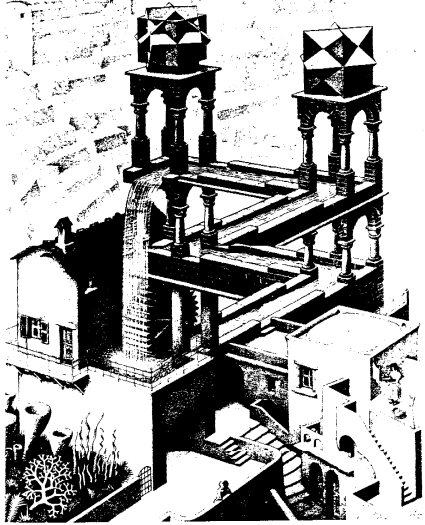


## 3. Software Life Cycle

---



---

*Software life cycle* refers to the set of activities that constitute a software project. A software life cycle starts with concept exploration and ends with the retirement of the system or with the cancellation of the project. In small projects, the software life cycle is implicit: a programmer experiences a problem and solves it by developing a small program. Once the program served its purpose, it is removed and forgotten. In larger project, the life cycle needs to be explicit: as activities are assigned to different persons, it becomes critical that all participants share a common view of the execution of the project such that they can produce the right information to the right participants in a timely manner.

A *software life cycle model* is a framework providing the ordering and dependencies of life cycle activities. As the complexity of the project increases, managing dependencies among activities can significantly impact the success and duration of the project. For example, a change in requirements during implementation may invalidate a substantial amount of work and delay the delivery of the system by several months. Different life cycle models prescribe different actions to handle such changes.

In this chapter, we describe the typical activities and products of a software development project as defined by IEEE standard 1074 [IEEE 1074, 1995]. We then discuss several life cycle models that describe the dependencies across these activities. We illustrate these concepts by describing the life cycle and products of the PROSE project.

### 3.1. Introduction

A software life cycle is the sequence of activities associated with a software project. For example, Figure 22 depicts an activity diagram for a simple life cycle (e.g., a shrinkwrapped software development). In this view, the project life cycle consists of the activities Problem definition, System development, and System operation.

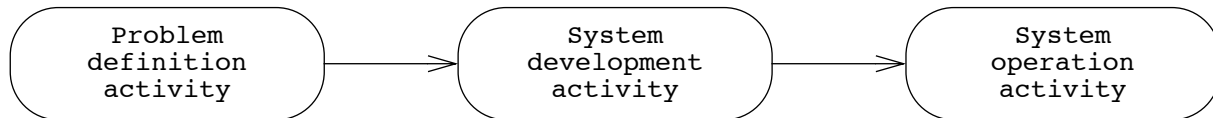


FIGURE 22. Simple life cycle of a project (UML activity diagram).

Note that, as a system can be described by many different system models, a project can be described by many different life cycles. Figure 23 is an alternate view of the same shrinkwrapped software project. In this view, the project life cycle consists of the activities: System development, Market creation, and System upgrade.

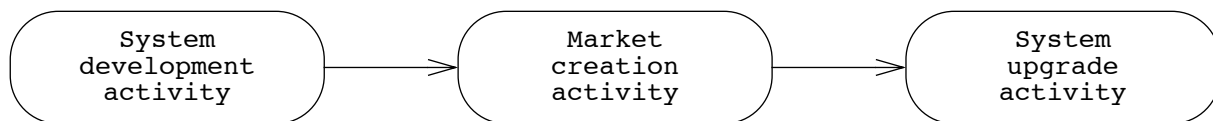
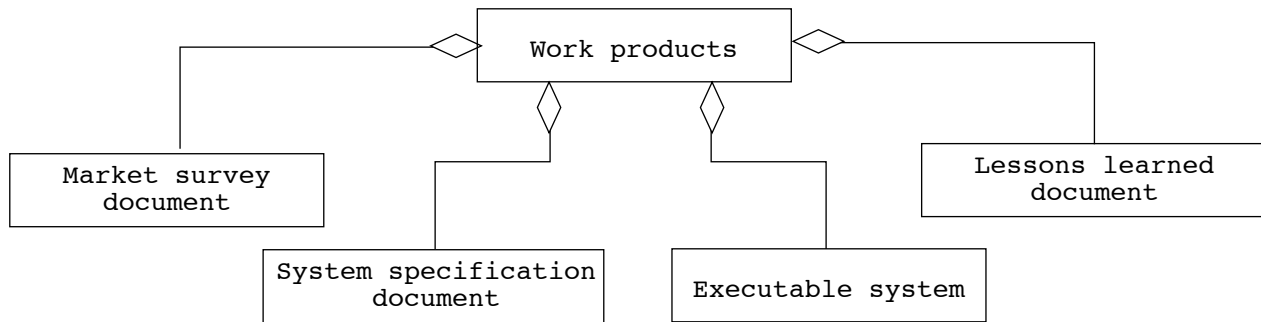


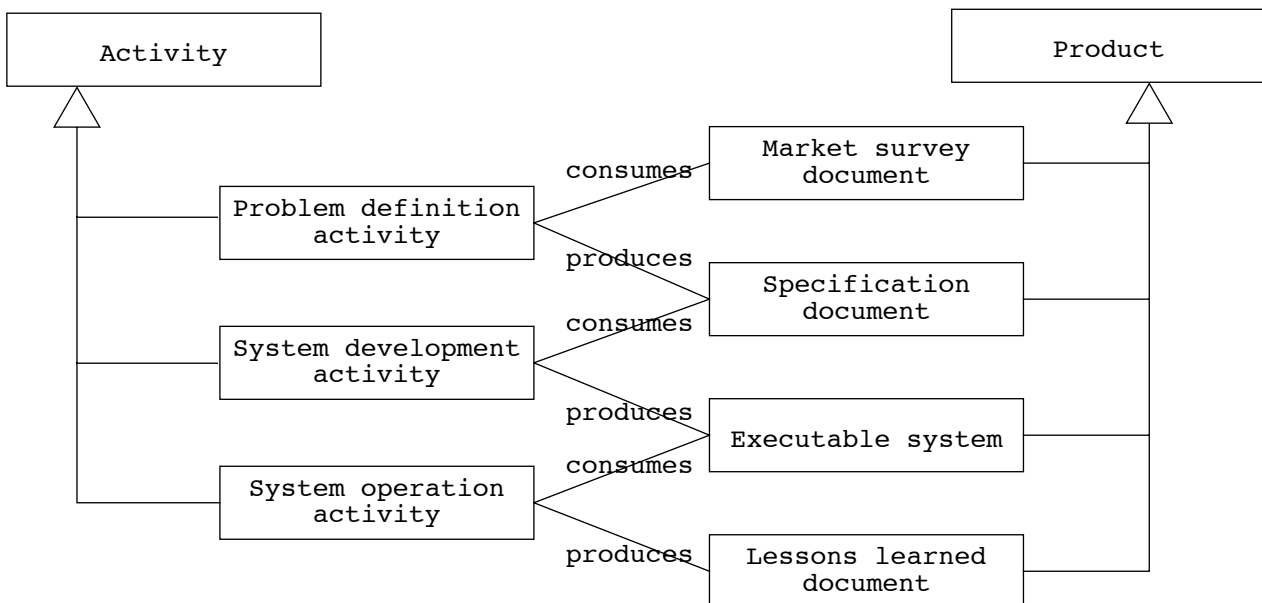
FIGURE 23. Another simple life cycle of a project (UML activity diagram).

A life cycle view of a software project emphasizes the activities and tasks in the project. Neither Figure 22 nor Figure 23 display the products created during development. Alternatively, a software project can be viewed as a set of products. The life cycle view is often called *activity-centered* while a view emphasizing products is called *entity-centered*. For example, Figure 24 is an entity centered view of the project depicted by Figure 22. Problem definition uses a Market survey document as input and generates a System specification document. System development takes the System specification document as input and produces an Executable system. System operation generates a Lessons learned document that will be used during the next product development.



**FIGURE 24.** Entity-centered view of a project (UML class diagram).

The activity-centered view and the entity-centered view are complementary, as illustrated by Figure 25. For every product there is one or more activity that generates it. Alternatively,



**FIGURE 25.** Activities and products (UML class diagram).

every activity generates one or more products. The activity-centered view is used for modeling the process and, more generally, for project management. The entity-centered view is used for modeling products, for example, in the context of development

methodologies (see *Chapter 6, Requirements Elicitation*, *Chapter 7, Requirements Analysis*, and *Chapter 8, System Design*).

In this chapter, we describe the life cycle activities defined by the standard IEEE 1074 [IEEE 1074, 1995] (Section 3.2). This standard introduces precise definitions that enable all project participants to understand and communicate effectively about the life cycle. For example, we consistently use the terms *processes*, *activities*, *tasks*, instead of the terms *stage*, *step*, or *phase*. Activities are groups of tasks that generate a product. Often activities also consume products from other activities. In Section 3.2, we also describe the information flows between activities and the communication support needed by project participants.

Note that IEEE 1074 does not impose any order between activities. In Section 3.3, we survey several life cycle models that propose different ordering of activities. In particular, we discuss the waterfall model [Royse, 1970], the spiral model [Boehm, 1987], the V model, the sawtooth and shark models, and Objectory [Rational, 1996]. In Section 3.4, we describe the life cycle model selected for the PROSE project, its activities, products, and information flow. In the next chapter, we discuss the problem of managing these activities, that is, the planning, monitoring, and controlling of these activities.

## 3.2. IEEE 1074: Standard For Developing Life Cycle Processes

IEEE 1074-1995 [IEEE 1074, 1995], *Standard For Developing Life Cycle Processes*, provides “the set of Activities that constitute the Processes that are mandatory for the development and maintenance of software.” It establishes a common framework for developing life cycles and shows mappings into typical software life cycles. Work on this standard was initiated in 1984. The first version of the standard was approved in September 1991. It was revised to correct trivial errors and was submitted for ballot in August, 1993, and approved September, 1995.

