects versus solution objects revisited: -

Application objects, also called domain objects, represent concepts of the domain that the system manipulates.

Solution objects represent support components that do not have a counterpart in the application domain.

Types, signatures, and visibility: -**Type**of an attribute specifies the range of values the attribute can take and the operations that can be applied to the attribute.**Signature: -**the tuple made out of the types of its parameters and the type of the return value of a given operation.**Visibility** of an attribute or an operation specifies whether it can be used by other classes or not.

Three levels of visibility:**Private: -** can be accessed only by the class in which it is defined.**Protected: -**can be accessed by the class in which it is defined and on any descendant of the class.**Public: -**can be accessed by any class.**Contracts: -** are constraints on a class that enable caller and callee to share the same a assumptions about the class.

Contracts include three types of constraints:An **invariant** is a predicate that is always true for all instances of a class.A **precondition** is a predicate that must be true before an operation is invoked.A **post condition** is a predicate that must be true after an operation is invoked.**OCL** is a language that allows constraints to be formally specified on single model elements or groups of model elements.

Review Question & Problems

1. Which phase's system development life cycle handles the systems hardware configuration?
   a. Planning
   b. Analysis
   c. Design
   d. Implementation
2. What are the results of design?
3. _____ is the process of breaking down a system in to parts.
4. The form in which a system makes its operation available to other system is called _____
   a. Service
   b. System interface
   c. Application programming interface
   d. a and b
5. Why is the desirable property of a subsystem decomposition is that subsystems are as loosely coupled as possible?
6. List the different types of software architecture and explain briefly
7. _____ is a tuple made out of the types of its parameters and they type of the return value for a give operation.
   a. Type
   b. Signature
   c. Visibility
   d. None

8. Match the following

A.	**B.**
A. attributes or operation that can be accessed by any class	1. Invariants
B. a predicate that is always true for all instances of a class	2. Protected
C. Object design activity that address performance requirements of the system model	3. Restructuring
D. How easy is it to add the functionality or new classes of the system	4. Optimization
	5. Extensibility
	6. Usability

References

- *Brahmin, Ali(1999), Object oriented System development , McGraw Hill, USA*
- *OOSE Practical software development using UML and Java*
- *OOSE 8th edition Stephen R. Schach*

Chapter 7: Software Quality Assurance

OBJECTIVES

After the completion of this chapter students will be able to:

- ⇨ Describe quality assurance issues
- ⇨ Describe how to perform non-execution-based testing (inspections) of artifacts
- ⇨ Describe the principles of execution-based testing
- ⇨ Explain what needs to be tested.

INTRODUCTION

Reliability is a measure of success with which the observed behavior of a system conforms to the specification of its behavior. **Software reliability** is the probability that a software system will not cause the failure of the system for a specified time under specified conditions. **Failure** is any deviation of the observed behavior from the specified behavior. An **error** means the system is in a state such that further processing by the system will lead to a failure, which causes the system to deviate from its intended behavior. A **fault**, also called defect or bug, is the mechanical or algorithmic cause of an error. The goal of testing is to maximize the number of discovered faults, which then allows developers to correct them and increase the reliability of the system.

Categorize the following according to whether each describes a failure, a defect or an error:

(a) A software engineer, working in a hurry, unintentionally deletes animportant line of source code.

(b) On 1 January 2040 the system reports the date as 1 January 1940.

(c) No design documentation or source code comments are provided for a complex algorithm.

(d) A fixed size array of length 10 is used to maintain the list of courses taken by a student during one semester. The requirements are silent about the maximum number of courses a student may take at any one time.

7.1 OVERVIEW OF SOFTWARE QUALITY ASSURANCE

We define testing as the systematic attempt to *find errors in a planned way* in the implemented software. Contrast this definition with another commonly used definition that says that "testing is the process of demonstrating that *errors are not presen*t." The distinction between these two definitions is important. Our definition does not mean we simply demonstrate that the program does what it is intended to do. The explicit goal of testing is to demonstrate the presence of faults.

Our definition implies that you, the developer, are willing to dismantle things. Most activities of the development process are constructive: During analysis, design, and implementation, objects and relationships are identified, refined, and mapped onto a computer environment.

