Chapter 2

Software Engineering Processes

In order for software to be consistently well engineered, its development must be conducted in an orderly process. It is sometimes possible for a small software product to be developed without a well-defined process. However, for a software project of any substantial size, involving more than a few people, a good process is essential. The process can be viewed as a road map by which the project participants understand where they are going and how they are going to get there.

There is general agreement among software engineers on the major steps of a software process. Figure 1 is a graphical depiction of these steps. As discussed in Chapter 1, the first three steps in the process diagram coincide with the basic steps of the problem solving process, as shown in Table 4. The fourth step in the process is the post-development phase, where the product is deployed to its users, maintained as necessary, and enhanced to meet evolving requirements.

The first two steps of the process are often referred to, respectively, as the "what and how" of software development. The "**Analyze and Specify**" step defines *what* the problem is to be solved; the "**Design and Implement**" step entails *how* the problem is solved.

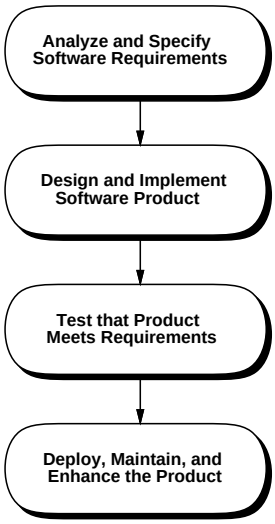


Figure 1: Diagram of general software process steps.

Problem-Solving Phase	Software Process Step
Define the Problem	Analyze and Specify Software Requirements
Solve the Problem	Design and Implement Software Product
Verify the Solution	Test that Product Meets Requirements

Table 4: Correspondence between problem-solving and software processes.

While these steps are common in most definitions of software process, there are wide variations in how process details are defined. The variations stem from the kind of software being developed and the people doing the development. For example, the process for developing a well-understood business application with a highly experienced team can be quite different from the process of developing an experimental artificial intelligence program with a group of academic researchers.

Among authors who write about software engineering processes, there is a good deal of variation in process details. There is variation in terminology, how processes are structured, and the emphasis placed on different aspects of the process. This chapter will define key process terminology and present a specific process that is generally applicable to a range of end-user software. The chapter will also discuss alternative approaches to defining software engineering processes.

Independent of technical details, there are general quality criteria that apply to any good process. These criteria include the following:

1. The process is suited to the people involved in a project and the type of software being developed.
2. All project participants clearly understand the process, or at minimum the part of the process in which they are directly involved.
3. If possible, the process is defined based on the experience of engineers who have participated in successful projects in the past, in an application domain similar to the project at hand.
4. The process is subject to regular evaluation, so that adjustments can be made as necessary during a project, and so the process can be improved for future projects.

As presented in this chapter, with neat graphs and tables, the software development process is intended to appear quite orderly. In actual practice, the process can get messy. Developing software often involves people of diverse backgrounds, varying skills, and differing viewpoints on the product to be developed. Added to this are the facts that software projects can take a long time to complete and cost a lot of money. Given these facts, software development can be quite challenging, and at times trying for those doing it.

Having a well-defined software process can help participants meet the challenges and minimize the trying times. However, any software process must be conducted by people who are willing and able to work effectively with one another. Effective human communication is absolutely essential to any software development project, whatever specific technical process is employed.

2.1. General Concepts of Software Processes

Before defining the process followed in the book, some general process concepts are introduced. These concepts will be useful in understanding the definition, as well as in the discussion of different approaches to defining software processes.

2.1.1. Process Terminology

The following terminology will be used in the presentation and discussion of this chapter:

- ***software process:*** a hierarchical collection of *process steps*; hierarchical means that a process step can in turn have *sub-steps*
- ***process step:*** one of the activities of a software process, for example "**Analyze and Specify Software Requirements**" is the first step in Figure 1 ; for clarity and consistency of definition, process steps are named with verbs or verb phrases
- ***software artifact:*** a software work product produced by a process step; for example, a requirements specification document is an artifact produced by the "**Analyze and Specify**" step; for clarity and consistency, process artifacts are named with nouns or noun phrases
- ***ordered step:*** a process step that is performed in a particular order in relation to other steps; the steps shown in Figure 1 are ordered, as indicated by the arrows in the diagram
- ***pervasive step:*** a process step that is performed continuously or at regularly-scheduled intervals throughout the ordered process; for example, process steps to perform project management tasks are pervasive, since management is a continuous ongoing activity
- ***process enactment:*** the activity of performing a process; most process steps are enacted by people, but some can be automated and enacted by a software development tool
- ***step precondition:*** a condition that must be true before a process step is enacted; for example, a precondition for the "**Design and Implement**" step could be that the requirements specification is signed off by the customer
- ***step postcondition:*** a condition that is true after a process step is enacted; for example, a postcondition for the "**Design and Implement**" step is that the implementation is complete and ready to be tested for final delivery.

In addition to these specific terms, there is certain general terminology that is used quite commonly in software engineering textbooks and literature. In particular, the terms "analyze", "specify", "design", and "implement" appear nearly universally. While the use of these terms is widespread, their definitions are not always the same. In this book, these terms are given specific definitions in the context of the process that is defined later in this chapter. This book's definitions here are consistent with mainstream usage, however the reader should be aware that specific definitions of these terms can vary among authors.

2.1.2. Process Structure

There are a variety of ways to depict a process. A typical graphical depiction uses a diagram with boxes and arrows, as shown in Figure 1. In this style of diagram, a process step is shown as a rounded box, and the order of steps is depicted with the arrowed lines. Process sub-steps are typically shown with a box expansion notation. For example, Figure 2 shows the expansion of the "**Analyze and Specify**" step. The activities of the first sub-step include general aspects of requirements analysis, such as defining the overall problem, identifying personnel, and studying related products. The second sub-step defines functional requirements for the way the software actually works, i.e., what functions it performs. The last sub-step defines non-functional requirements, such as how much the product will cost to develop and how long it will take to develop. This expansion is an over-simplification for now, since there are more than three sub-steps in "**Analyze and Specify**". A complete process expansion is coming up a bit later in this chapter.

A more compact process notation uses mostly text, with indentation and small icons to depict sub-step expansion. Figure 3 shows a textual version of the general process, with the first step partially expanded, and other steps unexpanded. Right-pointing arrowheads depict an unexpanded process step. Down-

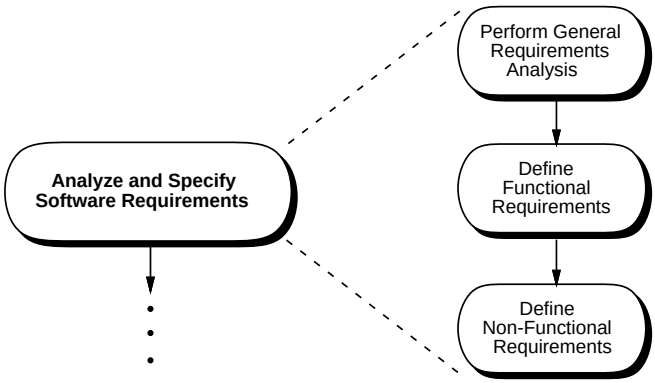


Figure 2: Expansion of the ‘‘Analyze and Specify’’ Step.

- **Analyze and Specify Software Requirements**
  - Perform General Requirements Analysis
    - State Problem to be Solved
    - Identify People Involved
    - Analyze Operational Setting
    - Analyze Impacts
      - Identify Positive Impacts
      - Identify Negative Impacts
    - Analyze Related Systems
    - Analyze Feasibility
  - Define Functional Requirements
  - Define Non-Functional Requirements
- **Design and Implement Software Product**
- **Test that Product Meets Requirements**
- **Deploy, Maintain, and Enhance the Product**

Figure 3: Textual process depiction.

pointing arrowheads depict a process step with its sub-steps expanded immediately below. A round bullet depicts a process step that has no sub-steps.

Depending on the context, one or the other form of process depiction can be useful. When the emphasis is on the flow of the process, the graphical depiction can be most useful. To show complete process details, the textual depiction is generally more appropriate.

An important property of the textual depiction is that it can be considered unordered in terms of process step enactment. In the graphical process depiction, the directed lines connote a specific ordering of steps and sub-steps. The textual version can be considered more abstract, in that the top-to-bottom order of steps does not necessarily depict the specific order in which steps are enacted.

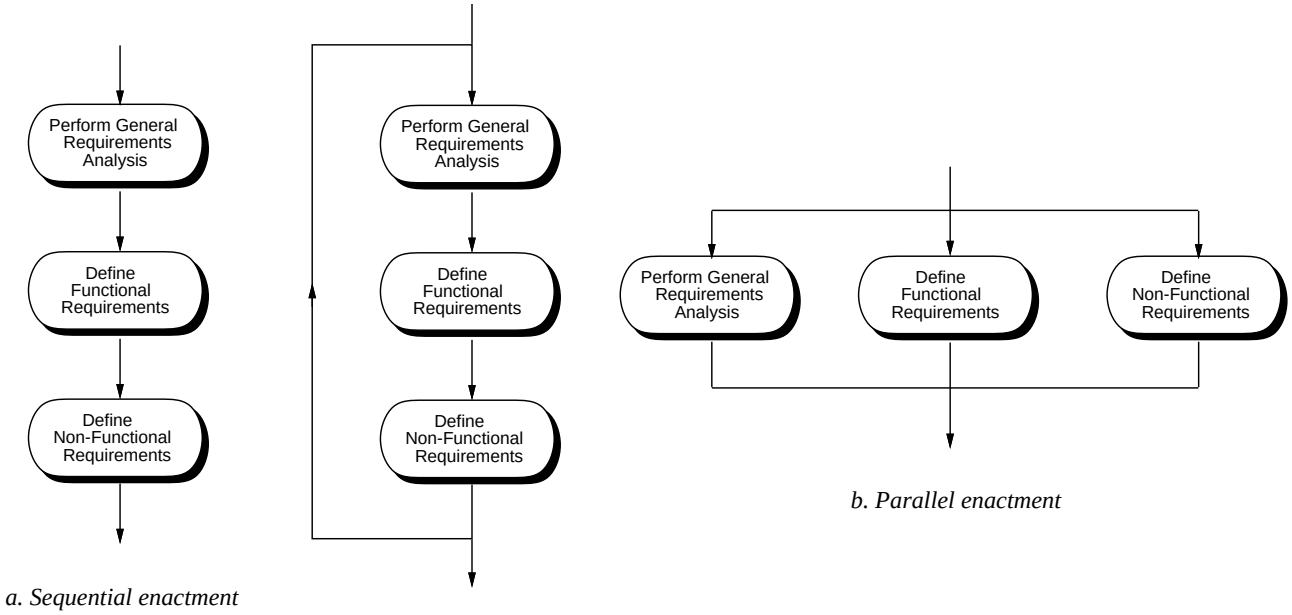
Given its abstractness, the textual depiction of a process can be considered the *canonical form*. Canonical form is a mathematical term meaning the standard or most basic form of something, for which other forms can exist. In the case of a software process, the canonical process form is the one most typically followed. The process can vary from its canonical form in terms of the order in which the steps are followed, and the number of times steps may be repeated.

Consider the three major sub-steps of under **Analyze and Specify** in Figure 3. The normal order of these steps is as listed in the figure. This means that "Perform General Requirements Analysis", is normally performed before "Define Functional Requirements" and "Define Non-Functional Requirements". However in some cases, it may be appropriate to analyze the non-functional requirements before the other steps, or to iterate through all three of the steps in several passes. The important point is that in abstracting out a particular enactment order, the textual process depiction allows the basic structure of the process to be separated from the order of enactment.

2.1.3. Styles of Process Enactment

Once the steps of a software process are defined, they can be enacted in different ways. The three general forms of ordered enactment are *sequential*, *iterative*, and *parallel*. These are illustrated in Figure 4 for the three sub-steps of the **Analyze and Specify** step.

Sequential enactment means that the steps are performed one after the other in a strictly sequential order. A preceding step must be completed before the following step begins. For the three steps in Figure a, this means that the general analysis is completed first, followed by functional requirements, followed by non-functional requirements.



a. Sequential enactment

b. Iterative enactment

b. Parallel enactment

Figure 4: Three styles of enactment.

Iterative enactment follows an underlying sequential order, but allows a step to be only partially completed before the following step begins. Then at the end of a sequence, the steps can be re-enacted to complete some additional work. When each step is fully completed, the entire sequence is done. In Figure b, some initial work on general analysis can be completed, enough to start the function requirements analysis. After some functional requirements are done, work on the non-functional requirements can begin. Then the three steps are repeated until each is complete.

Parallel enactment allows two ore more steps to be performed at the same time, independent of one another. When the work of each step is completed, the process moves on to the subsequent steps.

Which of these enactment styles to use is determined by the mutual dependencies among the steps. For some projects, the determination may be made that a complete understanding of the general requirements is necessary before the functional and non-functional requirements begin. In this case, a strictly sequential order is followed. In other projects, it may be determined that general requirements need only be partially understood initially, in which case an in iterative order is appropriate.

In this particular example that deals with analysis, a purely parallel order is probably not appropriate, since at least some understanding of the general requirements is necessary before functional and non-functional requirements are analyzed. Given this, a hybrid process order can be employed, such as shown in Figure 5. In this hybrid style of enactment, a first pass at general analysis is performed. Then the functional and non-functional analysis proceed in parallel. The process then iterates back to refine the general requirements and then proceed with further functional and non-functional refinements.

The three styles of process enactment discussed so far apply to process steps that are performed in some order relative to one another. A fourth kind of enactment is *pervasive*. A pervasive process step is performed continuously throughout the entire process, or at regularly scheduled points in time. A good example of pervasive process steps are those related to project management. A well managed project will have regularly-scheduled meetings that occur on specific scheduled dates, independent of what specific ordered step developers may be conducting. The steps of the process dealing with project supervision occur essentially continuously, as the supervisors oversee developer's work, track progress, and ensure

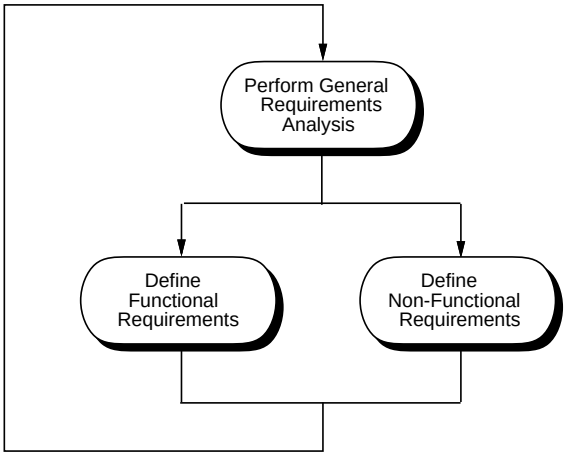


Figure 5: Hybrid process enactment.

that the process is on schedule.