In this section, we summarize the main processes and activities introduced by the standard and clarify its fundamental concepts using UML diagrams.

### 3.2.1. Processes and activities

A *process* is a set of activities that is performed towards a specific purpose (e.g., requirements, management, delivery). IEEE 1074 includes six groups of processes consisting of a total of 17 processes (see Table 1). These include, for example:

- the *Requirements Process*, during which the client and the developers describe the system,
- the *Design Process*, during which developers decompose the system into components,

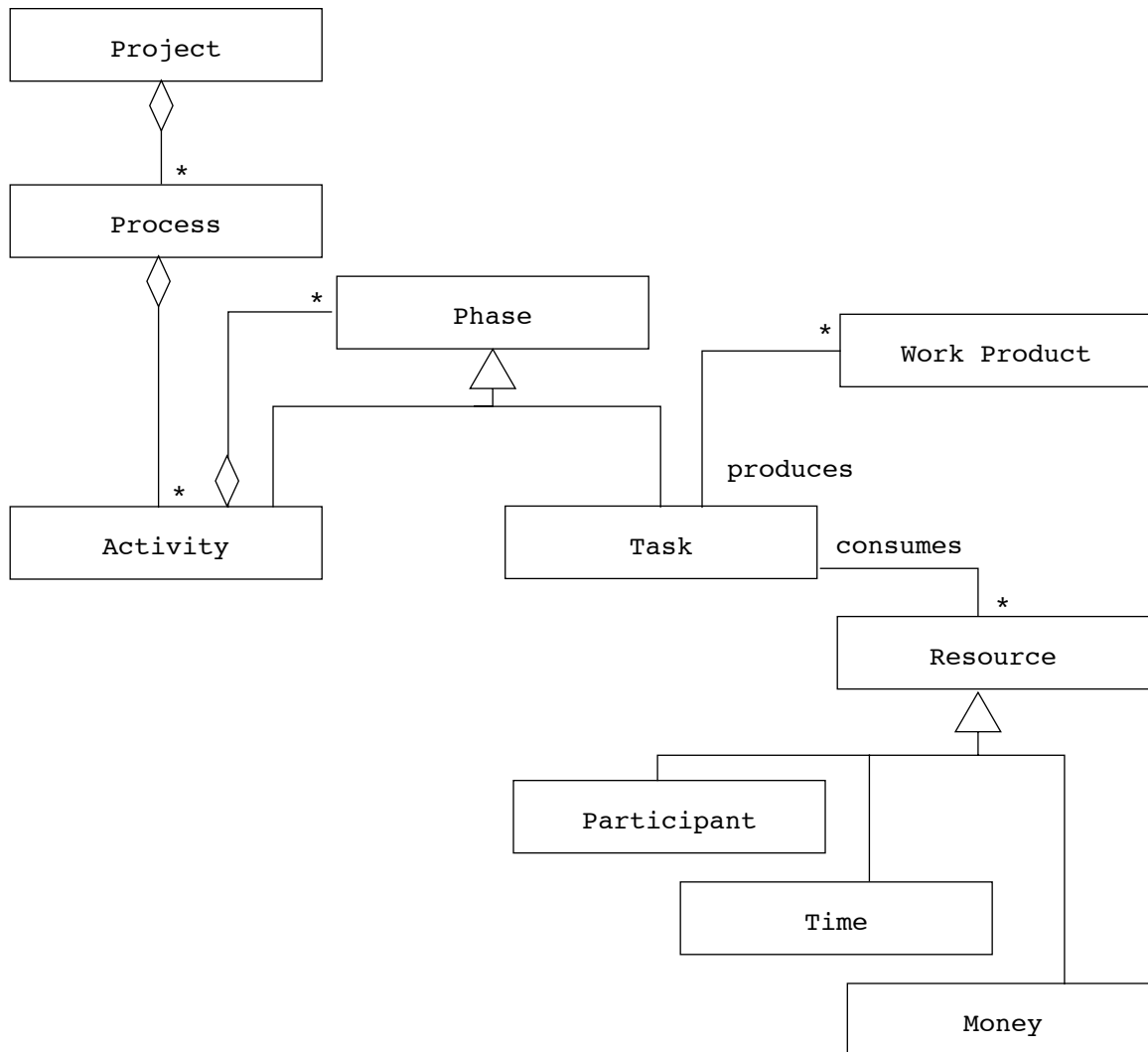
- the *Implementation Process*, during which developers realize each component,
- the *Project Monitoring and Control Process*, during which management monitors project progress and takes corrective actions.

Processes are composed of activities. An *activity* is a task or group of tasks that are assigned to project participants to achieve a specific purpose. The *Requirements Process*, for example, is composed of three activities:

- *Define and Develop Software Requirements*, during which the functionality of the system is precisely defined,
- *Define Interface Requirements*, during which the interactions between the system and the user are precisely defined, and
- *Prioritize and Integrate Software Requirements*, during which all requirements are integrated for consistency, and prioritized by client preference.

Tasks consume resources (e.g., personnel, time, money) and produce a work product. During planning, activities are decomposed into project specific tasks, are given a start and ending date and assigned to a team or a project participant (see Figure 26). During the project, actual work is tracked against planned tasks, and resources are reallocated to respond to problems.

The processes required by IEEE 1074 are listed in Table 1. Note that in the project course associated with this book, the first six processes (i.e., the *Life Cycle Modeling Process*, the *Project Management Processes*, and the *Pre-Development Processes*) have already been initiated.



**FIGURE 26.** Processes, activities, and tasks (UML class diagram). A process is performed to accomplish a specific purpose (e.g., requirements, design, installation). Processes are composed of activities which are in turn of sub activities or tasks. Tasks represent the smallest piece of work that is relevant to management (e.g., that can be assigned, scheduled, and tracked). Tasks consume resources and produces one or more work products.

Never the less, we describe these in the next sections (see Sections 3.2.2, 3.2.3, and 3.2.4) so you understand where your involvement with the project starts.

**Table 1 Software Processes in the IEEE Standard 1074**

Process group	Processes
Life Cycle Modeling (Section 3.2.2)	Selection of Life Cycle Modeling
Project Management (Section 3.2.3)	Project Initiation Project Monitoring and Control Software Quality Management
Pre-development (Section 3.2.4)	Concept Exploration System Allocation
Development (Section 3.2.5)	Requirements Analysis Design Implementation
Post-Development (Section 3.2.6)	Installation Operation and Support Maintenance Retirement
Cross-Development (Integral Processes) (Section 3.2.7)	Verification and Validation Software Configuration Management Documentation Development Training

### 3.2.2. Life cycle modeling

During *Life Cycle Modeling*, management selects and orders the activities defined in IEEE 1074 for a specific project. Not all projects require the same activities and the same ordering. For example, projects that do not require a database need not execute the activity *Design Data Base*. Similarly, if developers adopt a prototyping approach, all development activities will occur in parallel. The selected life cycle model serves as input to the *Project Initiation Process* described in the next section. We provide an example of *Life Cycle Modeling* in Section 3.4.1.

### 3.2.3. Project management

During *Project Management*, management initiates, monitors, and controls the project throughout the software life cycle. *Project Management* consists of three processes.

The *Project Initiation Process* creates the infrastructure for the project. During this process the task plan, schedule, budget, organization, and project environment are defined. The project environment includes project standards, communication infrastructure, meeting and reporting procedures, development methodology, and development tools. Most of the information generated during this process is documented in the Software Project Management Plan (SPMP). The *Project Initiation Process* completes as soon as a stable environment for the project is established.

The *Project Monitoring and Control Process* ensures that the project is executed according to the task plan and budget. If management observes any deviation from the schedule, it will take corrective action such as reallocating some of the resources, changing procedures, or replanning. The Software Project Management Plan is updated to reflect any change. The *Project Monitoring and Control Process* is active throughout the life cycle.

The *Software Quality Management Process* ensures that the system under construction meets the required quality standards (selected during Project Initiation). This process is usually executed by a different team than the developers to avoid conflicts of interest (i.e., the goal of the developers is to complete the system on time, the goal of the quality management team is to ensure that the system is not considered complete until it meets the required quality standard). The *Software Quality Management Process* is active throughout most of the life cycle.

**Table 2 Project Management Processes**

Process	Clause <sup>a</sup>	Activities
Project Initiation	3.1.3	Map Activities to Software Life Cycle Model
	3.1.4	Allocate Project Resources
	3.1.5	Establish Project Environment
	3.1.6	Plan Project Management
Project Monitoring and Control	3.2.3	Analyze Risks
	3.2.4	Perform Contingency Planning
	3.2.5	Manage the Project
	3.2.6	Retain Records
	3.2.7	Implement Problem Reporting Model
Software Quality Management	3.3.3	Plan Software Quality Management
	3.3.4	Define Metrics
	3.3.5	Manage Software Quality
	3.3.6	Identify Quality Improvement Needs

a. The 'Clause' column in this table and the other tables in this chapter is a clause number in IEEE 1074. This is a cross reference to the standards document as published in [IEEE 1074, 1995]



We describe in detail the activities of *Project Initiation* and *Project Monitoring and Control* related to planning, organization, and tracking in Chapter 4, *Project Management*. The activity *Establish Project Environment* requires particular attention in the context of a team-based project such as PROSE. One of the critical parts of the project environment is the communication infrastructure that will support information dissemination among the participants. For the project to be able to react rapidly to changes and report problems without introducing an unreasonable overhead, all project participants need to be aware of the information flow through the project and the mechanisms for disseminating information. Chapter 5, *Project Communication* describes in detail the activities related to defining and using the communication infrastructure. Note that in order to define the development team structure, and thus, the communication infrastructure, the system architecture (produced by the *System Allocation Process* described in the next section) needs to be defined.