Testing requires a different thinking, in that developers try to detect faults in the system, that is, differences between the reality and the requirements. Many developers find it difficult to do. One reason is the way we use the word "success" during testing. Many project managers call a test case "successful" if it does not find an error; that is, they use the second definition of testing during development. However, because "successful" denotes an achievement, and "unsuccessful" means something undesirable, these words should not be used in this fashion during testing.

7.1.1. Quality control techniques

There are many techniques for increasing the reliability of a software system. Figure 9-1 focuses on three classes of techniques for avoiding faults, detecting faults, and tolerating faults. *Fault avoidance techniques* try to detect errors statically, that is, without relying on the execution of any of the system models, in particular the code model. *Fault detection techniques*, such as debugging and testing, are uncontrolled and controlled experiments, respectively, used during the development process to identify errors and find the underlying faults before releasing the system. *Fault tolerance techniques* assume that a system can be released with errors and those system failures can be dealt with by recovering from them at run time.

7.1.2. Fault avoidance techniques

Fault avoidance tries to prevent the occurrence of errors and failures by finding faults in the system before it is released. Fault avoidance techniques include:

- Development methodologies
- Configuration management
- Verification techniques
- Reviews

Development methodologies avoid faults by providing techniques that minimize fault introduction in the system models and code. Such techniques include the unambiguous representation of requirements, the use of data abstraction and data encapsulation, minimizing of the coupling between subsystems and maximizing of subsystem coherence, the early definition of subsystem interfaces, and the capture of rationale information for maintenance activities. The

assumptions of most of these techniques is that the system contains many less faults if complexity is dealt with model-based approaches.

Configuration management avoids faults caused by undisciplined change in the system models. For example, it is a common mistake to a change a subsystem interface without notifying all developers of calling components. Configuration management can make sure that, if analysis models and code are becoming inconsistent with one another, analysts and implementor are notified. The assumption behind configuration management is that the system contains many less faults if change is controlled.

Verification attempts to find faults before any execution of the system. Verification is possible in specific cases, such as the verification of a small operating system kernel. Verification, however, has limits. It is not in a mature enough state that it can be applied to assure the quality of large complex systems. Moreover, it assumes that the requirements are correct, which is seldom the case: Assume we need to verify an operation and we know the pre- and postconditions for the operation. If we can verify that the operation provides a transformation such that, before the execution of the operation the precondition is true, and after the execution the postcondition holds, then the operation is correctly implemented; that is, it is verified. However, sometimes we forget important parts of the pre- or postcondition. For example, the postconditions might not state that the train should stay on the ground while moving from track 1 to track 2. Finally, verification addresses only algorithmic faults: It cannot deal with mechanical faults in the virtual machine. For verification, we must assume that the virtual machine on which the system is executing is correct and does not change during the proof.

A **review** is the manual inspection of parts or all aspects of the system without actually executing the system. There are two types of reviews: walkthrough and inspection. In a code **walkthrough**, the developer informally presents the API, the code, and associated documentation of the component to the review team. The review team makes comments on the mapping of the analysis and object design to the code using use cases and scenarios from the analysis phase. An **inspection** is similar to a walkthrough, but the presentation of the unit is formal. In fact, in a code inspection, the developer is not allowed to present the artifacts (models, code, and documentation). This is done by the review team, which is responsible for checking the interface and code of the component against the requirements. It also checks the algorithms for efficiency with respect to the nonfunctional requirements. Finally, it checks comments about the code and compares them

with the code itself to find inaccurate and incomplete comments. The developer is only present in case the review needs clarifications about the definition and use of data structures or algorithms.