Testing is another part of the software process that can be considered to be pervasive. In some traditional models of software process, testing is an ordered step that comes near the end, after the implementation is complete. The process used in this book considers testing to be a pervasive step that is conducted at regularly schedule intervals, throughout all phases of development.

The people who make the determination of a which style of enactment to use are those who define the process in the first place. Process definers are generally senior technical and management staff of the development team. These are the people who understand the type of software to be developed and the capabilities of the staff who will develop it. The remaining sections of this chapter contain further discussion on the rationale for choosing different styles of process enactment, as well as different overall process structures.

2.2. Defining a Software Process

This book presents and follows a specific software process. The purpose of presenting a particular process is three-fold:

a. to define a process that is useful for a broad class of end-user software, including the example software system presented in the book

b. to provide an organizational framework for presenting technical subject matter

c. to give a concrete example of process definition, that can be used for guidance in defining other software processes

Defining a software process entails the following major tasks: defining the process steps, defining process enactment, and defining the artifacts that the steps produce. Process steps and their enactment are defined here in Chapter 2. The structure of software artifacts is presented in Chapter 3.

An important point to make at the outset is that this is "a" software process, not "the" process. There is in fact no single process that is universally applicable to all software. The process employed in this book is useful for a reasonably wide range of end-user products. However, this process, as any other, must be adapted to suit the needs of a particular development team working on a particular project. A good way to regard the process is as a representative example of process definition. Further discussion of process adaptation appears later in the chapter.

One of the most important things to say about software process is "use one that works". This means that technical details of a process and its methodologies are often less important than how well the process suits the project at hand. Above all, the process should be one that everyone thoroughly understands and can be productive using. There is no sense having management dictate a process from on high that the customers and technical staff cannot live with. The management and technical leaders who define a software process must understand well the people who will use it, and consult with them as necessary before, during, and after the establishment of a process. In order for all this to happen, the process must be clearly defined, which is what this chapter is about.

The top-level steps of the book’s process are shown in Figure 6. These steps are a refinement of the general software process presented at the beginning of the chapter in Figure 1. The refined process has the following enhancements compared to the more general one:

- the "Analyze and Specify" step has been broken down into two separate steps;
- similarly, the "Design and Implement" step has been broken into two separate steps;

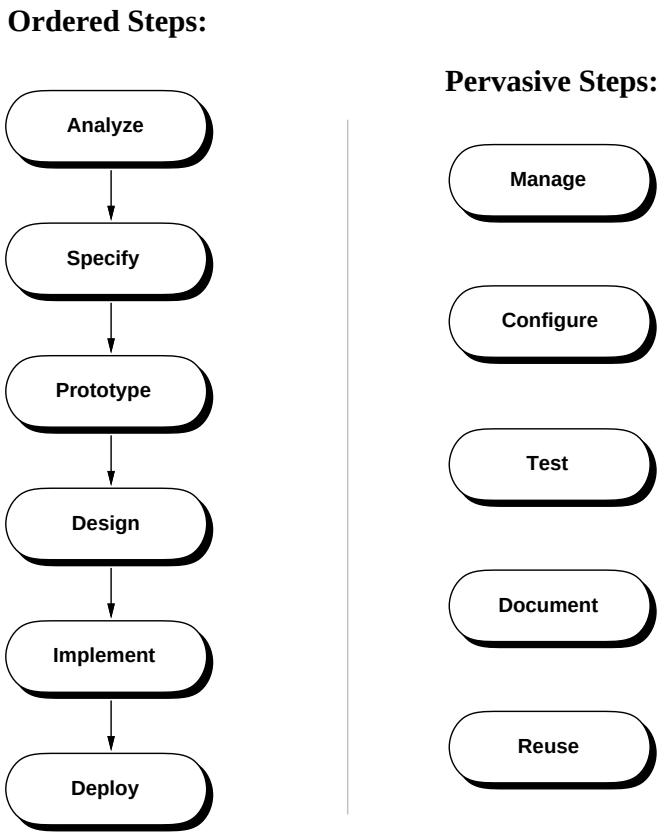


Figure 6: Top-level steps of the process used in the book.

- step names have been shortened to single words for convenient reference;
- prototyping and deployment steps have been added, details of which are discussed shortly;
- testing has been made a pervasive step of instead an ordered step following implementation; this signifies that testing will be carried out at regularly scheduled points throughout the process, not just after the implementation is completed;
- additional pervasive steps have been added for the process activities that manage the software project, configure software artifacts, document the artifacts, and reuse existing artifacts.

From a problem solving perspective, the **Analyze** and **Specify** steps taken together constitute the problem definition phase; the **Design** and **Implement** steps together comprise the problem solution phase. The new **Prototype** step is a "pre-solution", where the developers rapidly produce a version of the product with reduced functionality. The purpose of the prototype is to investigate key product features before all of the details are finished. The **Deploy** step elevates the process from one of plain problem solving to one that delivers a working product to the end users, once the implementation is completed.

The type of software for which the book's process is specifically suited can be characterized as medium-scale information processing with a substantial end-user interface. This category of software covers a significant percentage of commercially available and public domain software that people use. The major characteristics of this type of software are the following:

- a substantial end-user interface, with a reasonably wide range of interface elements; the interface is typically a GUI (graphical user interface)
- information processing functionality that requires the following development techniques to be employed:
  - *o* advanced techniques for data modeling and data design, including interface to external and remote databases
  - *o* advanced techniques for functional modeling and functional design, including distributed processing, event-based processing, exception handling
- a sufficiently large size and scope to require the following process activities:
  - *o* development by multi-person teams
  - *o* the use of techniques to develop non-trivial requirements specification artifacts, including large electronic documents and formal requirements models
  - *o* the use of non-trivial design and implementation techniques, including use of multiple design patterns
  - *o* the use of non-trivial testing techniques
  - *o* the use of non-trivial project management, configuration control, and documentation practices

The process is suitable for the development of software using general techniques of Computer Science. The process is not targeted to software that requires sophisticated specialized techniques, such as artificial intelligence or computer graphics. When knowledge in such fields is necessary, suitable experts need to be added to the development staff.

The process is not entirely suited to systems software, embedded software, highly experimental software, or small-scale software. In the case of systems and embedded software, aspects of the process that focus on human interface requirements are largely or wholly irrelevant. As explained in the introduction, systems and embedded software have little or no requirements for human-computer interaction. There are also technical details of systems and embedded software that this process does not explicitly focus upon. These include steps to analyze operating system and computer hardware requirements that systems and embedded software must meet.

Highly experimental software is characterized by an incomplete understanding of what the software is going to do before it is built. Given this characterization, it is difficult or impossible to have a full set of requirements before implementation of experimental software begins. The process of developing experimental software can be thought of as turning the ordered process in Figure 6 on its head. The experimental process starts with an implementation, which entails writing pieces of program to exhibit some sort of experimental behavior. When part of a working implementation is completed, the developers examine the experimental behavior to see what requirements can emerge, so that the experimental behavior can be refined and expanded. This iterative process continues until the developers are satisfied with the program's behavior as implemented.

Very often, an experimental program is poorly designed, in terms of design standards that software engineers typically consider acceptable. Poor design can make a program difficult and expensive to maintain. In addition, experimental programs are often inefficient in terms of execution speed, since little consideration was given to engineering techniques that produce efficient programs. Given the deficiencies of experimental software, an experimental development process can be followed by a traditional ordered process, if the developers believe that the experimental program forms a suitable basis for a production-quality product. The idea is that the experimental development leads to better understanding of product requirements in an experimental domain. This understanding can then be applied in a traditional development

process, where the requirements are more fully analyzed, a maintainable design is developed, and an efficient implementation produced.

The other type of software to which the book's process is not well suited is small-scale or medium-scale software with the following characteristics:

- development is conducted by one or a few people
- the roles of user, domain expert, analyst, and implementor are filled by the same person or a small number of persons

The development of computer game software can be a good example of this category. For this type of software, the developers are very often avid users themselves. They fully understand the application domain, and are able to transfer requirements ideas directly from their own imagination to a working program. For this type of development, the traditional process covered in this book may well be overkill.

Despite the unique characteristics of different types of software, there are certain aspects of the book's process that are nearly universally applicable. For example, the use of design patterns and the definition of program API are good practices for almost any type of software, except for the most highly experimental. As later chapters of the book cover the process in detail, the issues of process applicability and adaptability will be discussed further.

As noted earlier, software engineers must always strive for a process that is well-suited to their development team and software product. Process definers must continually adapt what they have learned in general about processes to their specific projects at hand. For software projects that are similar to the book's example, adapting the book's process may only be a matter of changing a few details. For other projects, adapting the process may involve major changes, such as adding or deleting steps, or changing the order of the steps.

The way software process is presented and employed in this book is idealized. The presentation can be likened to the way a mathematician presents a complicated proof. Often, the process of conducting the proof is quite messy, with ideas coming from all directions. When the proof is finally published, the author lays things out in a nice neat order, so it can be clearly understood. In a similar manner, the author of this book has laid the software process out in a nice neat order, again for the purpose of clearly understanding it.

Software engineers must be keenly aware that applying a software process in actual practice can indeed get messy. For this reason, those who oversee the project need to be flexible, and prepared to make adjustments to the process as a project is underway. Fine-tuning adjustments are almost always necessary, in response to normal occurrences like scheduling or staffing changes. Any major changes to a process midstream in a project must be more carefully considered, and the management staff must use good judgment when making such changes. Never the less, all project participants must be be prepared to change and adapt their process during the course of a project.

The next two sections of this chapter present an overview of the book's software process, presenting all of its steps but without delving into details. Chapter 3 presents an overview of the artifacts produced by the process. Chapters 4 and beyond then focus on process and artifact details in the context of the technical discussion related all of the process steps. Chapter 25 includes coverage of process evaluation and improvement, as well as details that further formalize the process. In summary, the process definition in this chapter presents the "big picture", with further process details appearing throughout the book.

2.3. Ordered Process Steps

Figure 7 is a one-level expansion the ordered process steps. The ordered steps can be viewed as a process of successive refinement, from an initial idea through to a deployable software product. In this sense, the process is based on a divide and conquer strategy, where the focus of each step is a particular aspect of the overall development effort. At each step, the developers have the responsibility to focus on what is important at that level of refinement. They also have the freedom to ignore or give only limited consideration to what is important at other levels of refinement. Table 5 summarizes the responsibilities and freedoms for each top-level step of the ordered process.

The focus of the **Analyze** step is on user requirements. The needs of the user are the primary concern at this level. Concern with details of program design and implementation should be limited to what is feasible to implement. For projects that are on a particularly tight budget or time line, requirements analysts may need to focus more on implementation feasibility. Also, it may be difficult to estimate implementation feasibility if the analysis team is inexperienced in the type of software being built or in the application domain. In such cases, a more iterative development approach can be useful, as is discussed a bit later in this chapter.

The focus of the **Specify** step is on building a "real-world" software model. Real-world in this context means that the model defines the parts of the software that are directly relevant to the end user, without program implementation details. The distinction between a real-world model and program

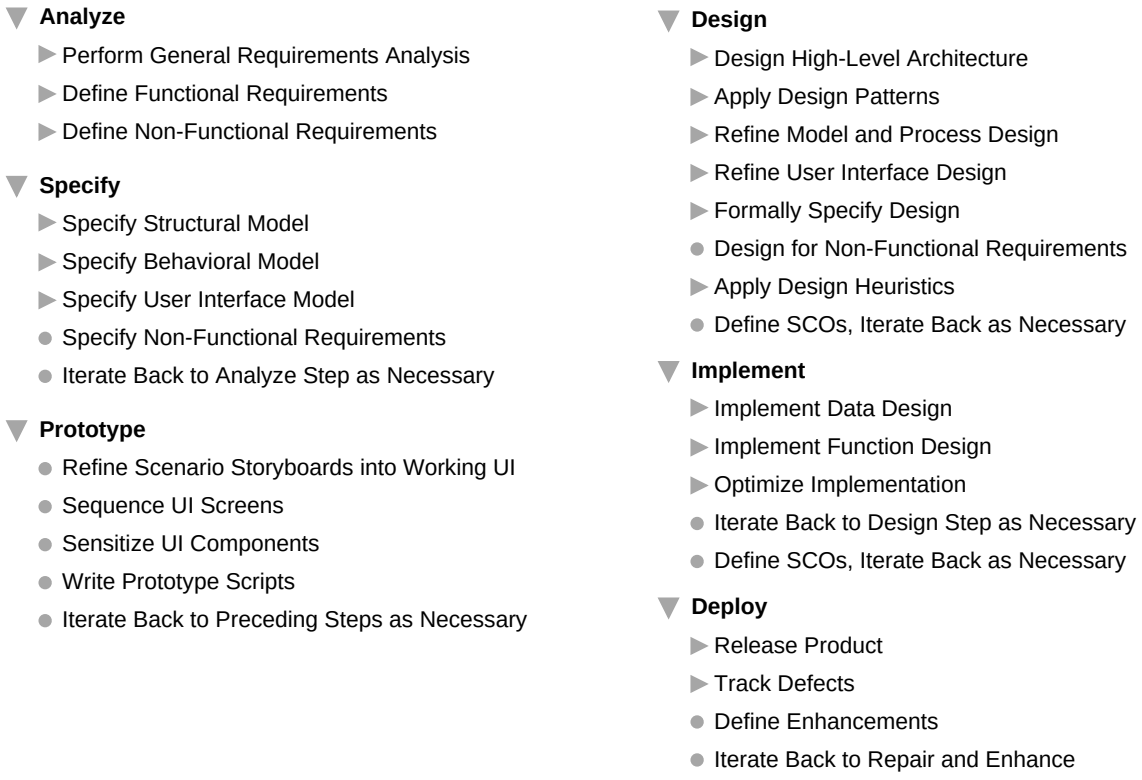


Figure 7: Ordered process steps expanded one level.

Step	Responsibilities	Freedoms
Analyze	Understand the human users and their needs; define the human-computer-interface (HCI).	Ignore program design and implementation details as much as possible.
Specify	Define a real-world software model; specify the behavior of all user-level operations and user-visible objects.	Ignore concrete implementation details; ignore programming language details.
Prototype	Rapidly develop the prototype.	Ignore time-consuming software design and implementation methods; ignore program efficiency.
Design	Define the architectural organization of the program; define the application programmer interface (API); work with the analysis team to address problems in the requirements or specification.	Assume that user-level requirements have been properly analyzed and specified, such that there will be a small number requirements problems; ignore low-level details of program implementation.
Implement	Implement the design as an efficient program; work with the analysis team on problems in the requirements or specification; work with the design team on problems in the design.	Assume that the previous steps have been carried out properly, such that there will be a small number of higher-level problems that need to be addressed.
Deploy	Install and configure the program for use; report problems to the maintenance staff.	As an end user, ignore internal details of the program and how it was developed; as both maintainer and user, assume that the developers have built a quality product, such that there will be few problems that need to be addressed.