### 3.2.4. Pre-development

During *Pre-Development*, management (or marketing) and a client identify an idea or a need. This can be a new development effort (greenfield engineering), or a change to the interface of an existing system (interface engineering) or software replacement of an existing business process (re-engineering). The *System Allocation Process* establishes the system architecture, in particular the subsystem decomposition and identifies the hardware, software and operational requirements. Note that the subsystem decomposition is the foundation of the communication infrastructure among the project members. The requirements, subsystem decomposition and communication infrastructure are described in the Problem Statement<sup>1</sup> which serves as input into the development process. The Problem Statement is written in terms of scenarios and user interface sketches that are refined into a complete use case model during the *Requirements Analysis Process* briefly described in the next section. The creation of scenarios and use cases is examined in detail in Chapter 6, *Requirements Elicitation*.

### 3.2.5. Development

*Development* consists of the processes directed toward the construction of the system.

The *Requirements Analysis Process* takes the system requirements in terms of high-level functional requirements and produces a complete specification of the system. In Objectory, this is done by developing a use case model, a domain model, and an analysis model. The *Requirements Analysis Process* is described in detail in Chapter 7, *Requirements Analysis*.

---

1. The Statement of Need mentioned in the IEEE 1074 is similar to the problem statement, but does not contain any project organization information.

**Table 3 Pre-Development Processes**

Process	Clause	Activities
Concept Exploration	4.1.3	Identify Ideas or Needs
	4.1.4	Formulate Potential Approaches
	4.1.5	Conduct Feasibility Studies
	4.1.6	Plans System Transition (If Applicable)
	4.1.7	Refine and Finalize the Idea or Need
System Allocation	4.2.3	Analyze Functions
	4.2.4	Develop System Architecture
	4.2.5	Decompose System Requirements <sup>a</sup>

a. System requirements are decomposed into Hardware Requirements and Software Requirements. Hardware development is not addressed in the IEEE 1074 standard.

The *Design Process* takes the architecture produced during the *System Allocation Process* (see Section 3.2.4) and produces a coherent and well-organized representation of the system. The high level design is completed (*Perform Architectural Design* and *Design Interfaces* activities) and results into the refinement of the subsystem decomposition. This also includes the allocation of requirements to hardware and software systems, the description of boundary conditions, the selection of off-the-shelf components and the definition of design goals. The detailed design of each subsystem is done during the *Perform Detailed Design* activity.

In Objectory, the *Design Process* results into the definition of design objects, their attributes and operations, and their organization into packages. By the end of this activity, all methods and their type signatures are defined. New classes are introduced to take into account non functional requirements and component specific details. Architectural and detailed design are described in detail in Chapter 8, *System Design*.

The *Implementation Process* takes the design model and produces an equivalent executable representation. The *Implementation Process* includes integration planning, and integration activities. Note that tests performed during this process are independent of those performed during quality control (see Section 3.2.3) or *Verification and Validation* (see Section 3.2.7). The activities of the *Implementation Process* and their relationship to the *Design Process* are described in more detail in Chapter 8, *System Design*.

### 3.2.6. Post-development

*Post-Development* consists of installation, maintenance, operation and support, and retirement processes.

During *Installation* the system software is distributed and installed at the client site. The installation culminates in the client acceptance test according to the criteria defined in the

**Table 4 Development Processes**

Process	Clause	Activities
Requirements Analysis	5.1.3	Define and Develop Software Requirements
	5.1.4	Define Interface Requirements
	5.1.5	Prioritize and Integrate Software Requirements
Design	5.2.3	Perform Architectural Design
	5.2.4	Design Data Base (If Applicable)
	5.2.5	Design Interfaces
	5.2.6	Select or Develop Algorithms (If Applicable)
	5.2.7	Perform Detailed Design
Implementation	5.3.3	Create Test Data
	5.3.4	Create Source
	5.3.5	Generate Object Code
	5.3.6	Create Operating Documentation
	5.3.7	Plan Integration
	5.3.8	Perform Integration

**Table 5 Post-Development Processes**

Process	Clause	Activities
Installation	6.1.3	Plan Installation
	6.1.4	Distribution of Software
	6.1.5	Installation of Software
	6.1.7	Accept Software in Operational Environment
Operation and Support	6.2.3	Operation the System
	6.2.4	Provide Technical Assistance and Consulting
	6.2.5	Maintain Support Request Log
Maintenance	6.3.3	Reapply software life cycle.
Retirement	6.4.3	Notify Users
	6.4.4	Conduct Parallel Operations (If Applicable)
	6.4.5	Retire System

project agreement. *Operation and Support* is concerned with user operation of the system and user training. *Maintenance* is concerned with the resolution of software errors, faults and failures after the delivery of the system. *Maintenance* requires a ramping of the software life cycle processes and activities into a new project. *Retirement* removes an existing system terminating its operations or support. *Retirement* takes place when the system is upgraded or replaced by a new system. To ensure a smooth transition between the two systems, both systems are often run in parallel until the users have gotten used to the new system. Except for client delivery and acceptance, we do not address the post-development processes in this book.

### 3.2.7. Cross-Development (Integral Processes)

Several processes take place during the complete duration of the project. These are integral processes, which we call *cross-development processes* in the following. Cross-development process include *Validation and Verification*, *Software Configuration Management*, *Documentation Development*, and *Training*.

**Table 6 Cross-Development Processes (called Integral Processes in IEEE 1074)**

Process	Clause	Activities
Verification and Validation	7.1.3	Plan Verification and Validation
	7.1.4	Execute Verification and Validation Tasks
	7.1.5	Collect and Analyze Metric Data
	7.1.6	Plan Testing
	7.1.7	Develop Test Requirements
	7.1.8	Execute the Tests
Software Configuration Management	7.2.3	Plan Configuration Management
	7.2.4	Develop Configuration Identification
	7.2.5	Perform Configuration Control
	7.2.6	Perform Status Accounting
Documentation Development	7.3.3	Plan Documentation
	7.3.4	Implement Documentation
	7.3.5	Produce and Distribute Documentation
Training	7.4.3	Plan the Training Program
	7.4.4	Develop Training Materials
	7.4.5	Validate the Training Program
	7.4.6	Implement the Training Program

*Verification and Validation* includes verification and validation tasks. Verification tasks focus on showing that the work products and the system comply with the specification. Verification includes reviews, audits, and inspections. Validation tasks focus on ensuring that the system addresses the client's need. Validation includes system testing, beta testing, and client acceptance testing. Verification and Validation activities occur throughout the life cycle with the intent of detecting anomalies as early as possible. For example, in Objectory, each model is reviewed against a checklist at the end of the process that generated it. The review of a model, say the design model, may result in the modification of models generated in other processes, say the analysis model. The activity *Collect and Analyze Metric Data* may also serve for future projects and contribute to the knowledge of the organization. The activities *Plan Testing and Develop Test Requirements* can be initiated as early as the completion of the requirements analysis. In large projects, these tasks are performed by different participants than the developers.

Reviews, audits and inspections are described in Chapter 5, *Project Communication*. The review of requirements, analysis, and design models is described in Chapters 6, 7, and 8, respectively. Testing activities are described in Chapter 11, *Testing*.

The *Configuration Management Process* focuses on the tracking and control of changes of work products. Items under configuration management include the source code for the system, all development models, the software project management plan, and all documents visible to the project participants.

The *Documentation Process* deals with the work products (excluding code) documenting the results produced by the other processes. IEEE 1074 does not prescribe specific documents. The planning and selection of document templates is done during this activity.

For example, in PROSE, the pre-development processes produce the Problem Statement. The project initiation process produces the initial version of the Software Project Management Plan. These documents are available before the project starts. The *Requirements Analysis Process* describes the current and desired system and produces several system models. These are described in the Requirements Analysis Document. The *Architectural Design* and *Identify System Requirements* activities produce the System Design Document.

Configuration management and documentation are described in detail in Chapter 12, *Configuration Management*, and Chapter 13, *Documentation*, respectively. The distribution of documents to project participants is described in Chapter 5, *Project Communication*.

### 3.2.8. Communication in the software life cycle

Figure 27 depicts the information flow among processes. Each arrow represents a work product that is generated by a process and consumed by another process. Each arrow represents a formal communication channel between project participants supported by the exchange of documents, models, or code. In addition to formal communication, project participants communicate informally to address transient information needs during change and crises. Several activities, such as “5.3.8 Perform Integration,” have stringent informal communication requirements as they address unforeseen problems. Finally, several activities in the IEEE 1074 model address exclusively communication and information dissemination (e.g., “3.2.7 Implement Problem Reporting Method,” “7.2.6 Perform Status Accounting,” and “7.3.5. Produce and Distribute Documentation”). When a project encompasses multiple teams, as the PROSE project does, communication becomes critical and is often a bottleneck. Chapter 5, *Project Communication*, addresses these issues and focuses on the design and installation of the communication infrastructure.

As suggested by Figure 27, the information flow in a software development project can be complex. The next section describes different life cycle models that attempt to address this complexity by serializing activities.