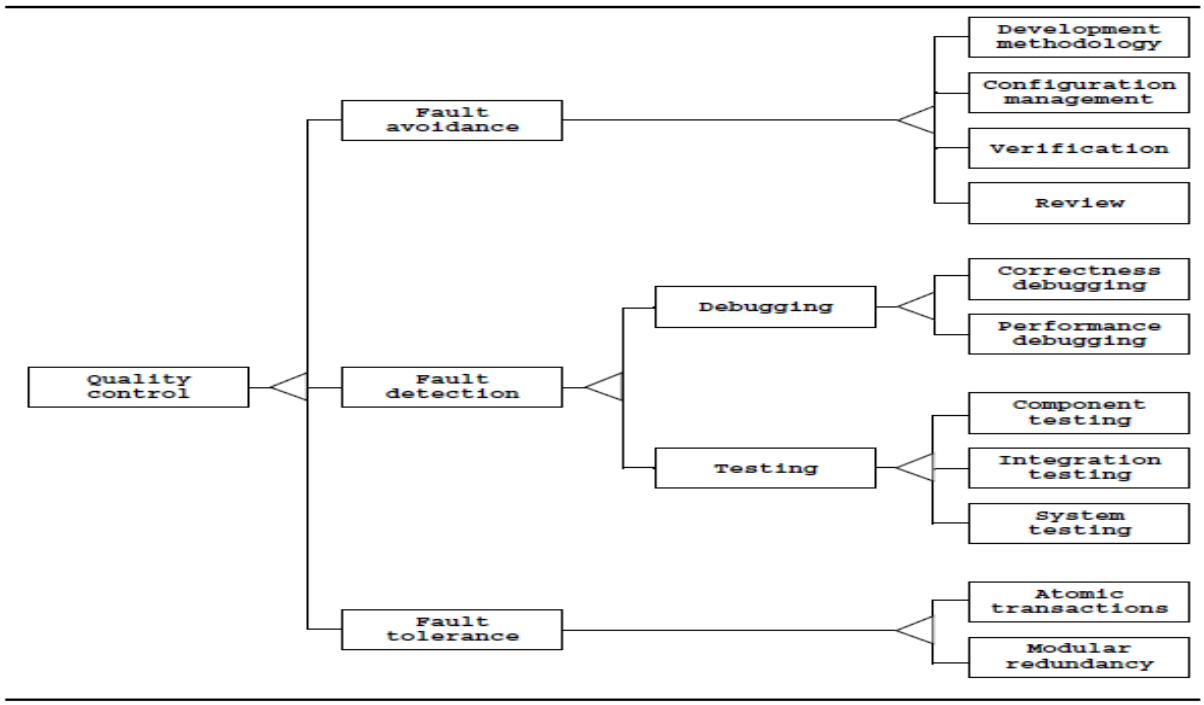


Figure 6-1 Taxonomy of quality control activities (UML class diagram).

7.1.3. Fault detection techniques

Fault detection techniques assist in finding faults in systems but do not try to recover from the failures caused by them. In general, fault detection techniques are applied during development, but in some cases they are also used after the release of the system. The black boxes in airplanes logging the last few minutes of a flight before the crash is an example of a fault detection technique. We distinguish two types of fault detection techniques, debugging and testing.

Debugging assumes that faults can be found by starting from an unplanned failure. The developer moves the system through a succession of states, ultimately arriving at and identifying the erroneous state. Once this state has been identified, the algorithmic or mechanical fault causing this state needs to be determined. There are two types of debugging: The goal of **correctness debugging** is to find any deviation between the observed and specified functional requirements. **Performance debugging** addresses the deviation between observed and specified nonfunctional requirements, such as response time.

Testing is a fault detection technique that tries to create failures or errors in a planned way. This allows the developer to detect failures in the system before it is released to the customer. Note that this definition of testing implies that a successful test is a test that identifies errors. We will use this definition throughout the development phases. Another often-used definition of testing is that "it demonstrates that errors are not present." We will use this definition only after the development of the system when we try to demonstrate that the delivered system fulfills the functional and nonfunctional requirements.

If we would use this second definition all the time, we would tend to select test data that have a low probability of causing the program to fail. If, on the other hand, the goal is to demonstrate that a program has errors, we tend to look for test data with a higher probability of finding errors. The characteristics of a good test model is that it contains test cases that identify errors. Every input that can possibly be given to the system should be tested; otherwise, there is a chance that faults are not detected. Unfortunately, such an approach requires extremely large testing times for even small systems.