Table 5: Responsibilities and freedoms of the ordered process steps.

implementation can be a subtle one. Concrete examples and discussion of this distinction appear in the later chapters that cover software modeling.

The focus of the **Prototype** step is building a partially operational program as rapidly as possible. The purpose of the prototype is to help solidify everyone's understanding of the requirements. For the users, the prototype provides a concrete view that can help them focus on what the software will do. For the developers, the prototype helps them explore concrete ideas for functionality and human-computer interface. In building the prototype, the developers need to be free to employ whatever techniques support very rapid results. This generally means ignoring important but time-consuming steps of design and implementation that must be followed when building the full-scale, production-quality product.

The focus of the **Design** step is the overall architecture of the program, based on the results of the previous steps of the process. A high-level architecture defines large-grain program units and the

interconnection between the units. A lower-level architecture defines further details, down to the level of the application program interface (API). The designers do not to focus on lower-level implementation details, such as concrete data structuring and the procedural implementation of program functions. The later chapters of the book on design discuss the different levels of the design process, the specific definition of an API, and what constitutes design versus implementation detail.

The focus of the **Implement** step is the algorithmic and data detail of an efficient program. The implementors assume that previous steps have been conducted properly, so there will be a small number or problems that need to be addressed at the previous levels while the implementation is under way. In terms of the original idea of a problem-solving process, the implementors are free to assume that the problem has been well defined before they implement its solution.

The focus of the **Deploy** step is to put the developed product to use. This entails distribution, installation and, as necessary, maintenance. The maintenance may be carried out by a separate post-develop team, by the original developers, or by some combination of these. Users and maintainers alike should have the freedom to assume that the developers have built a quality product, that will work correctly and meet the users' needs. Some will say that users have more than the freedom to assume quality, but the *right* to assume it, particularly when they pay for a software product. The issues of societal rights and responsibilities related to software are addressed in a later chapter on software engineering ethics and law.

Throughout the software process, there can be a delicate balance between the freedoms and responsibilities of the different process steps. Questions can arise in particular about how free the developers are to employ a purely divide-and-conquer subdivision of efforts. For example, how thoroughly do the analysts need to understand implementation issues in order to specify a product that is feasible to implement? How well can the analysts define the HCI when they do not fully understand the difficulties of HCI implementation? Such questions will be addressed continuously in the upcoming chapters of the book, as the details of the process steps are further explored.

The preceding questions about software process are much like questions that arise in other engineering efforts. For example, building contractors regularly question the ability of architects to design buildings that can be constructed in an efficient and cost-effective manner. Civil engineers can question the architect's ability to design a building that will stand up to external forces of nature. For their part, the architects want the engineers and contractors to appreciate the architectural aesthetic of a building, even when that aesthetic may be difficult to implement.

When people confront difficulties in other engineering efforts, they do not abandon an overall divide-and-conquer strategy. Rather, they recognize that the process must take into consideration the interaction between the different development steps to ensure that the final product is successfully built. Software engineers are by no means alone in having to deal with the intricacies of a workable development process.

2.3.1. Analyze

Figure 8 shows a full expansion of the **Analyze** step. This step starts by performing a general analysis of user requirements. The initial sub-step is to interview all participating stakeholders. After the initial interviews, communication with affected stakeholders will be an ongoing activity.

In keeping with the overall problem-solving process, the next sub-step of general analysis is to state the problem to be solved. This results in a succinct presentation of the specific problem(s) to be solved and the needs to be met by the software.

- **Analyze**
  - **Perform General Requirements Analysis**
    - Interview Stakeholders
    - State Problem to be Solved
    - Identify Personnel
    - Analyze Operational Setting
    - Analyze Impacts
      - Identify Positive Impacts
      - Identify Negative Impacts
    - Analyze Related Systems
      - Identify Desirable Features
      - Identify Undesirable Features
      - Identify Missing Features
      - Build Feature Comparison Matrix
    - Analyze Feasibility
      - Survey Projected Users and Customers
      - Perform Customer Demographic Analysis
      - Perform Cost/Benefit/Risk Analysis
      - Perform Prototype Usage Studies
  - **Define Functional Requirements**
    - Interview Users
    - Define User Interface Storyboards
    - Identify Functional Categories and Hierarchy
      - Overview Full User Interface
      - Refine Storyboards into User Interface Components
      - Define User Interface Interaction Map
    - Define Requirements Scenarios
      - Describe User Action
      - Describe System Response
      - Refine Input Dialogs
        - Show Representative Input Values
        - Describe Inputs Fully
        - Further Refine Inputs as Necessary
          - Illustrate Alternative Input Values
          - Decompose Non-Atomic Interactions
      - Refine Output Displays
        - Illustrate Alternative Outputs
        - Describe Fully
      - Refine Non-Interactive Behavioral Details
        - Draw Diagrams or Other Appropriate Depictions
        - Describe User Actions
        - Describe Resulting Output or System Behavior
    - Define Non-Scenario Requirements
      - Add Background Information
      - Add Explanatory Information
  - **Define Non-Functional Requirements**
    - Define System-Related Non-Functional Requirements
      - Define Performance Requirements
        - Define Time Requirements
        - Define Space Requirements
      - Define Operational Environment Requirements
        - Define Hardware Platform
        - Define Software Platform
        - Define External Software Interoperability
      - Define Product Standards Requirements
      - Define General System Characteristics
        - Define Reliability Requirements
        - Define Robustness Requirements
        - Define Data Accuracy Requirements
        - Define Correctness Requirements
        - Define Security Requirements
        - Define Privacy Requirements
        - Define Safety Requirements
        - Define Portability Requirements
        - Define Modifiability/Extensibility Requirements
        - Define Simplicity Versus Power Requirements
    - Define Process-Related Non-Functional Requirements
      - Define Development Time
      - Define Development Cost
      - Define Software Life Cycle Constraints
      - Define System Delivery Requirements
        - Define Extent of Deliverables
        - Define Deliverable Formats
      - Define Installation Requirements
        - Define Developer Access to Installation
        - Define Phase-In Procedures
      - Define Process Standards Requirements
      - Define Reporting Requirements
      - Develop Marketing Plan
        - Determine Pricing
        - Determine Target Customer Base
      - Define Contractual and Other Legal Requirements
    - Define Personnel-Related Non-Functional Requirements
      - Define Requirements for Developers
        - Define Credentials Required
        - Define Applicable Licensing, Certification
      - Define Requirements for End Users
        - Define Skill Levels
        - Define Special Accessibility Needs
        - Define Training Requirements

Figure 8: The Analyze step fully expanded.

Following the general problem statement, the analysts identify the personnel involved with the project. These are all stakeholders who will be participating in the project. The specific list of stakeholders can be organized using the categories presented in Section 1.2 of the Introduction.

Analyzing the operational setting entails characterizing the human and computing environment in which the software is to be used. The human environment is the organization who will use custom software, or the general user community who will use an off-the-shelf product. The discussion of the setting includes how operations are conducted in the environment before and after the software is installed. The following questions are addressed:

a. What computer-based support is in use prior to installation of the new system?
b. Does the new system need to interface with existing software or is the existing software to be replaced entirely?

The impact analysis sub-step assesses the impacts of the software within its operational setting. Both positive impacts (e.g., increased productivity, higher product sales) as well as negative impacts (e.g., job displacement, potential negative legal impacts) are addressed.

The analysis of related software requires identification of existing software that provides functionality similar or related to the functionality of the software being proposed. The following issues are addressed:

a. What is good about the related software, i.e., what features does it have that should be included in the system software being proposed.
b. What is bad about the related software, i.e., what features should not be included in the proposed software, or what features should be included but done in a different or better way.
c. What is missing, i.e., what new features should be included in the proposed software that are not found in the related products.

If appropriate, related software can be overviewed in a feature comparison matrix. This is a table that lists all of the features of the related software, for the purposes of side-by-side comparison.

Feasibility analysis addresses issues related to the user community and, if appropriate, the commercial market for which the software is targeted. Some or all of the following sub-steps can be carried out;

a. surveys of projected users and customers, to determine their wants and needs
b. demographic analysis of customers, to determine the potential profitable market for an off-the-shelf product
c. cost/benefit/risk analysis to determine if the a profit can be made in the target market, and if the risks associated with the project are outweighed by the benefits
d. prototype usage studies, where potential customers use a system prototype to test their reactions

Chapter 4 of the book covers the general analysis step in complete detail.

Following general analysis, the next major analysis sub-step is devoted to functional requirements. This is typically the part of the analysis that consumes the most time and energy.

Functional requirements define the specific functions that the software performs, along with the data operated on by the functions. In the process defined here, the primary form for presenting functional requirements is scenarios that depict an operational software system from the perspective of its end users. Included are one or more examples of all system features and an enumeration of all the specific requirements associated with these features.

When formulating the initial ideas for a product, analysts use *storyboards* to work out the way a program will appear to its end users. Storyboarding is a practice borrowed from the movie industry, where a

director sketches out the scenes of a movie before it is filmed. In a similar way, a software analyst sketches out the user interface of a program before it is implemented.

Following the initial storyboarding, an overall functional hierarchy is developed. In concrete terms, this is the top-level user interface to the software. In current practice, the top-level UI is very typically menu-based, but other forms are widely used, such as toolbars and control panels. As the functional hierarchy is developed, the initial sketched storyboards are refined into concrete user interface components, so that the user can view the requirements in explicitly user-centered terms, namely through the interface that the user will employ to communicate with the software.

A UI interaction map may be defined in this sub-step. An interaction map shows a thumbnail view of each interaction screen. The thumbnails are connected with directed arrows that describe the form of user interaction that leads from one screen to the next.

The specific methodology presents scenarios using *action/response sequences*. The action is performed by the user, the response is generated by the software system. The key steps of the scenario process are the following:

a. Describe an action performed by the user, such as selecting a menu item or typing in some data value.

b. Describe the the system response, such as displaying an information window or adding a value to some data collection.

c. When a system response is a request for user input, illustrate a representative set of sample input values, and define new scenarios around subsequent user actions.

d. When a system response is an output display, describe the output precisely and illustrate a representative set of alternative output forms, adding new scenarios for major output alternatives.

To establish a complete definition of all functional requirements, the interaction scenarios are augmented with descriptions of the non-interactive system behavior. Non-interactive behavior is computation performed by the system that generates no immediate interactive response, such as internal computation or communicating with external programs.

To present a complete set of requirements, scenarios are augmented with additional content that is not in the action/response style . This portion of the requirements provides necessary background information and other explanatory details to make the requirements completely clear to all readers.

All system behavior is defined strictly in user-level terms, never in terms of an underlying program implementation. The overriding rule is "If a user sees it , define it", otherwise consider it an implementation detail. What it means for a user to "see" a particular behavior or data value is that the user either sees it explicitly in visual form, or is aware of it based on computation performed by the system. Chapter 5 describes the functional requirements process in full detail, augmented with many concrete examples.

The third sub-step of the **Analyze** is devoted to non-functional requirements. These requirements address aspects of the system other than the specific functions it performs. Aspects include system performance, costs, and such general system characteristics as reliability, security, and portability. The non-functional requirements also address aspects of the system development process and operational personnel.

There are three major categories of non-functional requirements, corresponding to the three sub-steps of the non-functional process:

- ***system-related*** -- these are non-functional requirements related to the software itself, such as performance, operational environment requirements, product standards, and general system characteristics
- ***process-related*** -- these are requirements for the software development process, including how long

it will take, how much it will cost, and other relevant matters

- ***personnel-related*** -- these are requirements related to the people involved with software development and its use

There are a number of details shown Figure 8 that have not been fully enumerated in the preceding overview. Complete details of the non-functional process are covered in Chapter 6.

In comparing the **Analyze** step of the process to the other major steps that follow, it has more details, particularly in the area of non-functional requirements. The reason for this is that the requirements phase defines general project goals and product requirements that are broadly applicable to most types of software. Once these goals are established, they apply to the overall process and product being developed, throughout the subsequent development steps. In effect, the subsequent steps carry forward the goals established in the definition of the requirements.

2.3.2. Specify

Figure 9 shows a full expansion of the **Specify** step. The Specify step of the process involves the development of a formal model of the requirements. The purpose of the model is two-fold:

- it helps the analysts uncover flaws and omissions in the requirements;
- it defines the formal specification that can be used as a contract with implementors.

In the process of this book, the formal model is defined in a language that can be mechanically analyzed. The analyzer checks the model in basically the same way that a compiler checks a program. Namely, it checks the syntax and some aspects of the semantics of the model. This mechanical analyzer helps the human analyst find flaws and omissions in the model.

The idea of the specification forming a contract with the implementors is extremely important when it comes time to verify that a delivered software product meets its requirements. When the requirements are distilled into a formal specification, then the process of testing the implementation against its specification is much more rigorous then when requirements are defined in a less formal form, such as only English and pictures.

The two main sub-steps of **Specify** involve the construction of structural and behavioral software models. The structural model defines the static structure of the software. The behavioral model defines precisely the way the software behaves in terms of the inputs it receives and the outputs it produces.

The structural model is derived initially from the requirements scenarios using some general derivation heuristics. A heuristic is a "rule of thumb" that defines in general terms how to perform some task. The heuristics for model derivation define how model objects, operations, and attributes are derived from the user-centered requirements scenarios. An object is the formal definition of a user-visible piece of data. An operation is the formal definition of an action performed by the software using the objects. In particular, operations take objects as inputs and produces objects as outputs. A model attribute is a general characteristic of the software.