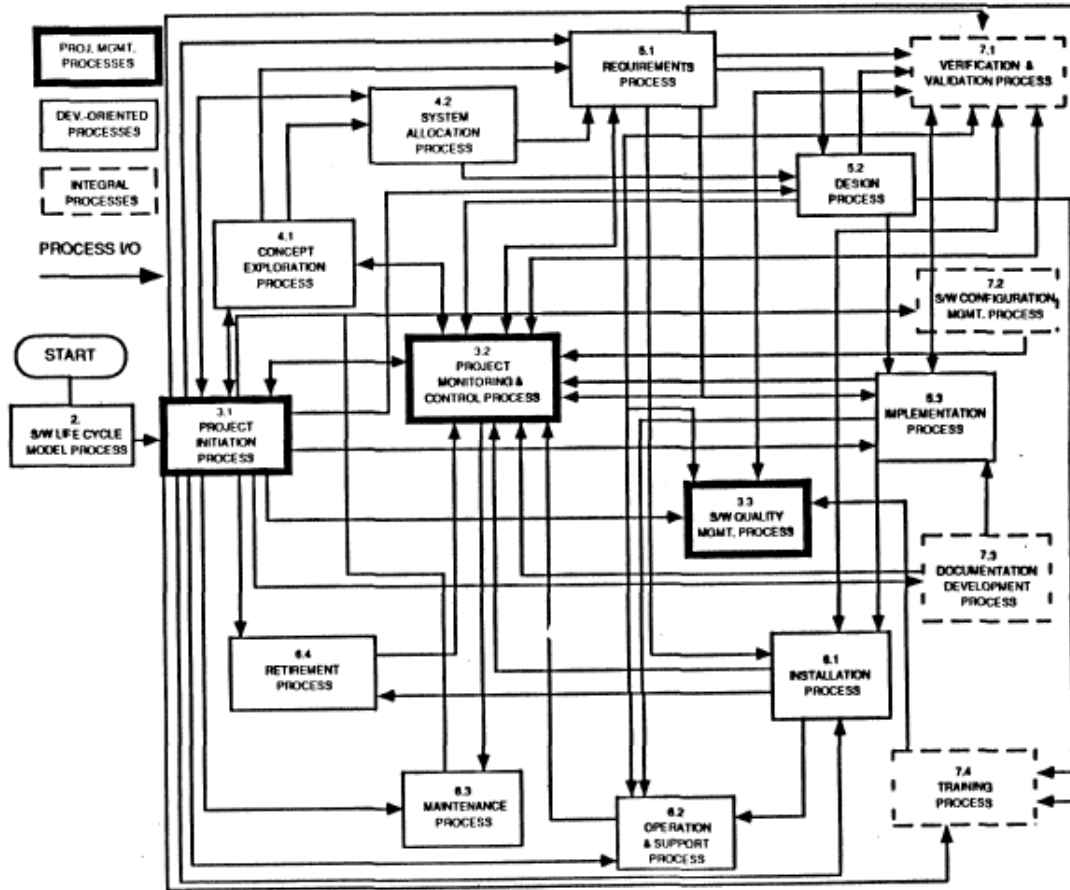


FIGURE 27. Process interrelationships (from [IEEE 1074, 1995]). As suggested by this picture, dependencies among processes and activities are complex and seldom allow a sequential execution of processes.

### 3.3. Life Cycle Models

We have now defined a set of processes and activities that occur in the software life cycle. How are they related to each other? What are their dependencies? In what order can we

schedule them? That is, in what order should these activities be performed and managed to deliver a high quality system within budget and within time?

There are no single answers for these questions. First, different projects have different process requirements. For example, if the application domain is well known, as in the case of a business re-engineering project, the ordering of the development activities may be sequential, given that developers may require minimum training. In another instance, a first of kind project may require substantial prototyping in which case the development processes will be executed concurrently and the management and configuration management processes will be much more complex. In a third instance, an air traffic control project may require stringent quality assurance and validation processes. For these reasons, IEEE 1074 does not dictate a specific life cycle, but rather, provides a template that allows many different life cycle models.

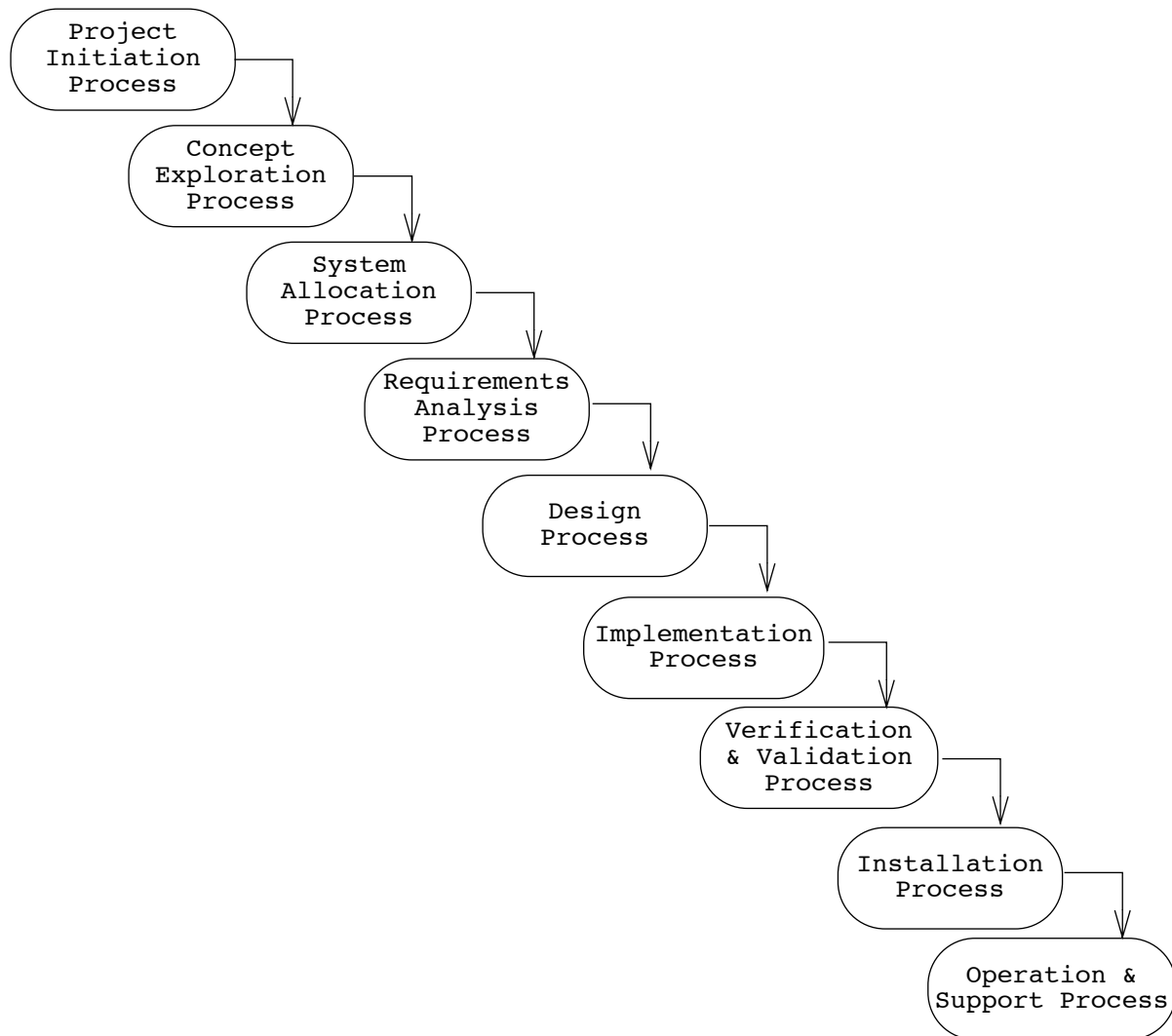
In this section, we review various life cycle models that have been proposed in the literature and others that are used in practice. Most of these models focus on the development processes exclusively. This section does not provide algorithmic knowledge that can be applied immediately. It is rather heuristic knowledge that you need to acquire for yourself by applying it by trial and error. The project associated with this book allows you to apply it in a complex setting.

### 3.3.1. Waterfall model

The waterfall model was first described by Royse [Royse, 1970]. The waterfall model prescribes a sequential execution of the development processes. The requirements analysis activities are all completed before the system design activity starts. The goal is to never turn back once a process is completed. The key feature of his model is the constant verification activity (called “verification step” by Royse), that ensures that each development activity does not introduce unwanted or deletes mandatory requirements. This model provides a simple (or even simplistic) view of software development that measures progress by the number of tasks that have been completed. The model assumes that software development can be scheduled as a step-by-step process that transforms user needs into code. There is an explicit set of checkpoints, reviews, and documents after each development activity, which makes it easy to manage the process.

#### *DOD 2197A*

Figure 29 shows the DOD 2167A standard life cycle model. The main characteristic of this model is that each development activity is followed by a review. The starting point in this model is the system requirements analysis activity whose goal is to generate unambiguous system requirements. The first review is the system requirements review in which the requirements are reviewed with respect to completeness, consistency, and clarity. The