Figure 6.2 depicts an overview of the testing activities:

- **Unit testing** tries to find faults in participating objects and/or subsystems with respect to the use cases from the use case model
- **Integration testing** is the activity of finding faults when testing the individually tested components together, for example, subsystems described in the subsystem decomposition, while executing the use cases and scenarios from the RAD.
    - The **system structure testing** is the culmination of integration testing involving all components of the system. Integration tests and structure tests exploit knowledge from the SDD using an integration strategy described in the Test Plan (TP).
- **System testing** tests all the components together, seen as a single system to identify errors with respect to the scenarios from the problem statement and the requirements and design goals identified in the analysis and system design, respectively:
    - **Functional testing** tests the requirements from the RAD and, if available, from the user manual.
    - **Performance testing** checks the nonfunctional requirements and additional design goals from the SDD. Note that functional and performance testing are both done by developers.

- **Acceptance testing** and **installation testing** check the requirements against the project agreement and should be done by the client, if necessary with support by the developers.

7.1.4. Fault tolerance techniques

If we cannot prevent errors we must accept the fact that the released system contains faults that can lead to failures. Fault tolerance is the recovery from failure while the system is executing. It allows a system to recover from a failure of a component by passing the erroneous state information back to the calling components, assuming one of them knows what to do in this case. Database systems provide atomic transactions to recover from failure during a sequence of actions that need to be executed together or not at all. Modular redundancy is based on the assumption that system failures are usually based on component failures. Modular redundant systems are built by assigning more than one component to perform the same operation. The five onboard computers in the space shuttle are an example of a modular redundant system. The system can continue even if one component is failing, because the other components are still performing the required functionality. The treatment of fault tolerant techniques is important for highly reliable systems but is beyond the scope of this book.