Once model objects, operations, and attributes are derived from scenarios, they are refined further based on the detailed scenario narrative. This part of the process very typically involves a significant amount of iteration with the **Analyze** step, that proceeds in high-level terms like this:

a. Define a requirements scenario.
b. Derive objects, operations, and attributes, leading to the discovery of flaws or omissions in the scenarios.
c. Go back to the requirements to update the scenarios accordingly, by fixing the flaws and adding

- **Specify**
  - **Specify Structural Model**
    - Derive Objects from Requirements Scenarios
      - Derive Input Objects from Data-Entry Dialogs
      - Derive Output Objects from Data Output Displays
      - Derive Other Objects from Narrative Nouns
    - Derive Operations from Scenarios
      - Derive Operations from Menus and Buttons
      - Derive Other Operations from Narrative Verbs
    - Derive Model Attributes from Scenarios
    - Refine Objects
      - Define Component Details to Atomic Level
      - Identify Underlying Collection Objects
      - Define Inheritance from Generic Objects
      - Add Descriptions Based on Narrative
    - Refine Operations
      - Specify Inputs and Outputs
      - Identify Default Inputs
      - Add Descriptions Based on Narrative
    - Refine Attributes
    - Modularize Structural Model
      - Define Module Packaging
      - Specify Module Imports
  - **Specify Behavioral Model**
    - Define Predicative Specification
      - Derive Preconditions and Postconditions
        - Sketch Conditions as Prose Comments
        - Refine Conditions into Formal Logic
      - Refine Object and Operation Definitions as Necessary
      - Define Auxiliary Functions as Necessary
    - Define Inter-Operation Specification
      - Refine Preconditions and Postconditions
      - Define Inter-Operation Dataflow
    - Define Equational Specification
      - Define Object Equations
      - Define Auxiliary Functions as Necessary
    - Define Axiomatic Specification
      - Define Global Variables
      - Define Axioms
      - Define Auxiliary Functions as Necessary
    - Define Attribute Grammar Specification
      - Define Attributes
      - Define Rules
      - Define Attribute Equations
      - Define Auxiliary Functions as Necessary
    - Define Constructive Functional Specification
      - Define Operations Functionally
      - Define Auxiliary Functions as Necessary
  - **Specify User Interface Model**
    - Define User Interface Structure
    - Define User Interface Behavior
  - **Specify Non-Functional Requirements**
  - **Iterate Back to Analyze Step as Necessary**

Figure 9: The Specify step fully expanded.

new scenarios for the omitted functionality.

The final sub-step of structural modeling is the modularization of the model into functionally-related packages of objects and operations. This model packaging is used subsequently in the design step to derive the initial program architecture.

Specifying the behavioral model is where the specification becomes fully formal. In the process shown in Figure 9, the main form of behavioral specification is called *predicative*. A "predicate" is a boolean expression, of the type familiar to all programmers. There are some additional details to the language of predicates used in formal specification, but fundamentally a predicate simply states a condition that must be true or false.

A *predicative* specification defines two types of conditions for program behavior. A *precondition* must be true before an operation executes, and a *postcondition* must be true after an operation completes its execution. Using just these two forms of condition, a formal definition can be specified for most of a program's behavior. The specification can be further refined by enacting one or more additional process steps. These additional steps address the following aspects of specification:

- ***inter-operation behavior*** -- this is the definition of the way in which operations interact with one another; a dataflow diagram can be used to specify the way the outputs of operations feed into the inputs of other operations
- ***equational specification*** -- this defines specific constraints on objects in terms of what the operations are allowed to do with and to the objects
- ***axiomatic specification*** -- this defines formal rules, i.e., axioms, that can be particularly useful in specifying the behavior of distributed and concurrent software
- ***attribute grammar specification*** -- this defines a model in terms a formal language definition that is part of the software; for example, a query language that is part of a database software system can be defined using an attribute grammar
- ***constructive functional specification*** -- this form of specification is useful when certain details of behavior are most easily specified in operational terms; a constructive specification is a form of very high-level program

There are other forms of behavior specification not explicitly cited in this process. These include behavioral specifications based on state machines, stochastic techniques, and temporal logic. These forms of specification are explained and discussed in a later chapter, but not employed in the book.

The third sub-step of **Specify** is devoted to the specification of a user interface model. It is important in software specification and design to separate the details of abstract functionality from concrete user interface. For this reason, the specification of user interface structure and behavior is separated from the specification of the underlying functional model. This separation is reflected significantly in the **Design** step of the process, as will be explained shortly.

The fourth **Specify** step is the definition of those non-functional requirements that can be formally modeled. This includes the specification of such model properties as the size of data objects, and the speed at which operations must execute. Such formal model attributes form a bridge between functional and non-functional requirements. In general, functional requirements can be specified fully formally, whereas non-functional requirements may be specified only partially formally.

The last step of **Specify** shown in Figure 9 is not an actual operative step, but an indication that the **Specify** step is likely to be part of an iterative process involving the **Analyze** step. While the iteration may occur at any point during **Specify**, it is listed at the end as a general indication that iteration is a normal part of the combined **Analyze** and **Specify** phase of the process. Further details of ordered process enactment are discussed in Section 2.5.1.

2.3.3. Prototype

The full expansion of the **Prototype** step appears in Figure 7 since it does not expand beyond one level. As outlined earlier in Chapter 2, prototyping involves the rapid creation of a partially operational program. The prototyping process begins by refining scenario pictures into a working user interface. This entails using a prototyping tool or user-interface builder to create operational interface screens.

To create a very basic form of prototype, the interface screens can be presented in a step-by-step sequence, illustrating a particular set of prototypical interactions. This form of prototype is a "slide show"

of user interaction, that does not allow the user to interact dynamically with the prototype.

To create a more dynamically interactive prototype, the components in the interface screens can be *sensitized*, such that the user may treat them like operating elements of the user interface. Certain prototyping tools allow plain interface pictures to be sensitized. For example, the drawn picture of a button in an interface screen can be sensitized to behave like an actual clickable button. Alternatively, the prototype developer can use an interface building tool, where an operational prototype interface is created to have the same appearance as the screens drawn in the scenarios. In either case, the result is an interface with which end users can directly interact.

To define actual program behavior, the prototype developer writes action *scripts* that are associated with particular interface components. For example, suppose prototype interface contains a button labeled "`Find`" that when pressed displays the result of some search operation. The script for `Find` button can be defined to display an interface screen that shows a prototypical search result, thereby simulating a prototypical behavior. This type of prototype presents completely "canned" behavior. That is, the prototype responds to user interaction by displaying only pre-defined results, without performing any actual computation.

To define a prototype with uncanned behavior, the scripts can extended to perform actual computation. In the case of search example, the script for the `Find` button is written to search some form of prototype data store, and present the results of the actual search. The representation of the prototype data store is some form that can be rapidly developed, without regard for storage efficiency or other data storage requirements that cannot be rapidly implemented.

For some types of software, it may be possible to evolve a prototype into a production product by reusing some or all of the prototype interface and scripting. This is an *evolutionary* style of prototyping. When little or none of a prototype can be reused in the production software, the prototype is considered a *throw-away*. Once a throw-away prototype has served its purpose, that is to clarify the requirements, the prototype is discarded.

Whether the prototype evolves or is discarded, its use in requirements clarification is primary. For this reason, the last step of the **Prototype** process is to iterate back to the **Analyze** and **Specify** steps, so that what is learned from the prototype can be integrated into the requirements and specification.

2.3.4. Design

Figure 10 is a full expansion of the **Design** step. The step starts by deriving the high-level architecture of the program from the abstract model constructed in the **Specify** step. The modularization defined for the structural model is carried forward into the packaging of the program design. This enforces traceability between the abstract specification and the corresponding architectural program design.

The high level architecture of a program is defined in terms of data classes and computational functions. These are derived, respectively, from the objects and operations of the specification. The classes and functions derived directly from the specification constitute the *model* portion of the design. The classes derived from concrete user interface and UI model are the *view* portion of the design. Other properties of the design are derived from attribute definitions in the specification. In addition, the commentary in the specification is used as the basis for design comments.

Once the top-level design elements are derived from the requirements specification, software design patterns are applied. A design pattern is a pre-packaged piece of design, based on experience that has been gained over the years by software engineers. A widely-used design pattern for end-user software is

- **Design**
  - **Design High-Level Architecture**
    - Derive Initial Design from Specification
      - Derive Architectural Packaging from Modules
      - Derive Model Classes from Objects
      - Derive Model Functions from Operations
      - Derive View Classes from UI Pictures and Model
      - Derive Design Properties from Spec Attributes
      - Derive Design Comments from Spec Comments
    - Design Inter-Package Sharing and Communication
  - **Apply Design Patterns**
    - Apply Architectural Design Patterns
    - Apply Model-View-Process
    - Apply Data Design Patterns
    - Apply Control Patterns
    - Apply Communication Patterns
    - Apply Other Appropriate Design Patterns
    - Refine and Customize Applied Patterns
  - **Refine Model and Process Design**
    - Refine Model Package Design
    - Refine Model Class Design
      - Associate Functions with Classes
      - Objectify Function Signatures
      - Define Class Member Visibility
      - Define Class Inheritance and Other Relations
      - Choose Appropriate Data Representations
        - Select Data Representations from Libraries
        - Design Custom Data Representations
    - Design Process Packages and Classes
      - Design Controller Classes
      - Design Adaptor Classes
      - Design Wrapper Classes
      - Design External Data Input/Output
      - Design Other External Data Interfaces
    - Design Control Flow
      - Refine Specification Dataflow If Appropriate
      - Design Functional Control Flow
      - Design Event Handling
      - Design Exception Handling
  - **Refine User Interface Design**
    - Refine View Package Design
    - Choose User Interface Library Components
    - Design User Interface Layouts
    - Add View-Supporting Functions to Model Classes
    - Apply Observer/Observable Design Pattern
    - Refine Model/View Communication
  - **Design for Non-Functional Requirements**
  - **Formally Specify Design**
    - Fully Identify all Function Inputs and Outputs
    - Refine Derived Preconditions, Postconditions
    - Define Preconditions, Postconditions for New Functions
  - **Apply Design Heuristics**
    - Minimize Coupling
    - Maximize Cohesion
    - Apply Other Appropriate Heuristics
    - Employ Appropriate Design Metrics
  - **Define SCOs, Iterate Back as Necessary**

Figure 10: The Design step fully expanded.

Model-View-Process. This pattern organizes the design into three major segments:

- the *Model* is directly traceable to the abstract functionality defined in the requirements model, and is independent of the concrete end-user interface;
- the *View* segment of the design is devoted specifically and solely to the end-user interface
- the *Process* segment defines underlying processing support for the model, in particular processing

that encapsulates platform-dependent aspects of the design.

Other patterns are employed to assist with design of program data, control, and communication.

The derived, pattern-based design produced by the first two steps must be refined into a concrete, object-oriented program design. This is accomplished in the next two design steps. At the high-level design, derived packages must be refined. At the class level, derived functions must be associated with specific model classes. This step is necessary since the operations of the functional specification do not necessarily belong to specific objects. Functions associated with classes become class *methods*, with appropriate adjustment to method signatures based on object-oriented design concepts. Other necessary design refinements are in the areas of class member visibility, inheritance, and the selection of concrete data representations. In a modern program design, data representations are typically selected from reusable program libraries.

Process class design entails determining the underlying processing support that is necessary to produce an efficient program. To encapsulate platform-dependent data processing, process classes are interfaced with model classes via controller, adaptor, and wrapper classes. These model/process interface classes encapsulate aspects of the program that are specific to specific operating systems, hardware platforms, and external data stores.

An important part of model and process refinement is detailed control flow design. This includes the design of inter-class data flow, functional control flow, event handling, and exception handling.

The fourth step of design is devoted to refining the end-user interface. In the current state of the art, user interface design typically relies heavily on libraries of reusable interface classes. The class libraries define commonly-used interface elements and layouts. In a Model-View design, the model classes must be refined to support the view classes, based on the specifics of the user interface. A particularly useful design pattern in this regard is called "Observer/Observable". This pattern defines the way in which multiple view classes can be systematically updated in response to data changes made by the user. Additional work in this design step involves refining the communication between model and view classes. This level of refinement focuses on how input data are sent from view classes into model classes, and how output data are sent from the model to the view.

The fifth design step focuses on system-related non-functional requirements. Non-functional requirements that were formally modeled will already have been incorporated into the design as a result of the initial design derivation step. Also, certain design patterns may be oriented to the design of non-functional requirements, such as security. Any other non-functional requirements that were not modeled in the specification or are not yet incorporated in the design are dealt with in this step. The purpose of this step is therefore to ensure that all system-related non-functional requirements are fully addressed in the design.

Once a detailed program design is established, the design is formally specified. This entails the precise definition of function (i.e., method) input/output signatures, followed by the specification of preconditions and postconditions for all functions. For the model functions derived directly from the specification, the function conditions are derived directly from the preconditions and postconditions defined in the derived-from operations. For other model and process functions, preconditions and postconditions are defined with the same methodology used in the abstract specification model. Namely, preconditions are expressions that must be true before function invocation; postconditions must be true after function executions.

Various design heuristics (i.e., general guidelines) can be applied throughout the process of design. Minimizing coupling among program elements aims to reduce the dependency and communication to only that which is essential Maximizing cohesion means that program elements that are functionally related are

grouped together, without extraneous unrelated elements. Other heuristics can be applied, such as controlling the size of various program units.

During the course of program design, the developer may discover aspects of the requirements specification that need to be modified or enhanced. In such cases, the designer defines a *specification change order* that clearly states the necessary modifications or enhancements. This formalized change order is in keeping with the high-level process decomposition into problem definition and problem solution phases. As discussed earlier in this chapter, the **Analyze** and **Specify** process steps comprise the problem definition phase. The **Design** and **Implement** steps then comprise the problem solution phase. In this software process, as in a traditional problem-solving process, changing the problem definition while the solution is underway requires careful consideration. The specification change order codifies this careful consideration in a precise way.

2.3.5. Implement

Figure 11 shows a full expansion of the **Implement** step. The **Implement** step fills in the operational details of the program by fully refining program data and functions. Implementing the data design requires the selection of fully concrete data structures for the representation of the data in all classes. This may in turn lead to the definition of new inheritance relations, and the design of lower-level classes to support an efficient implementation.

The implementation of the functional design involves the coding of all function bodies. This is the most concrete aspect of software programming. This typically leads to the definition of additional low-level

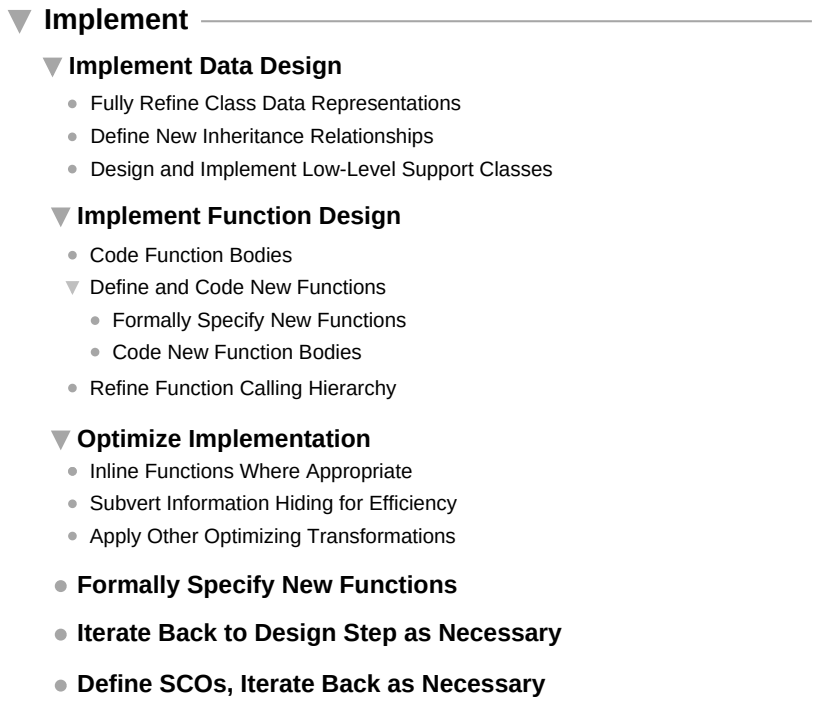


Figure 11: The Implement step fully expanded.

functions and classes, as the implementation details evolve.