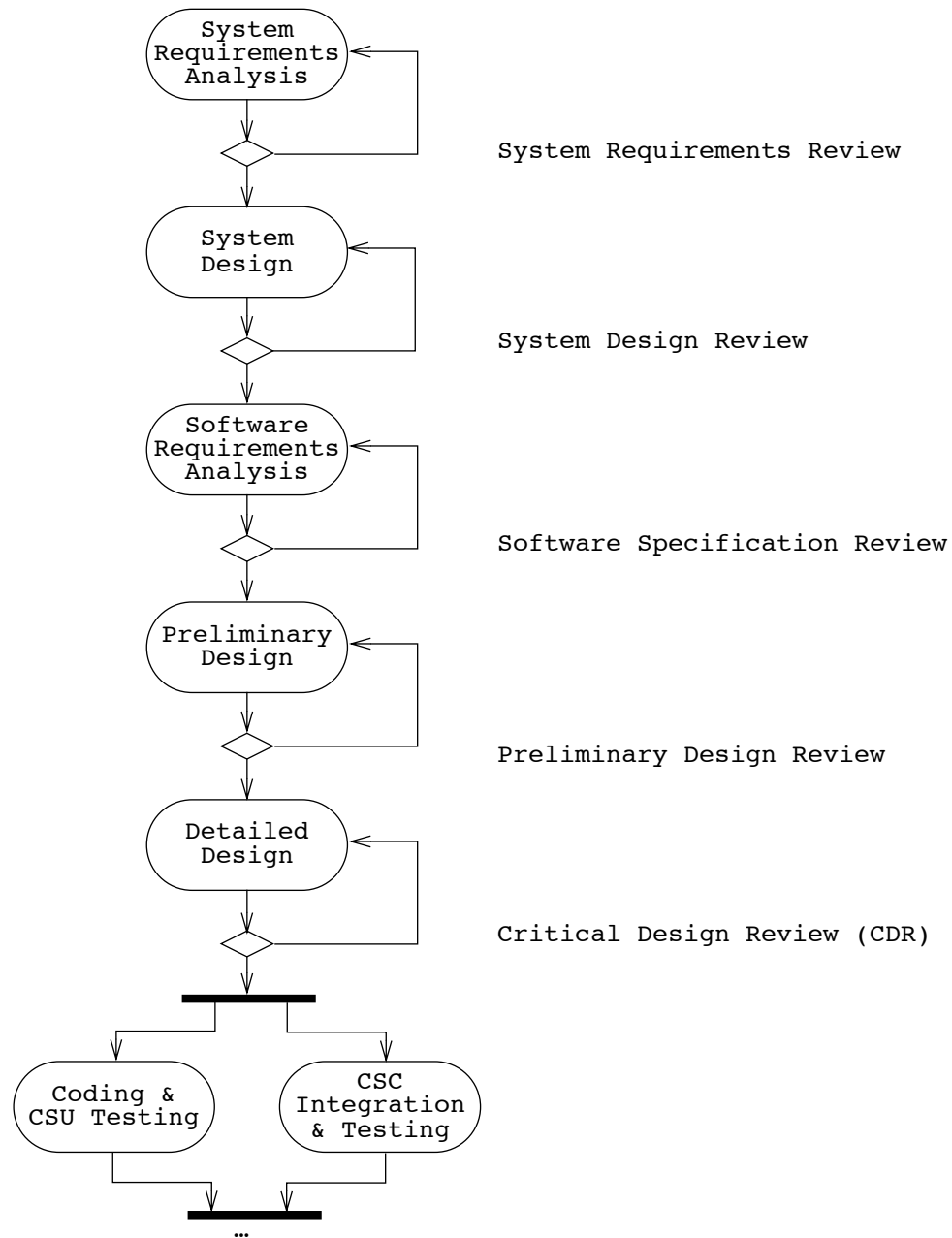


**FIGURE 28.** The waterfall model of software development: software development activities are performed in sequence (UML activity diagram of IEEE 1074 processes adapted from [Royse, 1970]; project management and cross-development processes were omitted).

system requirements are the basis for the system design activity which generates the system design. The system design is reviewed in the system design review activity.

The system design is baselined once the system design review is successfully completed. The functional baseline serves as the starting point for the software requirements analysis which creates the software requirements. The software requirements are then reviewed and





**FIGURE 29.** Waterfall model for the DOD Standard 2167A (UML activity diagram). 2167A activities are used instead of IEEE 1074 activities. Decision points denote reviews: the subsequent activity is initiated only if the review is successful.

baselined before serving as a basis for implementation. Implementation starts with the preliminary design followed by the detailed design activity. An important review is the critical design review (CDR). Coding does not occur until the CDR completes successfully.

The 2167A standard is based on functional decomposition, starting from requirements analysis where functions are specified to implementation where subroutines are combined to realize user level functions.<sup>1</sup> A 2167A compliant system consists of a set of subsystems called Computer Software Configuration Items (CSCI). A CSCI is a software subsystem (in contrast to a hardware item) that is under configuration management control. During system design the system is decomposed into a set of Computer Software Configuration Items (CSCIs) which in turn are decomposed into Computer Software Components (CSCs) during preliminary design and into Computer Software Units (CSUs) during detailed design. Several CSUs make up a Computer Software Component (see Figure 30). A CSC is defined as a functionally distinct piece of software which is part of a CSCI. The implementation occurs in terms of the CSUs. Testing occurs in three activities: CSU Testing, CSC Integration and Testing, and System Integration and Testing.

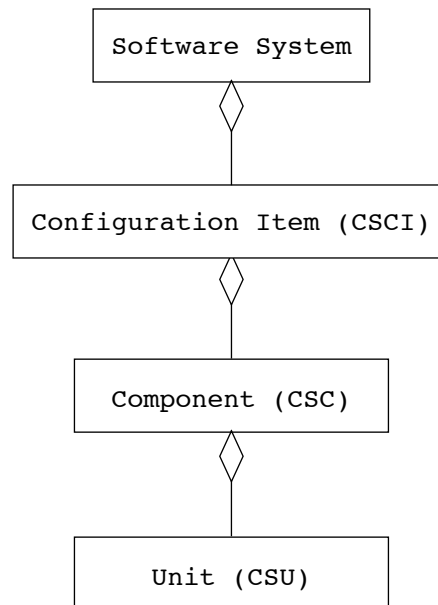
### *The V-Model*

The V-model is a variation of the waterfall model (see Figure 31). Each Development activity is paired with its corresponding Validation activity. For example, the goal of the *Unit Test* activity is to validate units against their description in the detailed design. The *Component Integration and Test* activity validates functional components against the preliminary (or high level) design. Higher levels of abstractions of the V-model deal with the requirements in terms of elicitation and operation. The middle-part of the V-model focuses on mapping the understanding of the problem into a software architecture. The lower level of the V-model focuses on details such as the assembly of software components and the coding new ones. The difference between the waterfall model and the V model is that the latter makes explicit the notion of level of abstraction. All activities from requirements to implementation consist of building more and more detailed representation of the system while all activities from implementation to operation consist of validating the system.

The waterfall model and its variants are abstractions of the software development process. In many aspects, they are simplistic. The biggest problem with these models is that they assume that after an activity is finished and reviewed the associated work product can be

---

1. Functional decomposition is a development methodology while 2167A is a life cycle model. Development methodologies are recipes indicating how the development activities should be carried out by the developers. Life cycle models are indicating how life cycle activities should be ordered. Although we attempt to describe life cycle models and development methodologies independently (given that they address different aspects of the development), development methodologies often impose constraints on the life cycle model. The 2167A life cycle was developed assuming a functional decomposition methodology. Development methodologies are described in further detail in Chapter 8, *System Design*.



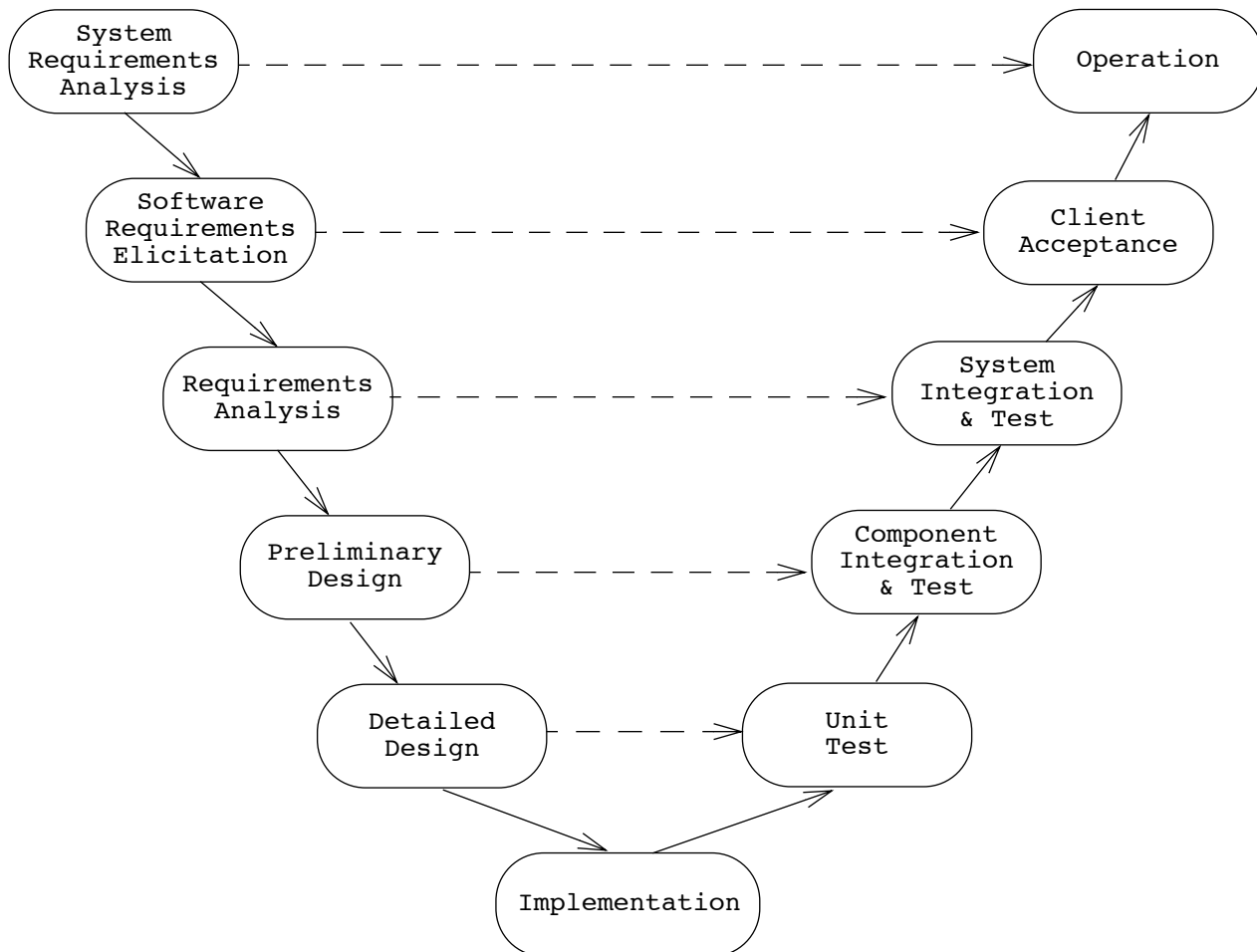
**FIGURE 30.** Levels of abstraction in 2167A (UML class diagram).

baselined. Such an idealized model is only appropriate if the specification of the requirements is of high level of assurance and the delivery conditions are easily achievable. In practice, system development rarely confirms to this ideal model and discoveries made during development require changes to work done during previous activities.