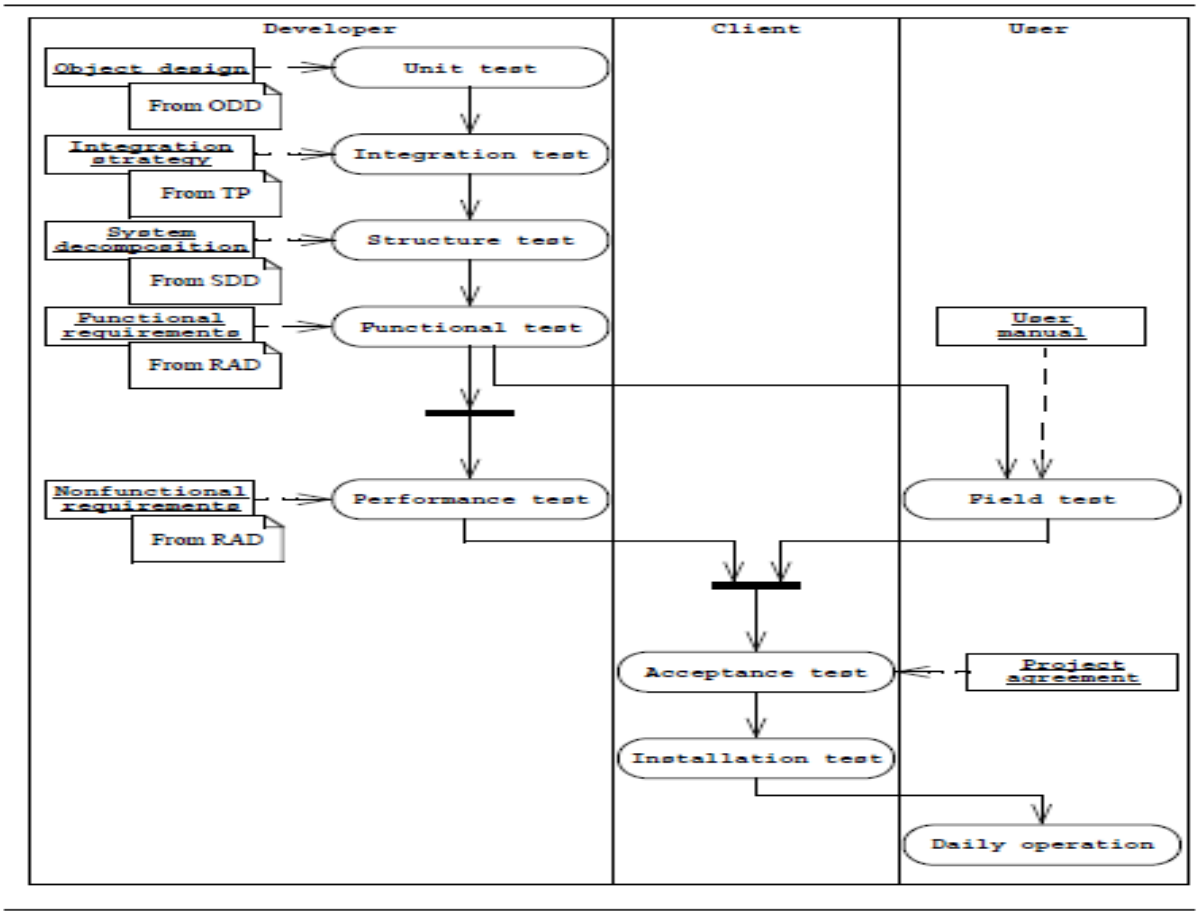


Figure 6.2 Testing activities and their related work products (UML activity diagram). Swimlanes indicate who executes the test.

7.2 TESTING CONCEPTS

In this section, we present the model elements used during testing (Figure 6-3):

- A **component** is a part of the system that can be isolated for testing. A component can be an object, a group of objects, or one or more subsystems.
- A **fault**, also called bug or defect, is a design or coding mistake that may cause abnormal component behavior.
- An **error** is a manifestation of a fault during the execution of the system.
- A **failure** is a deviation between the specification of a component and its behavior. A failure is triggered by one or more errors.
- A **test case** is a set of inputs and expected results that exercises a component with the purpose of causing failures and detecting faults.

- A **test stub** is a partial implementation of components on which the tested component depends. A **test driver** is a partial implementation of a component that depends on the tested component. Test stubs and drivers enable components to be isolated from the rest of the system for testing.
- A **correction** is a change to a component. The purpose of a correction is to repair a fault. Note that a correction can introduce new faults.

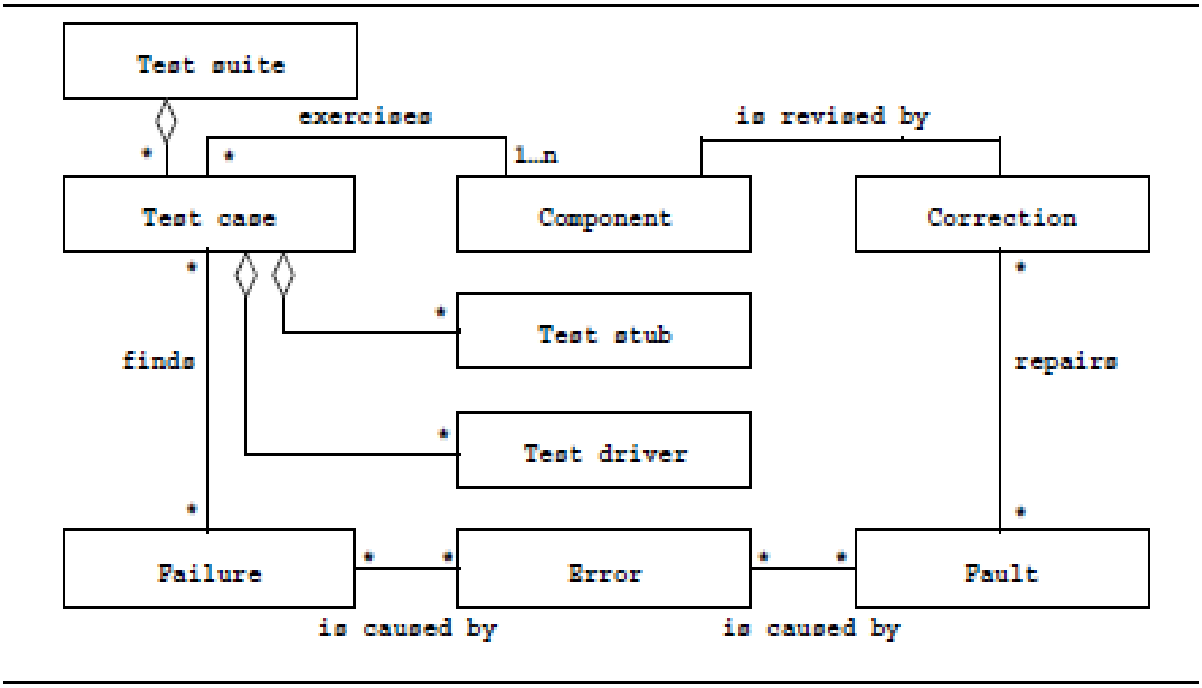


Figure 6-3 Model elements used during testing (UML class diagram).