As new functions are defined during implementation, they must be formally specified in same manner as during the design process. In general, as the refinement of the implementation leads to the definition of new classes and functions, the implementation process iterates back to design, where the appropriate design steps are applied to the newly-defined classes and functions.

A key aspect of function implementation is the refinement of the function calling hierarchy. As function implementations expand, it may well be necessary to subdivide the functions into additional sub-functions, and refine data implementations accordingly.

Techniques for optimizing the implementation include the use of inline functions, and partial subversion of information hiding where necessary for efficiency. Other optimizing techniques can be applied based on how strong the requirements are for implementation efficiency. Modern compilers can be relied on to perform a variety of optimizing transformation to improve program execution efficiency.

Iteration between design and implementation steps is a normal part of the process. As noted above, the iteration occurs as new classes and functions are defined during implementation. In addition, implementation may reveal incompleteness of flaws in the higher-level design, requiring iteration back to the design.

If the need arises to modify or enhance the requirements specification during implementation, a specification change order is defined. The process then iterates back to an appropriate pre-design step.

The number of specific substeps within `Implement` is relatively smaller than any of the preceding ordered steps, particularly `Design`. The reason for this is that the design substeps can in effect be considered implementation substeps as well. The implementation is the concrete realization of the design. Hence, the general steps of the implementation are applied to each specific element of the program created in the `Design` step.

2.3.6. Deploy

Figure 12 is a full expansion of **Deploy**. Deployment is the post-development phase of the ordered

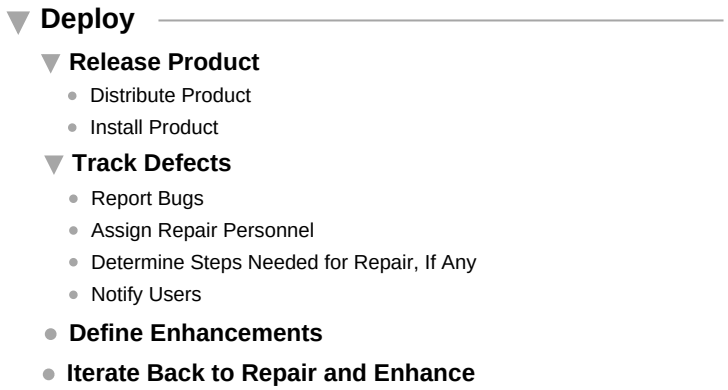


Figure 12: Deploy step fully expanded.

process. It begins with the release of the completed software product. Releasing includes the necessary product distribution and installation. Once the product is put into service, any defects detected by the users are tracked and handled as necessary. Defect tracking entails the user's reporting a perceived bug, assigning someone from the maintenance staff to analyze the problem, and determining the steps to accomplish the repair if repair is needed. As the bug is assigned and handled, users are notified of its repair status.

For many software products, the definition of feature enhancements is a regular part of the post-delivery process. Enhancements may be suggested by users or defined by members of the development staff who's job it is to continue product development in response to evolving user needs.

Both the repair of defects and the development of enhancements are handled in the process by iterating back to the appropriate development step. For example, if a defect is determined to be strictly an implementation problem, the implementation step is invoked to perform the repair. In contrast, a substantial program enhancement may involve iteration all the way back to the beginning of the **Analyze** step, where new requirements are gathered, then specified, designed, and implemented.

2.4. Pervasive Process Steps

Figure 13 is a one-level expansion the pervasive process steps. As introduced earlier, a pervasive step is performed continuously throughout the ordered process, or at regularly scheduled points in time.

Figure 13: Pervasive process steps expanded one level.

In some cases, the work performed in a pervasive step applies generically to all of the ordered steps. For example, many of the tasks performed to manage a software project apply uniformly across all of the ordered steps. In other cases, the work performed in a pervasive step must be tailored individually to the different ordered steps. For example, the testing of an implemented program has specific details that are different than the testing of the requirements.

The particular aspects of pervasive processing that are uniform across ordered steps are the following:

- most aspects of the **Manage** step are generic for all ordered steps;
- almost all aspects of **Configure** apply uniformly to all artifacts;
- testing based on human inspection is generic for all ordered-step artifacts, however other aspects of testing are artifact-specific;
- basic documentation practices are generic for all artifacts, but specific content obviously varies.

2.4.1. Manage

Figure 14 shows a full expansion of the **Manage** step. The management process begins by establishing the project infrastructure. This entails defining the software process, defining the structure of software artifacts, and defining standard operating procedures (SOPs). The SOPs define specific details of conducting the project on a day-to-day basis. SOPs typically contain process details that are specific to single project or development group, where such details are considered too specific or specialized to be part of the overall process.

Project scheduling and resource allocation are key aspects of the management process. Development and deployment timelines are typically defined in terms of *milestones* that specify precisely what needs to be accomplished on what dates. Resource allocation involves the determination of what human and

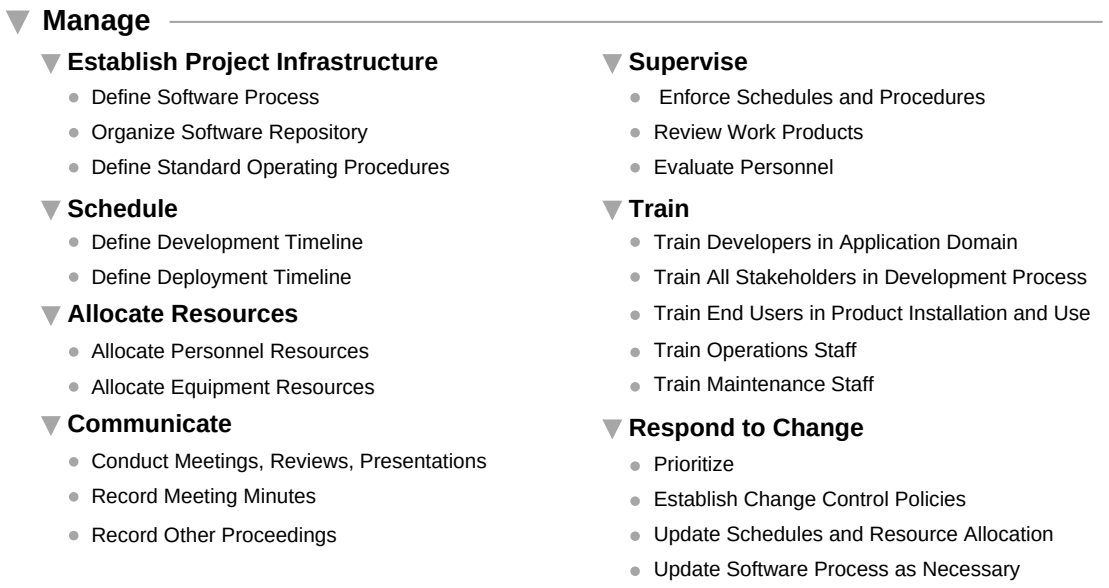


Figure 14: The Manage step fully expanded.

equipment resources are necessary to carry out a project.

Communication is an ongoing activity. It occurs during regularly scheduled meetings, project reviews, and presentations. The proceedings of all such communication sessions must be recorded, to become a permanent part of the software project repository.

The Supervise step involves what is traditionally considered the core of project management. The supervisors enforce schedules, review work products, and evaluate personnel.

Depending on the knowledge and skill levels of project stakeholders, various forms of training may be necessary. For developers unfamiliar with the software application domain, training must take place in this area. All stakeholders need to be trained in the development process; end users and customers in particular need to be familiarized with the requirements analysis process. When a product is delivered, end users may need training, as may the operations and maintenance staff.

Responding to change is a key aspect of any software project. As a project proceeds, development tasks and work products must be prioritized based on their importance and the amount of available time and resources. The most typical form of prioritization is that of requirements, where the users and customers are consulted to determine which requirements are high priority versus those that are of lower priority. This priority information is used to determine which requirements are worthy of continued pursuit and which are not, given time and resource limitations.

In order to manage changing priorities effectively, a change control policy must be established. This policy defines how, when, and where changes are recorded. As changes are addressed, schedules and resource allocations must be updated accordingly. In some cases, it may be necessary to update the software process, if it is not accomplishing the development in an effective and timely manner.

2.4.2. Configure

The full expansion of the **Configure** is shown in Figure 15. The first step of configuration management is to establish the project repository. The repository contains all of the artifacts produced throughout the lifetime of the project. All software artifacts are assigned ownership, in order to be clear who is responsible for which artifacts.

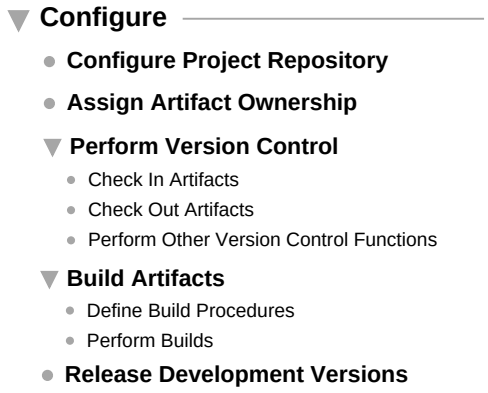


Figure 15: The Configure step fully expanded.

Once the repository is established, in is maintained using standard version control procedures. The check-in and check-out operations are used to commit artifacts to the repository and retrieve the artifacts once committed. A wide variety of other version control functions are performed to manage the configuration and history of the repository.

The build substep applies to all software artifacts for which some form of automated building is performed. The most typical form of build is that performed by a compiler, where executable programs are generated from source code. Other levels of artifacts can have automated building tools. At the **Analyze** level for example, there are tools to build web-browsable requirements from word processor documents, and tools to generate summary reports when requirements documents are particularly lengthy. At the **Specify** level, there may be tools to generate web-browsable views of the formal model, or parts of it. Build tools at the design level include design documentation generators, such as *javadoc*. There are also tools that can generate code templates from design diagrams or drawn pictures of user interfaces.

At well-defined milestones, versions of development artifacts are released to the appropriate stakeholders. The most significant release is that of the finished software product, released to the end users. This product release is an explicit part of the **Deploy** step outlined earlier.

It is worth noting the commonality in the **Configure** step with the **Manage** and **Deploy** steps. In the case of **Manage**, the assignment of artifact ownership and some aspects of version control can be considered managerial tasks. They are included as part of configuration since they relate to specific technical details of artifact ownership and control. The commonality between **Configure** and **Deploy** is in the area of product releases, as noted above. Namely, the release of a finished product to end users is considered to be a part of deployment. This is reasonable, since deployment to an end user is an external release, compared to the internal development releases that are considered to be part of **Configure**.

2.4.3. Test

Figure 16 shows a full expansion of the **Test** step. The first six substeps constitute the major types of testing that are applicable to some or all software artifacts. The repair substep applies generically to determining the cause of test failures and effecting necessary repairs.

Inspection testing is carried out by humans, who thoroughly and systematically inspect software artifacts. To do so, the inspectors define a testing plan that specifies what inspection test standards need to be applied to the artifacts being tested. For example, inspection testing the requirements entails activities such as careful proof reading of the prose, ensuring the figures are clear, and enforcing any domain-specific constraints for the requirements. Inspecting an implementation involves careful and thorough review of program code, typically performed by someone other than the code's author.

Inspection tests can be performed by individuals, as well as by groups. Group testing can be conducted in informal walkthroughs and in formal reviews. The members of review groups are all stakeholders for whom the inspection is relevant. The results of inspection tests are recorded and the appropriate authorities sign off that the tests have been conducted successfully. For example, a customer with authority to sign off on requirements does so at the culmination of a successful requirements inspection review. Any inspection test failures are recorded in the test record, for subsequent repair.

Analytic validation is typically performed using automated analysis tools, frequently in conjunction with an artifact build. Static analysis is performed to validate the structure of an artifact. Probably the most typical form of static analysis is that performed by a compiler on program source code. Static analysis tools are also available for other artifacts, such as spelling and grammar checking for requirements documents. Syntactic and semantic analysis tools are available for formal software models and designs.

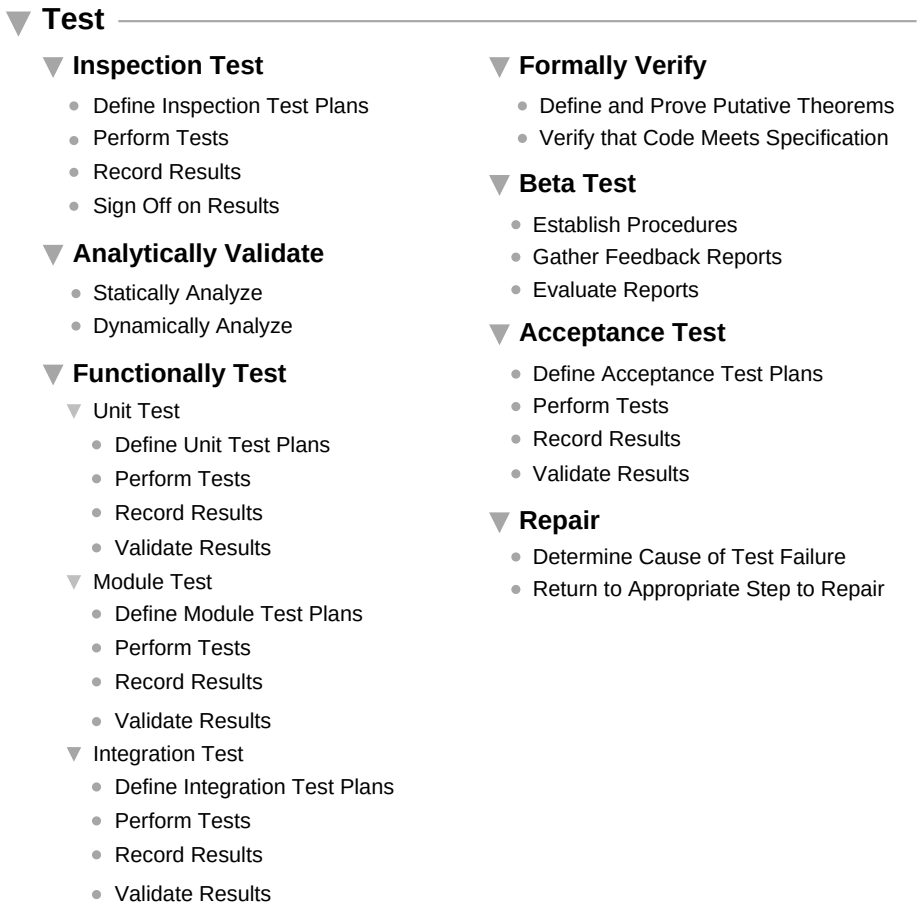


Figure 16: The Test step fully expanded.