### 3.3.2. The Spiral Model

The spiral model [Boehm, 1987] was devised to accommodate change during the software development, the biggest criticism of the waterfall model. It has the same processes as the waterfall model, however, it adds several activities such as risk management, reuse, and prototyping. These extended processes are called *cycles*. or *rounds*. Each round starts with risk identification and is completed by a review involving the participants concerned with the product being developed, e.g., developers, clients, and users. This review covers the products developed during the previous and current rounds and the plans for the next round. The spiral model distinguishes the following rounds:

1. Concept of Operation
2. Software Requirements
3. Software Product Design

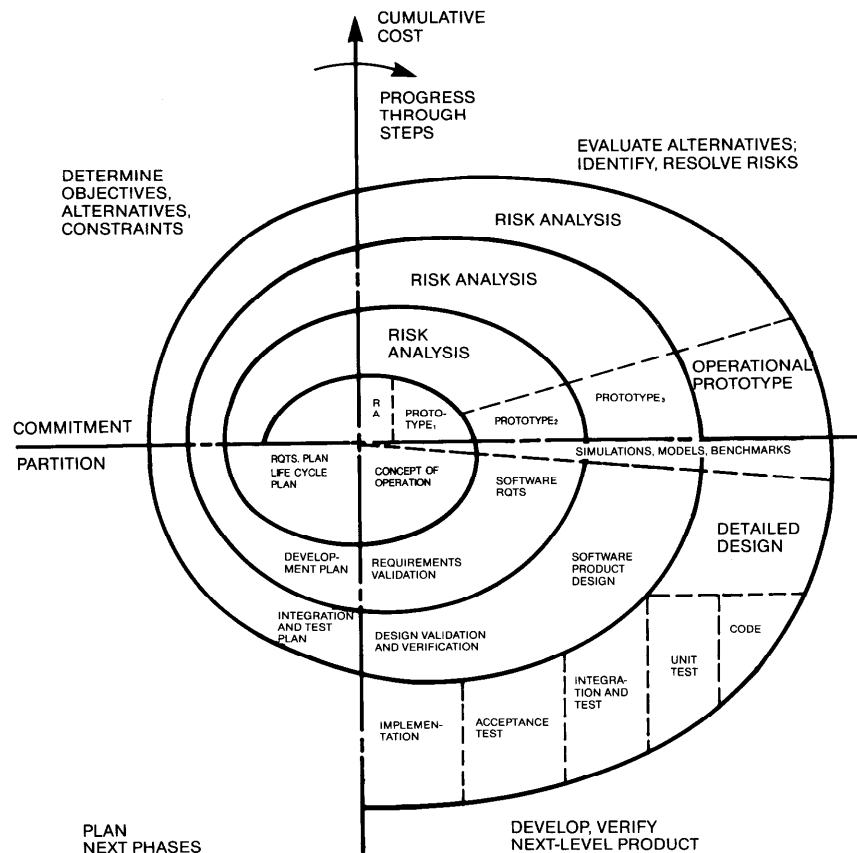


**FIGURE 31.** V-Model of Software Development (UML activity diagram; adapted from [Jensen & Tonies, 1979]). The horizontal object flow denote the information flow between activities of same abstraction level (e.g., the requirements analysis results are validated during system and integration test). The V shape layout of the activities was conserved to reflect the original drawing. However, the layout of the activities has no semantics in UML.

4. Detailed Design
5. Code
6. Unit Test
7. Integration and Test
8. Acceptance Test

## 9. Implementation

These rounds can be viewed as a spiral displayed in Figure 32.



**FIGURE 32.** Spiral Model (from [Boehm, 1987]). The first round, concept of operation, starts in the upper left quadrant. Subsequent rounds are represented by additional layers on the spiral. The distance from origin represents the cost accumulated by the project.

In Figure 32, the spiral describes the status of the project over time. The distance from the origin is the cost accumulated by the project. The angular coordinate indicates the progress accomplished within a cycle. Each cycle includes the following activities:

1. Define Objectives
2. Specify Constraints
3. Generate Alternatives

4. Identify Risks
5. Resolve Risks
6. Plan
7. Commit

The first two activities define the problem addressed by the current cycle. The third activity, Generate Alternatives, defines the solution space. The activities 4 and 5, Identify risks and Resolve Risks, serve to identify future problems that may result in high costs or, worse, in the cancellation of the project. Activity 6, Plan, is a management activity, and activity 7 is the realization of the cycle.

Note that each cycle follows the waterfall model and that the figure illustrates only the first three activities of the waterfall model (labelled Concept of Operation, Software Requirements and Software Product Design). The cycles for the remaining activities - called Detailed Design, Code, Unit Test, Integration and Test, Acceptance Test, Implementation - are not shown in detail.

### 3.3.3. The Sawtooth Model

The life cycle models we described until now emphasize the management of software developers. They do not address the needs of the customer or the user. They assume that the software requirements will not change drastically and that it is sufficient to show progress according to the development plan. The drawback of this approach is that the client and the user do not see a running system before the client acceptance test, and thus, cannot correct any requirement problems.

Users and implementors, however, have different needs when trying to understand software systems. The sawtooth model [Rowen, 1990] tries to solve this discrepancy by showing the user and software developer's perception of the system as different trajectories over time and not as the same trajectory as in the waterfall or V-model.

At the beginning of a project, both trajectories have the same starting point, namely the requirements of the system as described in the problem statement. During development these trajectories differ quite a bit. The user most probably stays at the level of requirements. Is the system realizing the required functionality? Does it perform adequately? The developers concern is about feasibility. Can the functional requirements be realized? What hardware/software needs to be developed to realize the requirements?<sup>1</sup> The software

---

1. There is actually a trajectory for each person in the system. For example, different developers will have different trajectories, depending on which team they belong to. The manager's point of view is yet another trajectory which is focusing on management: Can the requirements be decomposed such that they can be developed and test in parallel, thereby allowing faster development time?

development process has to ensure that both trajectories meet at the end of the project. The sawtooth model achieves this goal by introducing new activities. The span between the trajectories represents the gap between the user's perception of the system and the developer's perception of the system. To make sure that the trajectories meet at the end, it is wise, to introduce checkpoints during development in which the trajectories briefly meet. This is usually done by getting the client involved *at their level of abstraction*.

For example, after the requirements analysis and system design phase, the developers can prototype the screen sequences in terms of the use cases describing the functional requirements. By demonstrating this prototype to the client, the client is able to evaluate quite early in the development whether the prototype satisfies the functional requirements. By repeating this process several times during the development, the manager makes sure that the trajectories intersect several times during the development. This makes it much more probable that they meet at the end of development.

The sawtooth model is a modified V-Model that includes these intersections of trajectories. It is called sawtooth model, because each prototype demonstration results in a "tooth" in the developer trajectory. The tip of each tooth is an intersection with the client's trajectory. Figure 33 shows the sawtooth model for a development project with two prototypes, a revolutionary prototype and an evolutionary prototype.

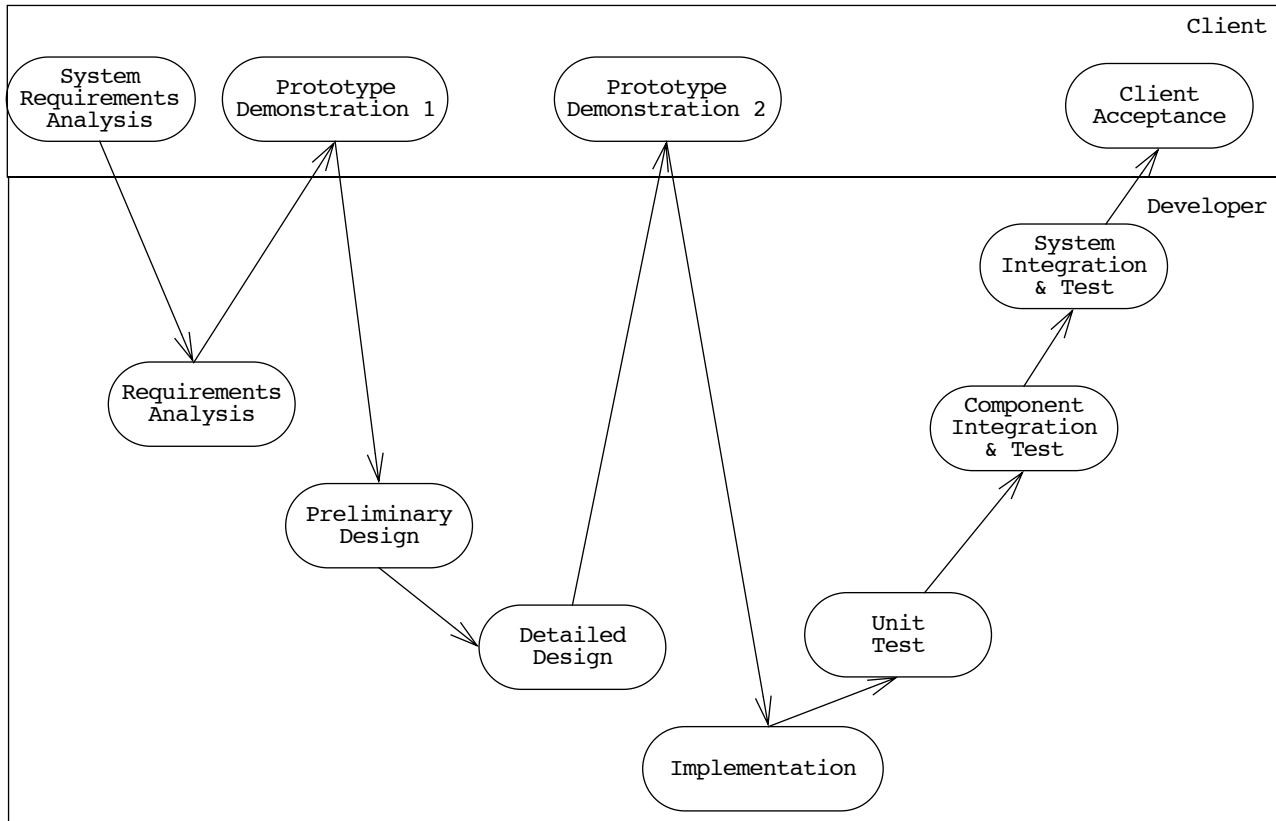
The revolutionary prototype is often an illustrative prototype because it needs to be built quickly for demonstrating the functionality of the system. There is little intent to deliver this prototype for production use.<sup>1</sup> No matter how realistic, this prototype is still only a model of the system. The scenarios demonstrated will be contrived and represent only a small fraction of the required functionality. The short cuts taken to develop a quick version would be a maintenance nightmare if this prototype was promoted to production use.