7.3 TESTING ACTIVITIES

In this section, we describe the activities of testing. These include:

- **Inspecting components**, which finds faults in an individual component through the manual inspection of its source code
- **Unit testing**, which finds faults by isolating an individual component using test stubs and drivers and by exercising the component using a test case
- **Integration testing**, which finds faults by integrating several components together
- **System testing**, which focuses on the complete system, its functional and nonfunctional requirements, and its target environment

7.3.1. Inspecting components

Inspections find faults in a component by reviewing its source code in a formal meeting. Inspections can be conducted before or after the unit test. The first structured inspection process was Michael Fagan's Inspection method [Fagan, 1976]. The inspection is conducted by a team of developers, including the author of the component, a moderator who facilitates the process, and one or more reviewers who find faults in the component. Fagan's Inspection method consists of five steps:

- **Overview**. The author of the component briefly presents the purpose and scope of the component and the goals of the inspection.
- **Preparation**. The reviewers become familiar with the implementation of the component.
- **Inspection meeting**. A reader paraphrases the source code of the component and the inspection team raise issues with the component. A moderator keeps the meeting on track.
- **Rework**. The author revises the component.
- **Follow-up**. The moderator checks the quality of the rework and determines the component needs to be reinspected.

7.3.2. Unit testing

Unit testing focuses on the building blocks of the software system, that is, objects and subsystems. There are three motivations behind focusing on components. First, unit testing reduces the complexity of the overall test activities, allowing us to focus on smaller units of the system. Second, unit testing makes it easier to pinpoint and correct faults, given that few components are involved in the test. Third, unit testing allows parallelism in the testing activities; that is, each component can be tested independently of one another.

The specific candidates for unit testing are chosen from the object model and the system decomposition of the system. In principle, all the objects developed during the development process should be tested, which is often not feasible because of time and budget reasons. The minimal set of objects to be tested should be those objects that are participating objects in use cases. Subsystems should be tested after each of the objects/classes within that subsystem have been tested individually.

Existing subsystems, which were reused or purchased, should be treated as components with unknown internal structure. This applies in particular to commercially available subsystems, where the internal structure is not known or available to the developer. Many unit testing techniques have

been devised. Below, we describe the most important ones: equivalence testing, boundary testing, path testing, and state-based testing.

Equivalence testing

Equivalence testing is a black box testing technique that minimizes the number of test cases. The possible inputs are partitioned into equivalence classes, and a test case is selected for each class. The assumption of equivalence testing is that systems usually behave in similar ways for all members of a class. To test the behavior associated with an equivalence class, we only need to test one member of the class. Equivalence testing consists of two steps: identification of the equivalence classes and selection of the test inputs. The following criteria are used in determining the equivalence classes.

- **Coverage**. Every possible input belongs to one of the equivalence classes.
- **Disjointedness**. No input belongs to more than one equivalence class.
- **Representation**. If the execution demonstrates an error when a particular member of a equivalence class is used as input, then the same error can be detected by using any other member of the class as input.

Boundary testing

Boundary testing is a special case of equivalence testing and focuses on the conditions at the boundary of the equivalence classes. Rather than selecting any element in the equivalence class, boundary testing requires that the elements be selected from the "edges" of the equivalence class. The assumption behind boundary testing is that developers often overlook special cases at the boundary of the equivalence classes (e.g., 0, empty strings, year 2000).

Path testing

Path testing is a white box testing technique that identifies faults in the implementation of the component. The assumption behind path testing is that, by exercising all possible paths through the code at least once, most faults will trigger failures. The identification of paths requires knowledge of the source code and data structures.