Dynamic analysis validates the dynamic behavior of an executable artifact, most typically the implementation. For example, dynamic analysis tools can be used to validate the behavior of distributed and parallel programs, checking for such problems as deadlock or race conditions. Dynamic analysis is also used to validate that system performance is adequate to meet stated performance requirements.

Functional testing is performed on the operational program produced by the **Implement** step. Functional testing is carried out at three levels of program structure:

- unit testing of program functions (i.e., methods)
- module testing of program classes (or other encapsulation constructs)
- integration testing among the classes

At each of these levels, the testing steps entail defining the test plans, performing the tests, recording the testing results, and validating that the results are correct.

Formal program verification is the ultimate level of ensuring program correctness. Using verification techniques, the program and its specification are treated as formal mathematical constructs, about which formal theorems are defined and proved. *Putative* theorems are used to verify the specification itself, independent of the implementation. A putative theorem formally states a desired property that is then

proved correct with respect to the formal specification. Once the soundness of the specification is established, the program code is formally proved to meet the specification.

Beta testing is conducted by voluntary end uses, who are willing to test a product before its official delivery. Depending on the type of product and willingness of users, beta testing may be performed before functional testing is fully complete. The point is that beta test users are made aware that a software product has not yet been deemed ready for official release. Specific beta testing procedures are established, including how users will provide feedback to the developers. The developers gather the feedback reports and evaluate them to determine the appropriate course of action. The actions may be to add, remove, or change user-level features, in response to user suggestions. If beta testers encounter system-level bugs, then the developers proceed as when system tests reveal bugs, as described just below.

Acceptance testing is also conducted by end users, or their representatives, on the delivered software product. It entails the same sub-steps as functional testing, namely plan, perform, record, and validate. Whereas functional tests are defined in terms of the program implementation, acceptances tests are defined strictly at the end-user level of functionality. When validated, acceptance tests signal that the product is ready to be released to the full user community.

When tests of any kind fail, the necessary repairs must be made. This entails determining the cause of the failure, and returning to the appropriate process step(s) to effect the repairs. Probably the most well-known form of repair is program debugging. It is noteworthy that debugging per se is not an explicit step in this process. The reason is that debugging should not be carried out in an isolated, ad hoc manner. Rather, debugging is conducted in the context of well-defined test plans, which help significantly in isolating the cause of errors. Hence "program debugging" is defined here as the correction of errors discovered during the process of validating functional tests.

Tests may fail due to a flaw in any artifact, including the tests themselves. Hence, part of the testing process is determining which artifact(s) need repair when tests fail. For example, an implementation bug may be due to a flaw in the program logic, or a flaw in the plan that tests the logic. Upcoming chapters on testing details will discuss the specific means to effect repairs.

2.4.4. Document

Figure 17 shows a full expansion of the **Document** step. Documentation is produced for both developers and end users. For developers, the primary source of documentation is the software artifacts themselves, stored in the software repository. Specialized forms of developer documentation, such as reports and

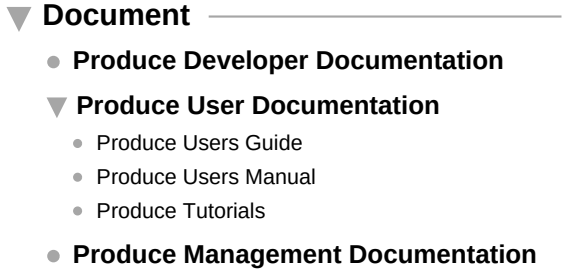


Figure 17: The Document step fully expanded.

summaries, can be helpful in some projects.

User documentation consists of brief user guides, full users manuals, and step-by-step tutorials. Some or all of these forms of end-user documentation may be called for, depending on the operational complexity of a delivered software product. Any level of user documentation can be integrated within the operational software in the form of online help.

Documentation for project managers consists of reports and other documents that relate to the managerial aspects of the project.

2.4.5. Reuse

The full expansion of the **Reuse** step appears in Figure 13 since it does not expand beyond one level. Software reuse is the part of the process where existing software components are put to use in a new software context. Libraries of reusable components are designed with the explicit intention of being reused. In some cases, components of previously developed software can be reused, even if reuse was not explicitly envisioned for the components when they were initially developed.

In order to perform reuse, existing components must first be researched, and then their potential for reuse must be determined. If such potential exists, the components are adapted as necessary to their new software environment, and employed there as they fit. If a project produces new components that are reusable, they can be configured for that purpose, and stored in a component library.

Conceptually, reuse applies to all ordered process steps. In practice, reusable artifacts have evolved in a bottom-up fashion, where reusable implementation artifacts are the most widely available. Some design reuse is practiced, typically in the form of design patterns. Significant reuse of specifications and requirements is not yet common practice.

2.5. Process Enactment

Enactment is the activity of performing a software process. The term "enactment" is used to connote a process carried out in a deliberative manner by people, rather than one that is executed as a program or conducted in some mechanical way.

There are a few steps of the software process that are fully automated, for example translation of program source code into executable code by a compiler. Many other computer-based tools can be used to assist in the development process. These tools provide functionality that is typically called *Computer-Aided Software Engineering* (CASE). While CASE tools can be extremely useful, they are in fact only aids. For the most part, humans provide the intelligence that makes the process happen. There is more on process automation and CASE tools later in the book. This chapter discusses enactment as performed by the people involved.

The development staff are the primary actors in process enactment. Other stakeholders certainly participate in the process, but it is the development staff who lead the effort. Also, the development staff perform the majority of the work that produces tangible deliverables. Of the stakeholders described in Section 1.2, the development staff is comprised of the analysts, implementors, testers, and managers.

There are two major forms of enactment -- ordered and pervasive. These forms are now described, in the context of the process steps outlined above.

2.5.1. Ordered Enactment

In general, the ordered steps of the process are enacted sequentially, as the term "ordered" suggests. The purely linear diagram shown earlier in Figure 6 is an oversimplification of sequential enactment, since it does not depict any form of iteration. In practice, software processes almost always involve iteration, since discoveries are made in later steps that require earlier steps to be revisited.

Figure 18 shows a refined diagram of ordered process enactment. This is the style of enactment broadly followed in the book. Four forms of iteration are depicted:

- An unlimited amount of iteration is possible among the **Analyze Specify**, and **Prototype** steps.
- There is also unlimited iterative feedback between the **Design** and **Implement** steps.

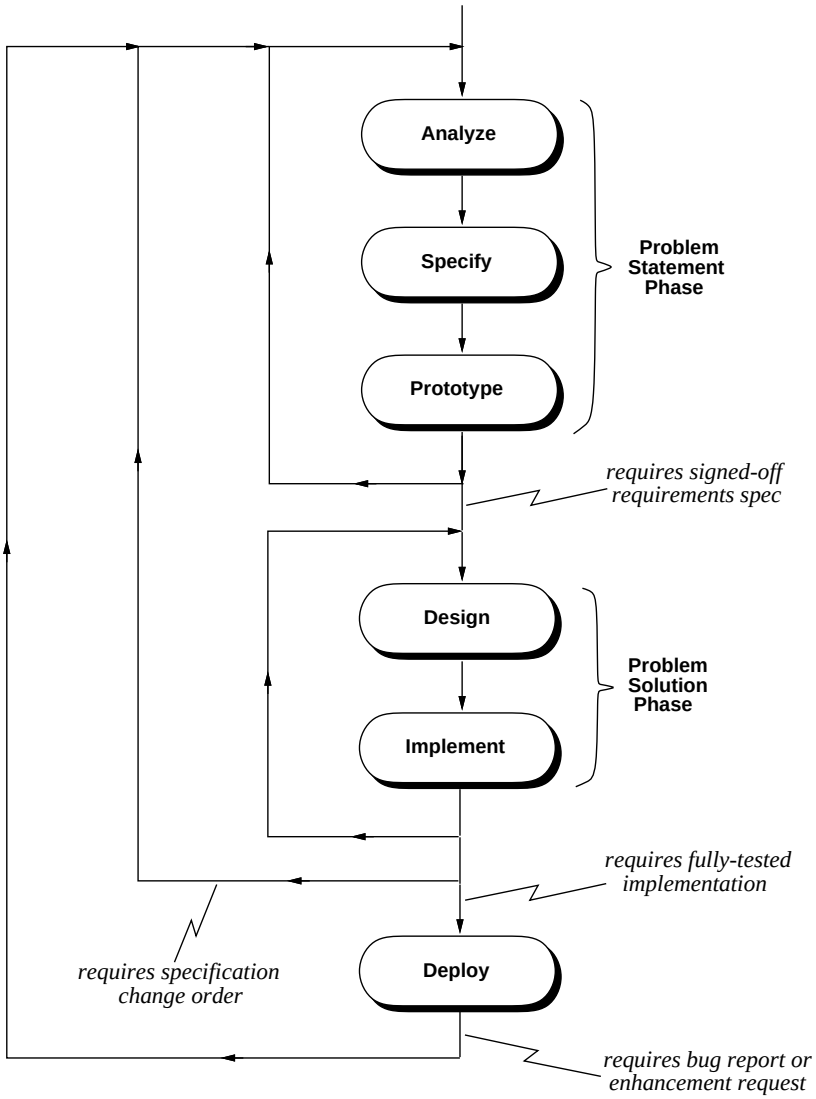


Figure 18: Iterative enactment of ordered process steps.

- Iteration from the **Design** and **Implement** steps back to **Analyze** is limited and controlled.
- Post-deployment iteration is also controlled.

The italicized annotations in the diagram indicate the conditions under which key process transitions are made. The transition into the design step requires that the preceding steps have been completed to the extent that a signed-off requirements specification has been produced. The general term "requirements specification" refers to the results of the first three steps of the process. "Signed-off" means that all artifacts have been fully tested and are complete to the satisfaction of all affected stakeholders.

The transition from the implement step back to analyze requires a *specification change order* to be defined. This change order clearly states the necessary modifications or enhancements that must be made to the requirements specification in order for the implementation to proceed.

The transition into the deploy step requires that the implementation has been fully tested. This is an entirely standard process requirement for any quality product. Namely, that the product is fully tested before it is released.

The transition from **Deploy** back into the development process is accompanied by a bug report for the deployed product, or a request for a product enhancement. In the case of a bug report, the iteration may not need to perform all of the process steps to effect a repair. For example, if a bug is identified as purely at the implementation level, the repair iteration can skip through all of the preceding steps straight to **Implement**. In the case of a product enhancement, the iteration will typically need to start at the top of the process, to analyze the user level requirements for the enhancement, and proceed from there.

A key rationale for this style of enactment is the importance of having a complete and sound requirements specification before the design and implementation begin. Further, it is important to control requirements specification changes during design and implementation. A common source of problems in large-scale software development can be of the "shifting requirements", i.e., requirements that change significantly once the design and implementation phases have begun.

Considering the software process as a traditional problem solving exercise can clarify why changing requirements and specification can be troublesome. The requirements analysis, specification, and prototyping can be considered the "problem statement" phase. The design and implementation are the "problem solution" phase. When the system requirements or specification change after the design and implementation are in progress, it is like changing the problem to be solved while the solution is being developed. Such changes can be problematic in the sense that having an unclear idea of what needs to be solved can make finding the solution difficult.

Since there are no conditions on the inner iterative paths, these may happen as much as the developers see fit. In this way, it is considered a normal part of development to have continual feedback among analyze, specify and prototype. The same is true for feedback between design and implement.

Within the two major problem solving phases, unlimited iteration is reasonable because each of the steps is focused on the same major task. Analyzing requirements, specify a model, and building a prototype are three different views of the problem statement. Discoveries made during the modeling and prototyping steps can help clarify and solidify the user-level requirements. Similarly, the design and implement steps are two levels of the same phase, such that iteration between them is a natural part of the process.

A practical rationale for subdividing the process into two major phases relates to the people involved and the differing skills required of the developers. As noted in Chapter 1, the analysts who lead the first phase are skilled in communicating with end users and defining requirements; they need good people skills. The implementors who conduct the second phase are skilled in the areas of software design and

programming; they need good technical skills. While there are people who are skilled in both of these areas, not everyone is able or desirous to excel in both analysis and implementation.

Another practical reason for the two-phase process relates to the organizations involved. In many cases, an organization may have the expertise to develop a requirements specification but not the product implementation. In such cases, the organization will outsource the implementation work on a contract basis to another firm that specializes in software design and implementation.

The style of enactment depicted in Figure 18 can be considered a "traditional" engineering approach. It is traditional in the sense that the problem statement is completed as thoroughly as possible before the solution begins. A more tightly iterative style of enactment is illustrated in Figure 19. In this case, there is only a single outer iteration, and the **Prototype** step is absent. The two basic phases of problem solving are still present, but the problem statement and solution are developed incrementally. The process begins by defining a part of the problem and proceeds to solve just that part. It then proceeds through successive iterations to refine and expand the problem and its solution. There is no limitation on the transition between the two problem-solving phases, as there is in the more traditional approach. In this way, a product evolves incrementally and hence this can be considered an *evolutionary* approach to enactment.