The second prototype is usually an evolutionary prototype. It is shown late in the development where some functionality has already been implemented. The primary distinction between the two types of prototyping is that the revolutionary prototype does not need an overall design whereas the evolutionary prototype does. It is in general counterproductive to insist on a complete subsystem decomposition when showing the first prototype to the user.

The tasks and activities of the prototype development process in the sawtooth model are shown in Table 7.

---

1. If the revolutionary prototyping environment is identical to the development environment, then it might be possible to reuse some parts of the revolutionary prototype during implementation. But reuse is not the goal when producing a revolutionary prototype.



**FIGURE 33.** Sawtooth Model with 2 Prototype demonstrations (UML activity diagram). The Client swimlane encloses the activities that are visible to the client, while the developer swimlane encloses the activities that are at a lower abstraction level.

**Table 7 Activities of Prototype Development Process in the Sawtooth Model**

Activity	Task
Preparation	Select prototyping environment Select functionality to be demonstrated Develop prototype Develop agenda for prototype demonstration Notify client and review agenda
Demonstration	Install prototype Demonstrate selected functionality Record minutes



**Table 7 Activities of Prototype Development Process in the Sawtooth Model**

Activity	Task
Evaluation of Feedback	Review minutes from demonstration Identify issues Discuss issues
Project Correction	Resolve open issues

### 3.3.4. Sharktooth Model

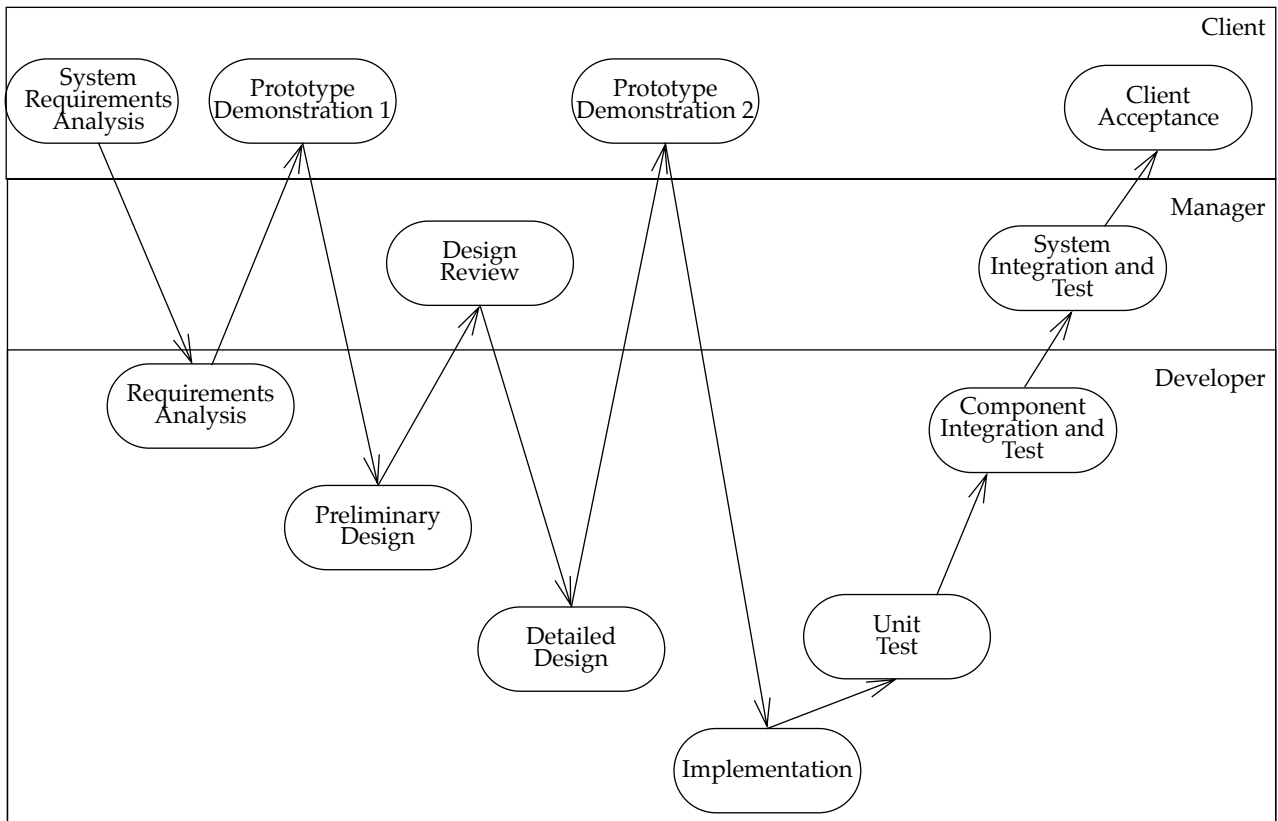
The sharktooth model is a refinement of the sawtooth model. In addition to the client demonstrations (*large teeth*) management reviews and demonstrations are introduced as well (*small teeth*).

For example, large teeth can include a functional prototype and a user interface mock-up. The former demonstrates the feasibility of the functions that will be implemented in the system while the latter illustrates the layout (or look and feel) of the user interface. Small teeth can include a system integration prototype which demonstrates the interaction between the components of the system. Such an integration prototype can be built as soon as the components are selected and need not implement any functionality. The client is of course not a good target for the demonstration of a system integration prototype. In general we target the project manager as the audience for the system integration prototype demonstration and call the associated review an internal review. The system integration prototype can be demonstrated several times during the project, each of these demonstration leading to a “small tooth”.

To describe the demonstrations for management, we add another swimlane to the sawtooth model that depicts the level of understanding of the project manager (see Figure 34). The project manager’s swimlane is shown between the client and the developer swimlanes. Demonstrating the system integration prototype is a tooth involving the developers and the project manager. The sharktooth model assumes that the project manager is interested in system design and object design issues and thus wants to reach deeper level of system understanding than the client, and neither wants to follow the developer to the lowest levels of detail in the system. Small teeth are internal reviews involving a prototype demonstration for the project manager. Large teeth are prototype demonstrations for the client.

### 3.3.5. Objectory Life Cycle Model

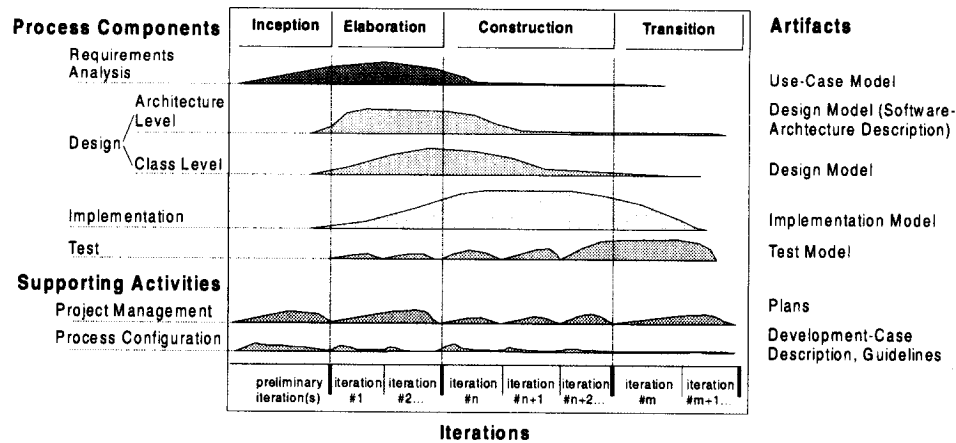
The *Rational Objectory Life Cycle* is a life cycle model proposed by Booch, Rumbaugh and Jacobsen [Rational, 1996]. Objectory is an iterative life cycle model similar to the Boehm spiral model. Each iteration is characterized by the identification and mitigation of risks.



**FIGURE 34.** Sharktooth Model with 2 Prototype demonstrations and 1 review (UML activity diagram, levels of abstraction are represented with swimlanes). Small teeth reaching the management trajectory are internal reviews involving a prototype demo by the developers for the project manager. Large teeth are prototype demonstrations to the client.

The highest risks are addressed first. Iterations are grouped into four phases: inception, elaboration, construction, and transition (see Figure 35). The inception phase corresponds to the IEEE 1074 *Concept Exploration* activity. During this phase, a need or an idea is defined and its feasibility is evaluated. The elaboration phase corresponds to the project initiation process, during which the project is planned the system is defined and resources are allocated. The construction phase corresponds to the development processes. The transition phase corresponds to the installation and post development processes.

Figure 35 depicts the Objectory life cycle model in terms of amount of effort expended into the phases and their iterations. It emphasizes two aspects of software development that are usually not captured in life cycle models: the concurrency of processes and the staging of



**FIGURE 35.** Rational Objectory Life cycle Model (from [Rational, 1996]). The left most column lists the processes and activities of Objectory while the right most column lists the artifacts produced by these processes. The horizontal axis represents time. The vertical axis represents the amount of resources committed to each process as a function of time.

resources. For example, even though requirements evolve throughout the entire project, most of them are completed by the time the construction phase begins. Managing revisions in the use case models and their consequences in the other project artifacts then becomes critical.

Figure 35 depicts both an activity-centered and an entity-centered view of the Objectory life cycle model. The left most column lists the activities that are conducted during the project while the right most column lists their products. In the Object life cycle model, there is a one to one mapping of activities and products.

Orthogonal to the life cycle activities shown in Figure 35, Objectory introduces three modes of developing requirements: *forward engineering*, during which analysis and design models are established from scratch and code is generated from these models, *reverse engineering*, during which these models are extracted from existing code. *Round-trip engineering*, is a combination of reverse engineering and forward engineering. It allows the developer to switch between these development modes at any time. We discuss these modes in more detail in Chapter 6, *Requirements Elicitation*.