The starting point for path testing is the flow graph. A flow graph consists of nodes representing executable blocks and associations representing flow of control. A basic block is a number of statements between two decisions. A flow graph can be constructed from the code of a component by mapping decision statements (e.g., if statements, while loops) to nodes lines.

Statements between each decision point (e.g., then block, else block) are mapped to other nodes. Associations between each node represent the precedence relationships.

State-based testing

Object-oriented languages introduce the opportunity for new types of faults in object-oriented systems. Polymorphism, dynamic binding, and the distribution of functionality across a larger number of smaller methods can reduce the effectiveness of static techniques such as inspections [Binder, 1994].

State-based testing [Turner & Robson, 1993] is a recent testing technique, which focuses on object-oriented systems. Most testing techniques focus on selecting a number of test inputs for a given state of the system, exercising a component or a system, and comparing the observed outputs with an oracle. State-based testing, however, focuses on comparing the resulting state of the system with the expected state. In the context of a class, state-based testing consists of deriving test cases from the UML state chart diagram for the class. For each state, a representative set of stimuli is derived for each transition (similar to equivalence testing). The attributes of the class are then instrumented and tested after each stimuli has been applied to ensure that the class has reached the specified state.

Summary

Reliability is a measure of success with which the observed behavior of a system conforms to the specification of its behavior. **Failure** is any deviation of the observed behavior from the specified behavior. **Error** means the system is in a state such that further processing by the system will lead to a failure. **Fault** is the mechanical or algorithmic cause of an error.

Following are the techniques to avoid fault:

- Development methodologies: - avoid faults by providing techniques that minimize fault introduction in the system models and code.
- Configuration management avoids fault caused by undisciplined change in the system models.
- Verification attempts to find faults before any execution of the system.
- Review is the mutual inspection of parts or all aspects of the system without actually executing the system.

- Following are the summary of fault detection techniques: Debugging assumes that faults can be found by starting from an unplanned failure. Testing is a fault detection technique that tries to create failures or errors in a planned way.

Unite Testing tries to find faults in participating objects and /or subsystems with respect to the use cases from the use case model. **Integration Testing** is the activity of finding faults when testing the individually tested components together. **System Testing** tests all the components together. **Functional testing** tests the requirements from the RAD. **Performance testing** checks the nonfunctional requirements and additional design goals from the SDD. **Acceptance testing** and **installation testing** check the requirements against the project agreement and should be done by the client.

Inspecting components, unit testing, integration testing, and system testing are the activities of system testing. **Equivalence testing** is a black box testing technique that minimizes the number of test cases. **Boundary testing is a** special case of equivalence testing and focuses on the conditions at the boundary of the equivalence classes. **Path testing** is a white box testing technique that identifies faults in the implementation of the component. State-based testing focuses on comparing the resulting state of the system with the expected state.

Review Questions & Problems

1. _____ is unacceptable behavior exhibited by a system.
   a. Error
   b. Failure
   c. Fault
   d. None
2. One is a technique that is used to avoid faults from a system which avoids faults caused by undisciplined change in the system models.
   a. Configuration management
   b. Verification technique
   c. Review
   d. Development methodologies
3. List and describe briefly the different types of debugging?
4. What is a bug and why is it being used in programming?
5. Discuss the different types of Testing
6. What is a black box testing technique and what makes it different from white box testing?

7. A state-base testing a recently introduced technique to test a system which is different from other types of test. Explain in detail what makes state-based testing different from other type of tests.

References:

- *Brahmin, Ali(1999), Object oriented System development , McGraw Hill, USA*
- *OOSE Practical software development using UML and Java*
- *OOSE 8th edition Stephen R. Schach*

REFERENCES

- The Unified Modeling Language User Guide SECOND EDITION Addison Wesley Professional USA
- Applying UML and Patterns 2nd edition, Craig Larman
- Scott, Kendall (2004) Fast Track UML 2.0 Après USA
- Brahmin, Ali(1999), Object oriented System development , McGraw Hill, USA
- OOSE Practical software development using UML and Java
- OOSE 8th edition Stephen R. Schach
- www.tutorialspoint.com, retrieved on June 2017