There are only two transition conditions. As in the more traditional style of enactment, moving from **Implement** to **Deploy** still requires testing, but in this case only of the partial implementation that is produced in each incremental iteration. A partial product is deployed to users, or a subset of them who

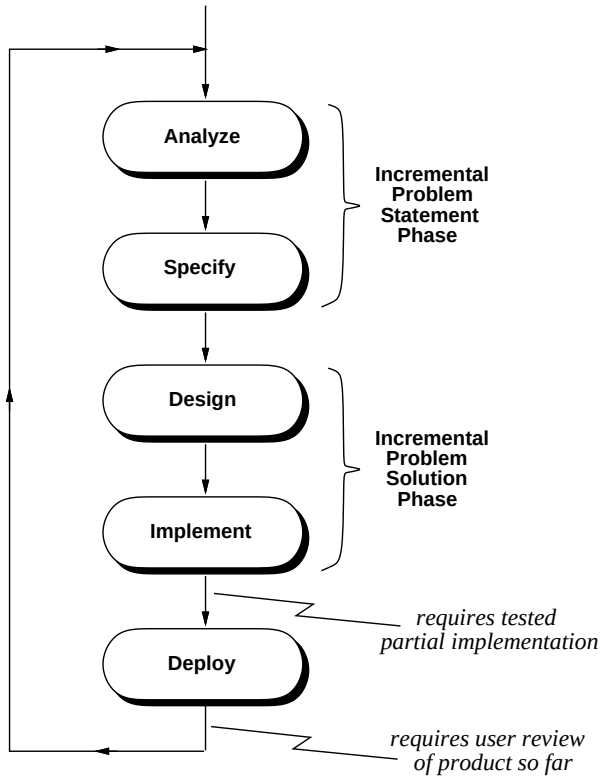


Figure 19: More tightly iterative enactment of ordered process steps.

participate actively in the process. The transition out of **Deploy** into the next development iteration requires users' review. They use and evaluate the product produced so far, such that further development can proceed to the users' satisfaction. At some point, when the users are fully satisfied, or when time and/or budget have been exhausted, the last deployed version of the product is considered the final deliverable.

The reason the **Prototype** step is absent is that the partially completed products produced in each pass of the process can be considered prototypes. Each version provides successively more functionality. In this case, a prototype is viewed differently than the potentially throw-away versions created in the traditional **Prototype** step. Each prototype provides less functionality than the final product, but ideally each version provides some actually usable functionality. That is, the users receive a partially useful product early in the process, and the product is successively refined through the iterations. At the end of some iterations, it may be that users do not like any of what they see. In such a case, the next iteration changes the requirements to meet the newly-discovered users' needs.

A key rationale for the evolutionary approach is that users see an actual working product sooner than in the traditional approach. This can help clarify to the users what additional product requirements are necessary. The evolutionary style of development does require that analysts, users, and implementors work more closely than in the traditional style. This may work in some cases, but not in others. As noted above, when an organization chooses to outsource implementation, the traditional approach is likely to be more appropriate.

People issues are also an important consideration in the evolutionary approach. Depending on the size of a project, it may be difficult for all end users to work effectively with programmers in a tight-knit development team. For this reason, an evolutionary approach may choose selected individuals to act as customer representatives during the process. The evolutionary approach also requires particular skills and attitudes from programmers. They must be able to focus clearly on what is required in each iteration, and deliver it quickly enough so that the users do not have to wait around. Implementors must also be willing and able to to communicate continuously with the users. Finally, implementors must be able to practice "egoless" programming, in that they are willing to abandon work they have produced in a development iteration if it is not satisfactory to the users.

The traditional versus evolutionary approaches to enactment may be considered two ends of an iterative spectrum. In an evolutionary approach with only a few iterations and larger deliverables in each, the approach gets closer to traditional. If the requirements for signing off on a requirements specification are loose, and several stages of deployment are planned, then a traditional approach becomes more evolutionary. As discussed earlier, an organization always needs to fit the process to the people and project at hand.

While the book generally follows the traditional form of process enactment, coverage of technical material is not fundamentally dependent on it. Material from all of the chapters can be applied to a traditionally-enacted process, an evolutionary process, or some point in between.

As with structural process details, defining enactment details helps to provide an organizational framework for presenting the book's subject matter. Where appropriate, there is discussion of how process enactment styles may affect the way development work is conducted and what artifacts are produced.

2.5.2. Pervasive Enactment

In conjunction with the ordered process steps, enactment of pervasive steps occurs continuously or at regular intervals. One way to make pervasive enactment concrete is to associate a specific schedule for pervasive steps or substeps of the process. For example, it is typical to schedule project meetings at regular

times. Establishing a weekly meeting schedule is a concrete way to enact the the Conduct Meetings substep of **Manage**.

Specific scheduling details are defined as part of standard operating procedures developed by the management staff, in conjunction with other affected stakeholders. For example, Table 6 shows a schedule for selected substeps of **Manage** and **Configure**.

Another way to carry out pervasive steps is to instantiate them as explicit ordered steps of the process. For example, Figure 20 shows a testing step instantiated between **Analyze** and **Specify**. The instantiation of a pervasive step specializes it, based on its placement in the ordered process. In this example, the generic testing step is instantiated to test the requirements artifact that is the result of **Analyze**. If the **Test**

Process Step	**Schedule**
Manage:	
Management meets	8AM every Monday
Technical staff meets	1PM every Monday
Supervisors review staff	Last Friday of every month
Configure:	
Developers check in and build	5PM every night
Developers do internal release	5PM every Friday

Table 6: Typical schedule for some pervasive process steps.

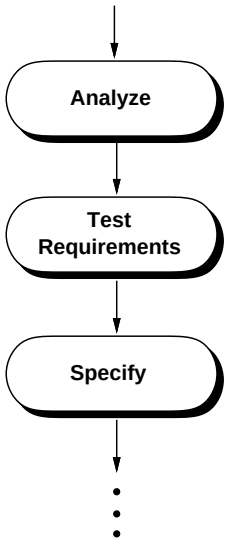


Figure 20: Testing step instantiated between Analyze and Specify.

step is fully instantiated, a specialized version of it follows each top-level step, to test the particular artifacts produced by each. Table 7 summarizes the manner in which pervasive steps are enacted in the process followed throughout the book. The **Manage**, **Configure** and **Document** steps are scheduled. The **Test** and **Reuse** steps are instantiated. These enactment styles are reasonable in general, but as always, processes must be specialized for people and products involved.

2.5.3. Enactment Details

The concepts just discussed provide a large-grain view of process enactment. Details missing from this view include the following:

a. the artifacts produced by the steps
b. enactment within the top-level steps
c. the frequency of iteration
d. what steps, if any, may be enacted in parallel
e. most scheduling details
f. precise conditions under which the steps start and finish

Item a is the subject of the next chapter. Item b is addressed throughout the technical chapters of the book. Items c through f are also addressed in the technical chapters, but primarily in Chapter 24 on project management. That chapter defines concrete process scheduling details for a typical project. These details are part of standard operating procedures (SOPs), which define precisely who does what, and when. The SOPs also define specific conditions that must be met before and after process steps are enacted, i.e., the preconditions and postconditions of the steps.

2.6. Well-Known Process Models

Taken together, the steps of a process and its style of enactment can be considered a *process model*. The model presented in this chapter, with its basis in traditional problem solving, is reasonably mainstream in comparison to models that appear in the literature. This section analyzes well-know process models and

Step	How Enacted
Manage	At regularly scheduled times (e.g., meeting) and continuously (e.g., supervision of personnel)
Configure	At regularly schedule times
Test	Instantiated after each top-level ordered step and at regularly scheduled times in some cases (e.g., monthly inspections)
Document	Scheduled before major releases.
Reuse	Instantiated before each step.

Table 7: General style of enactment for pervasive steps.

compares them to the process used in the book.

Overall, there is a clear thread of commonality among various process models. Virtually all models include requirements gathering, design, implementation, and testing as parts of the process. What distinguishes the models is less about the steps themselves than the manner in which they are enacted and the relative emphasis placed on each step. What distinguishes the models also has little to do with their catchy names. For example, the most salient feature of the "spiral model" is not its shape, but rather its focus on continual risk assessment throughout the process.

2.6.1. Waterfall

One of the earliest versions of a software process was published by Winston Royce in 1970 [Royce 70]. Figure 21 is a diagram of the process as originally presented. The diagram has been dubbed a "waterfall chart", due to its depiction of process flow downward from one step to the next.

The steps shown in the original waterfall diagram are notably similar to those still widely used today. Table 8 shows a reasonable correspondence between the waterfall steps and the ordered steps shown in Figure 6.

In its position as one of the earliest published models, the waterfall process has been the subject of many critiques. One of the most frequently raised criticisms is that the model is too inflexible in its strictly sequential style of processing. It is unrealistic to expect that each step will be fully completed before the

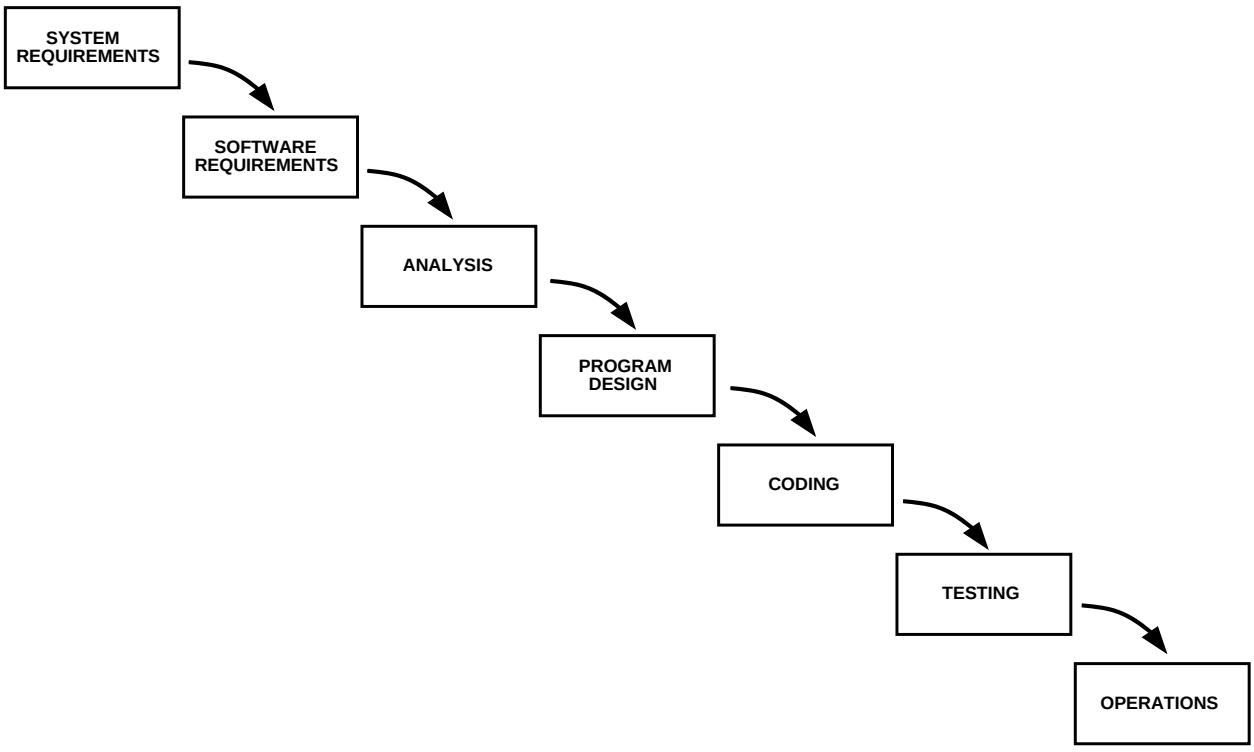


Figure 21: The original waterfall process model.

Waterfall Step	Ordered Process Step
System Requirements Software Requirements	Analyze
Analysis	Specify
Program Design	Design
Coding	Implement
Testing	Test Implementation
Operations	Deploy

Table 8: Waterfall steps in correspondence to the ordered steps in Figure 6.

next is begun. Water must sometimes flow uphill.

To his credit, Royce was aware that process iteration may be necessary. However, he suggested that it be limited as much as possible, and ideally occur only between immediately adjacent steps. This has led to a widely-held perception that the waterfall model is strictly sequential. However, it is easy enough to add iterative paths in a waterfall chart, and to consider iterative enactment to be a regular part of the process. This is essentially what the iterative enactment presented in Section 2.1.3 is about.

A more significant criticism of the waterfall model is the relegation of testing to near the end of the process. Many have observed that waiting until after implementation is too late to start testing. Errors in artifacts produced in the earlier steps can be very difficult to detect if they become embedded within the implementation. In most process models that have succeeded the waterfall, testing has become a more pervasive part of the process. In addition, the need for other pervasive steps has become clear.

Another critique of the waterfall as originally presented is its lack of detail in how each step is conducted. For example, the high-level waterfall presentation says little about how the `PROGRAM DESIGN` step transforms the result of the `ANALYSIS` step into a design artifact. As process modeling has matured, the presentation of new models has often included details of process enactment that were missing from the original waterfall.

Given its status as first on the block, the waterfall process has been fodder for much discussion. Despite a good deal of criticism, it is fair to say that many of the process basics presented by Royce are still valid today. The waterfall model remains an enduring benchmark of comparison against which other models are regularly compared, even if the comparison is negative.

As illustrated Table 8, there is commonality between the process used in the book and the original waterfall model. The significant advances in the book's process address key criticisms of waterfall, namely

- pervasive testing and other pervasive process steps
- more process iteration
- definition of process details

2.6.2. Spiral

The spiral process model was developed in 1988 by Barry Boehm [Boehm 88]. Figure 22 is a diagram of the model, as presented originally by its author. There are four pervasive activities in the spiral model,