### 3.4. Managing activities and products in PROSE

In this section we describe the process needs for our example project PROSE, select a life cycle model, define the products produced by PROSE activities, and define the communication needs.

#### 3.4.1. Process needs and life cycle model selection

PROSE is a pilot project that focuses on developing a demonstration prototype of a software system requested by a client. A second goal of the project is to expose its participants to state-of-the-art software engineering methods and tools. It has forty to sixty participants, most of which have not worked together before. The developers and writers involved in the project have good technical background in writing programs but no large scale development experience. The application domain is new to the participants. The delivery of the first prototype must occur within four months.

We select a life cycle model consisting of three phases. During the first phase, called pre-development, a preliminary requirements and design of the system is developed for the purpose of allocating resources for the rest of the project. Infrastructure choices are also made during the preliminary phase. Only a subset of the project participants are involved. During the second phase, called development, a vertical slice of the systems is constructed following a waterfall model. At this point, all participants have been assigned to a team and are actively working towards the construction of the system. The goal of the development phase is to validate the architecture of the system and expose the participants to all aspects of the life cycle. During the third phase, called post-development, functionality is added incrementally to the system. The goal of the third phase is to extend the functionality of the system while controlling risks. Functionality is added at all levels of abstractions: the requirements, design, implementation, and test cases are developed incrementally. The rationale for this approach is to deliver a functional system on time as opposed to delivering a complete system late.

Table 8 describes each of these three phases in terms of objectives and IEEE 1074 activities and processes.

Note that this life cycle is not complete, it does not include operation and support activities given that the goal of PROSE is to develop a demonstration prototype.

#### 3.4.2. Models and documents

PROSE produces three kinds of information: *system information* (i.e., what is the system? e.g., use cases, class diagrams), *management information* (i.e., how is it built? e.g., tasks, team

**Table 8 Processes and activities of PROSE**

Phase	Purpose	PROSE activities	Corresponding IEEE 1074 activities
Pre-Development	<ul style="list-style-type: none"> <li>Initiate project</li> </ul>	Project Initiation Concept Exploration Toplevel design	3.1. Project Initiation 4.1 Concept Exploration 4.2 System Allocation
Development	<ul style="list-style-type: none"> <li>Validate architecture</li> <li>Train participants</li> <li>Demonstrate feasibility</li> </ul>	Project Management Requirements Analysis System Design Object Design Implementation Unit Testing System Integration Testing System Testing (alpha test) Configuration Management Lectures, Recitations, & Homeworks	3.2 Project Monitor & Control 5.1 Requirements 5.2 Design 5.2.7 Perform Detailed Design 5.3 Implementation 7.1 Verification & Validation 7.2 Software Configuration Management 7.4 Training
Post-Development	<ul style="list-style-type: none"> <li>Complete functionality</li> <li>Deliver prototype</li> <li>Client acceptance</li> </ul>	Project Management Refinement Install Software Accept Software Field Testing (beta test) Configuration Management Lectures, Recitations, & Homeworks	3.2 Project Monitor & Control 6.3 Maintenance <sup>a</sup> 6.1.5 Install Software 6.1.6 Accept Software 7.1 Verification & Validation 7.2 Software Configuration Management 7.4 Training

a. Maintenance is defined in 1074 as a re-map of the software life cycle, that is, as iterations in the development. In this table, we use this process to denote iterations in the development, at not maintenance in its traditional meaning.

organization), and *rationale* (i.e., why such a system? e.g., design goals, design argumentation, justifications). In PROSE, this information is represented by three kinds of models: task models to represent management information, system models to describe the system (requirements, analysis, design, implementation), and issue models to represent the rationale behind design and management decisions.

Task models are detailed, project specific, process models. They are constructed during project initiation by management and revised throughout the project. The goal of the task models is to convey the current status and future assignments to all participants. Task models include a model of the activities and tasks to be done, the persons to which they are assigned, and the organization of the persons. Task models are represented as UML activity diagrams, organizational charts, PERT charts, or Gantt charts. They are documented in the Project Schedule, Software Project Management Plan, and the Top Level Design document (see Table 9). Task models and documents for PROSE are discussed in Section 4.11.

System models represent the system under construction at different levels of abstraction. The use case model describes what the system should do, from a user's perspective. The analysis model describes the domain entities that the system manipulates. The design model represents the structure of the system in classes and objects. The design model is usually a refinement or an extension of the analysis model. The design model is completed with sequence diagrams which illustrate the sequence of object interactions during use cases. The use case and analysis models are documented in the requirements analysis document that the client signs on. The design model and sequence diagrams are documented in the object design document. System models and their documentation are discussed in Section 7.4 (analysis models) and Section 8.4 (design models).

Issue models represent the rationale behind the system. In the case of most systems, the rationale is lost. Developers focus on the system, argue about different alternatives, have different design goals or based many decisions on prior experience. The system designers most often know the reasoning and the constraints that drove the decisions behind the systems. However, this information cannot be recovered from the system alone. Issue models are used in PROSE to capture this information. The motivation behind capturing the rationale in PROSE is to facilitate communication and make design goals and constraints explicit throughout the project. System goals and constraints are described in the system design document. Issues are described in the open issues document. Issue models for PROSE are described in Section 9.4.

Tables 9, 10, and 11 list all the documents produced by PROSE. In Section 13.4, we describe the issues associated with writing the PROSE documentation.

**Table 9 Management documents of PROSE**

Document	Purpose	Produced by
Problem Statement	Describes the needs, visionary scenarios, current system, target system, requirements and constraints	Concept Exploration
Project Schedule	Describe major milestones.	Project Initiation
Toplevel Design	Description of preliminary System Architecture, Teams, Constraints, Communication Infrastructure	Toplevel design
Software Project Management Plan (SPMP)	Controlling document for managing the project	Project Initiation Project Management
Software Configuration Management Plan (SCMP)	Controlling document for software configuration management activities	Configuration Management

**Table 10 System documents of PROSE**

Document	Purpose	Produced by
Requirements Analysis Document (RAD)	Describes the functional and global requirements of the system as well as 4 models - the use case model, the object model, the functional model and the dynamic model.	Requirements Analysis
User Manual	Describes the use of the system often in form of a tutorial	Requirements Analysis
System Design Document (SDD)	Describes the design goals, trade-offs made between design goals, the high level decomposition of the system, concurrency identification, hardware/software platforms, data management, global resource handling, software control implementation and boundary conditions.	System Design
Object Design Document (ODD)	Describes the system in terms of refined object models, in particular the chosen data structures and algorithms as well as full signatures for all public methods. this document results in the detailed specification of each class used by the programmers during the implementation phase.	Object Design
Test Manual	Describes testing strategy, the unit and system tests performed on the system along with expected and actual results.	Testing
Administrator Manual	Describes the administrative procedures to install, operate and bring down the system. Also contains a list of error codes, failure and termination conditions.	Install the Software

**Table 11 Rationale documents of PROSE**

Document	Purpose	Produced by
Issues Document (ID)	Describes the open issues, possible options, arguments, and their resolution	All processes

### 3.4.3. Communication in IEEE 1074 and in PROSE

Documents represent structured means of communication. For example, during the development phase of PROSE, the requirements analysis document is used as a basis for developing the design of the system. When life cycle activities overlap as in the refinement phase of PROSE, this mode of communication, although necessary, is not sufficient. Project

participants need to be able to share new ideas, raise issues, report problems, and brainstorm. These modes of communication are not supported well by documents and need to be supported otherwise. In many small to mid size projects, these modes of communication are supported by face-to-face meetings and hallway conversations. However, project with 40-60 participants such as in PROSE other mechanisms to support a communication infrastructure are needed. Examples of such mechanisms are electronic mail, web servers, meeting procedures, and issue databases.

In IEEE 1074, the design and implementation of such a communication infrastructure is done across several activities such as 3.1.5 Establish Project Environment, 3.2.5 Manage the Project, 3.2.7 Implement Problem Reporting Method, and 7.2.6 Perform Status Accounting. Communication pervades a number of activities and is not treated as an cross-development process in the IEEE 1074 standard. However, in this book we address communication issues and activities as a cross-development process. We discuss this process in Chapter 5, *Project Communication*, and more specifically for the PROSE project in Section 5.4.

### 3.5. Exercises

1. Adapt Figures 22, 23, and 24 to produce an integrated UML class diagram representing the activities and workproducts of PROSE.
2. Select one of the rounds in Boehm's spiral model and draw a UML activity diagram for it.
3. Draw a UML activity diagram describing the dependency between activities for a life cycle in which requirements, design, implementation, test, and maintenance occur concurrently. (This is called an evolutionary life cycle).
4. Describe how Testing activities can be initiated well before Implementation activities. Explain why this is desirable.
5. Assume you are part of the IEEE committee that will revise the IEEE 1074 standard. You have been assigned the task of modeling communication as an cross-development process. List the activities that would belong to this process.

### 3.6. References

[Boehm, 1987] B. Boehm, "A Spiral Model of Software Development and Enhancement," *Software Engineering Project Management*, pp. 128-42, 1987.

[Humphrey, 1989] W. Humphrey, *Managing the Software Process*, Addison-Wesley, Reading, MA, 1989.

[IEEE 1074, 1995] *IEEE Standard for Developing Software Life Cycle Processes*, IEEE Computer Society, New York, NY, 1995.



[Jensen & Tonies, 1979] R.W. Jensen & C.C. Tonies, *Software Engineering*, Prentice Hall, Englewood Cliffs, N.J., 1979.

[Rational, 1996] *Rational Rose: A Rational Approach to Software Development Using Rational Rose*, Rational Software Corporation, Santa Clara, CA, 1996.

[Rowen, 1990] R. B. Rowen, "Software Project Management Under Incomplete and Ambiguous Specifications," *IEEE Transactions on Engineering Management*, vol. 37, no. 1, 1990.

[Royse, 1970] W. W. Royse, "Managing the Development of Large Software Systems", in *Tutorial: Software Engineering Project Management*, pp. 118-27, Computer Society of the IEEE, Washington, D. C., 1970.