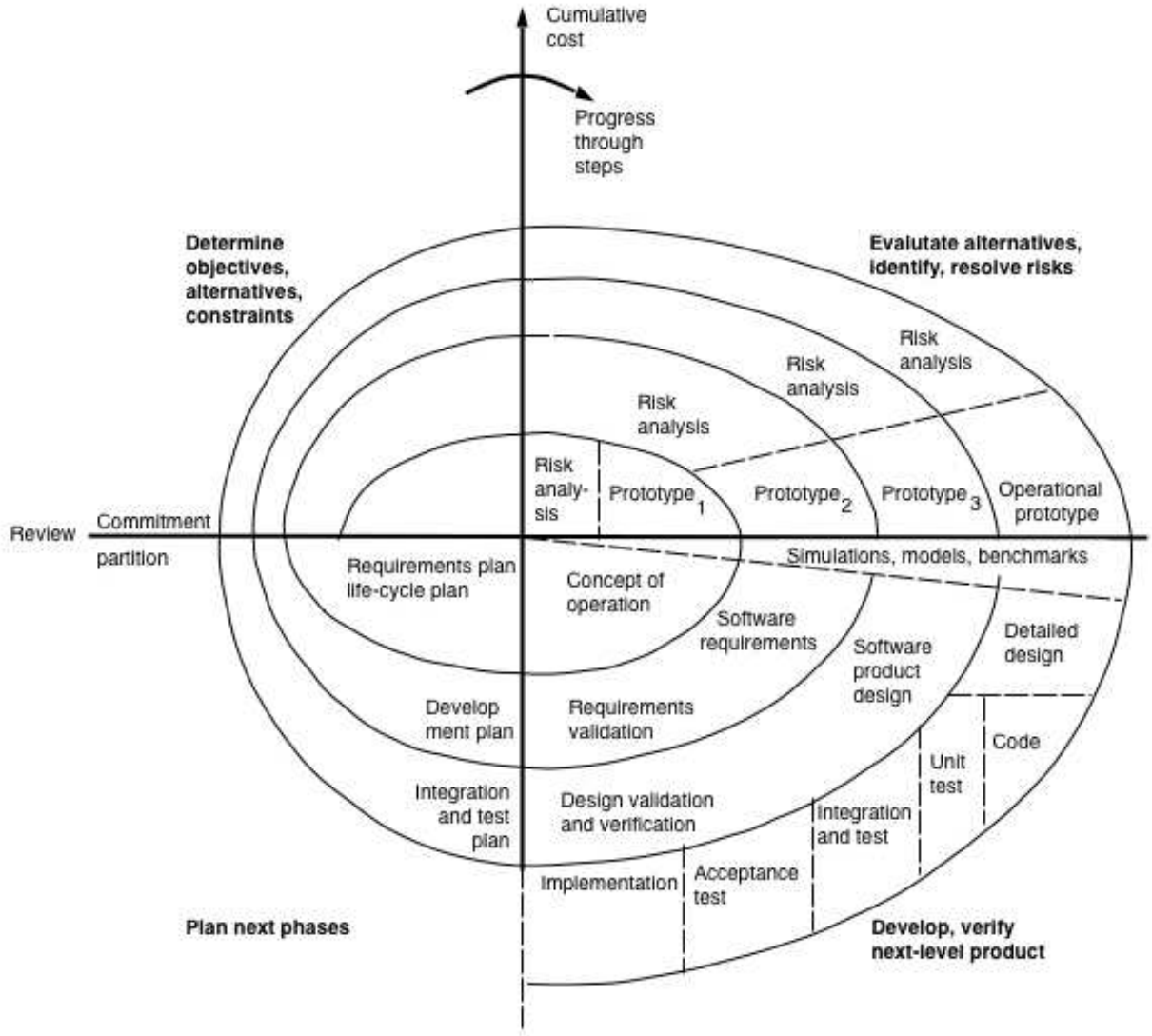


Figure 22: Spiral process diagram as originally presented.

shown in the four quadrants of the diagram:

- *Determine objectives* -- set out what is to be accomplished in the next phase of development.
- *Evaluate risks* -- determine the risks involved, using prototyping to help evaluate alternatives.
- *Develop and validate* -- develop the next phase of the project and validate the resulting artifacts.
- *Plan* -- plan for the next phase of the project, based on the results of the just-completed phase.

Each loop of the spiral represents a phase of development. The phases are indicated in the top part of the lower right quadrant:

1. *Concept of operation* -- formulate broadly what the software is intended to do.
2. *Software requirements* -- analyze user requirements.

3. *Software product design* -- design the high level software architecture.
4. *Detailed design and code* -- implement the product.

An important aspect of the spiral model not directly evident in the diagram is the dynamic development of process details. Part of the planning activities for each phase includes determining the process details for the coming phase. For example, the *Requirements and Life-Cycle Plans* lay out the details for the requirements analysis phase of the process. The *Development Plan* contains details of what type of design and implementation process to follow. The plans also determine how iterative the process will be, i.e., how many expansions of the spiral will be enacted.

To provide a common frame of reference, Figure 23 shows the spiral process in the style of process diagram used in the first part of the chapter. The diagram illustrates that there is nothing particularly important about the spiral shape of the model. Compared to other process models, the two most distinctive aspects of the spiral process are these:

a. the continual assessment of risk to guide process enactment
b. the dynamic development of process details, based on risk assessment and planning

Risks are any adverse circumstances that can impair the development process or negatively affect product quality. An important part of risk analysis is to identify the high-risk problems versus those of lower risk. In doing so, process planing then proceeds to avoid the high-risk areas. The development of prototypes is

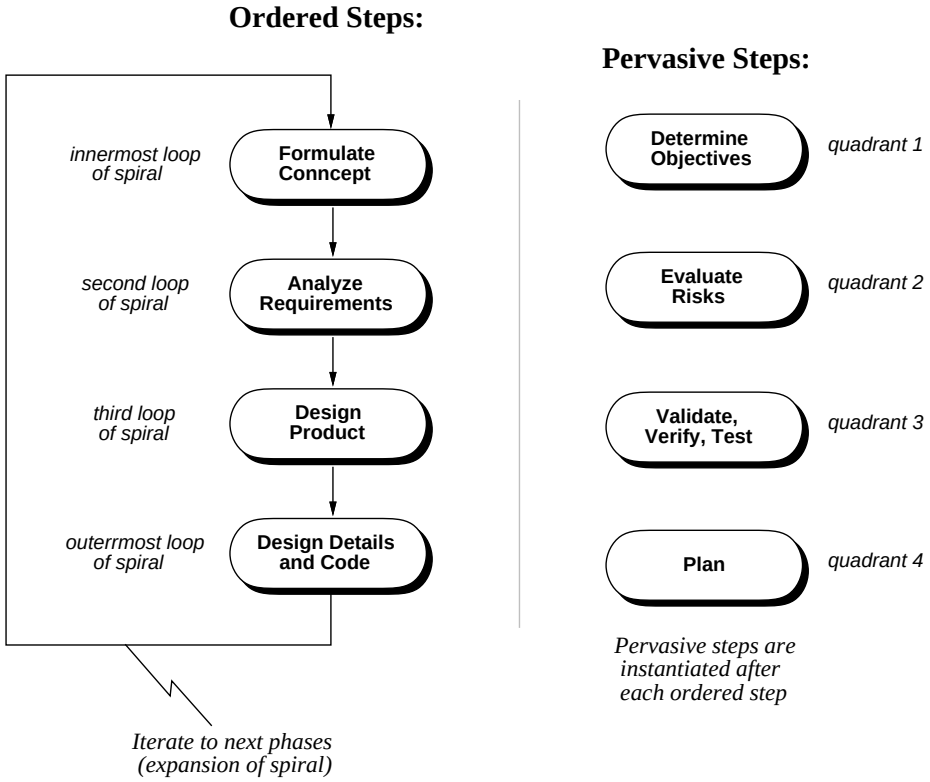


Figure 23: Spiral process unrolled.

part of the risk identification process. For example, if there are two possible approaches to development, two prototypes using each approach can be developed to determine which approach shows better promise.

From a business and managerial perspective, risk assessment is a discipline in its own right. Hence, in order for a spiral process to be successfully enacted, the development team must include individuals who are skilled in risk analysis.

The key principles of the Spiral model are treated with less emphasis in the process used in this book. In the book's process, risk assessment is performed as part of the **Analyze** step, not pervasively. Also, dynamic process development is de-emphasized, since most process details are defined in advance. Dynamic process updating is included as a pervasive activity, but as a lower substep within **Manage** (see Figure 14). Spiral practices could be more fully incorporated into the book's process by elevating risk analysis and dynamic process development to immediate substeps of **Manage**, or to top-level steps.

2.6.3. V-Model

The V-Model [German 93] is a comprehensive software process. It was developed for the German defense ministry beginning in 1986. The model addresses most of the process structure and enactment issues discussed in this chapter. It also defines standards for the tools to be used in the software development process. The V-Model has been used extensively in European industrial practice.

The development steps of the V-Model are essentially those shown in Figure 6. A noteworthy contribution is its early recognition that testing needs to be performed at each step of the ordered process. The model defines detailed standards for testing the implementation, the design, and the requirements. It also thoroughly defines procedures for configuration control and project management.

Overall, the V-Model is a good example of a comprehensive, industrial-strength process that has been used extensively in actual practice. For this reason, it is worthy of mention in a survey of well-know models.

The book's process is similar in overall structure to the V-Model. The primary differences are in technical process details. Also, the book's process does not specify specific tools to be used for development.

2.6.4. Cleanroom

The Cleanroom Process was introduced in 1987 by Harlan Mills [Mills 87] and updated in 1994 by Richard Linger[Linger 94]. Figure 24 depicts the 1994 version of the process. The hallmark of the Cleanroom process in its use of formal specification and verification to produce software that has, as its proponents claim, near zero defects.

As with most traditional process models, Cleanroom begins with analyzing customer (user) requirements. Unlike other models, Cleanroom does not define details for this process step, assuming that a stable set of requirements are provided as an input to subsequent development steps.

The `Specification` step of Cleanroom defines functional and usage specifications. The functional specification defines the required external behavior of the software, much like the **Specify** step discussed in Section 2.3.2. The usage specification defines scenarios of correct and incorrect usage. These are defined in terms of the formal functional model, not in end-user terms.

The step of "Incremental development planning" determines the number and extent of process iterations to be conducted in the subsequent development. That is, this planning step decomposes the development effort into pre-defined segments, each of which is developed in a separate process iteration. The

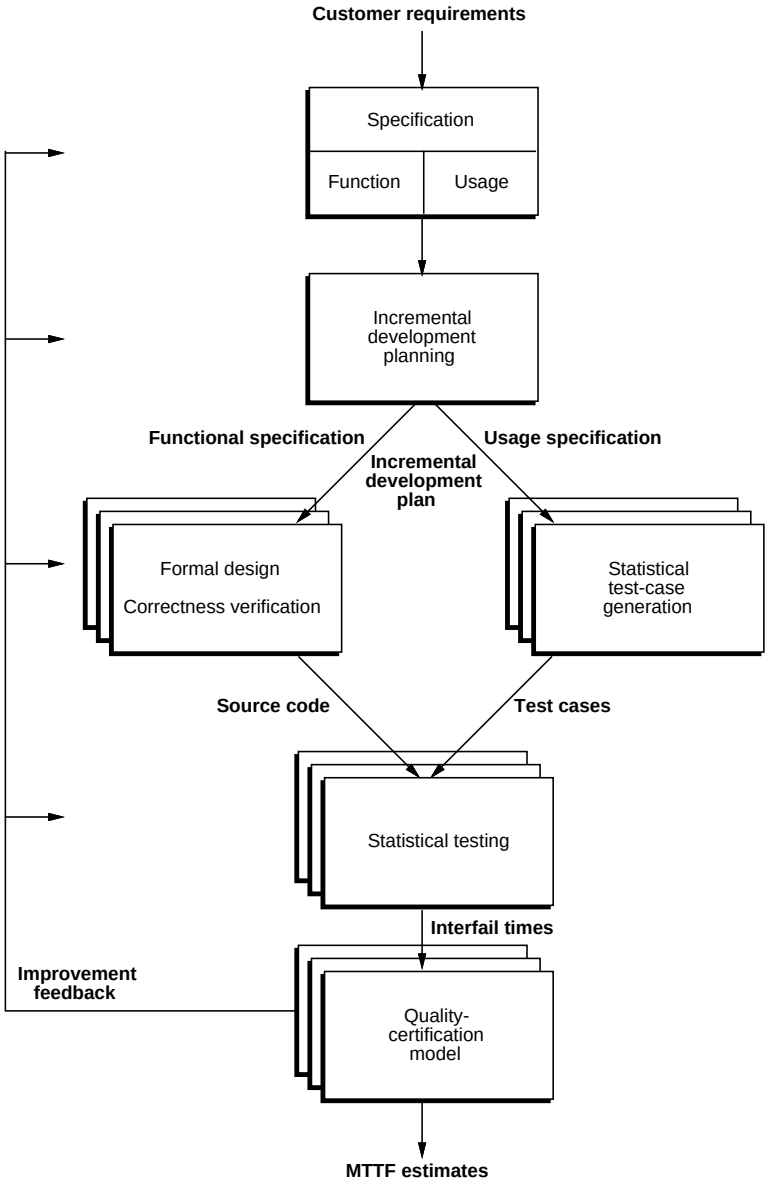


Figure 24: Cleanroom.

stacked boxes in Figure 24 depict this iteration within the development steps.

The step labeled "Formal design and correctness verification" encompasses design, implementation, and formal program verification. The parallel Statistical test-case generation step defines test cases for integration-level testing, to augment the formal verification.

The "Statistical testing" step conducts tests by applying the generated test cases to the program source code. The "Interfail times" output is a statistical measure of program correctness based on the concept of *mean time to failure*. The goal is to minimize the failure rate as much as possible.

The final step on "`Quality certification`" entails user-level acceptance testing of the software. The output is a refined estimate of mean time to failure.

The outer iteration loop in the process provides feedback from the culminating step back to the earlier steps of the process. This feedback is used to improve the product at the appropriate levels of development.

As designed, the Cleanroom process is well-suited for projects in which the requirements can be thoroughly determined in advance, since the process does not include the requirements analysis phase within any of the process iterations. Cleanroom also requires specialized skills from the developers in the areas for formal specification and verification. These skills are required in any process that involves formal methods.

With its emphasis on formal methods, the Cleanroom process is similar in spirit to the one used in this book. A significant difference is in the treatment of requirements analysis. In the book's process, the **Analyze** step is iteratively integrated with other steps, particularly with **Specify**. This allows the benefits of formal specification to be realized during the analysis of user requirements. The book's process and Cleanroom also differ in a number of technical and managerial details, particularly in the area of testing. While Cleanroom treats testing as a pervasive process, it focuses predominantly on formal verification and statistical quality control to ensure correctness, without using other forms testing employed in the book's process.

2.6.5. Agile

Agile development is rooted in the Extreme Programming (XP) methodology introduced by Kent Beck in the late 1990s [Beck 99]. The XP methodology evolved into the more general Agile Software Development process [Martin 03].

Agile development is a highly iterative process that is aimed at developing software rapidly by the means of strong user/implementor interaction. User requirements evolve and change throughout the development process, rather than being determined largely in advance of design and implementation. Rapidly changing requirements are considered a natural part of the Agile process. This is in contrast to the more traditional process view, where widely changing requirements are often considered problematic during the design and implementation phases.

The Agile process is well summarized in its *manifesto*, which is centered around the following values:

- **Individuals and interactions** are valued over processes and tools
- **Working software** is valued over comprehensive documentation
- **Customer collaboration** is valued over contract negotiation
- **Responding to change** is valued over following a plan

Agile developers do not entirely reject the items on the right, but value the items on the left more highly.

The first value is rooted in what agile proponents see as overly cumbersome software processes, under which participants can lose sight of the main project objective -- a working software product. A key goal for the agile process is to minimize time spent on unnecessary administrative tasks that take time away from work devoted directly to product development.

The second value is based on the idea that working software is *the* goal of a development process, with documentation highly secondary. In particular, a surviving requirements document is not considered necessary once the working software is produced.

The third point of the manifesto extends from what agile proponents see as another significant hindrance of more traditional processes -- that of contractually-binding plans and proposals. Agile proponents contend that software can be much more effectively produced in an environment largely if not entirely free of such contractual encumbrances.

The last point of the manifesto is likely the most fundamental statement of the agile process values. Rather than following a fixed plan, developers should be free to adjust to changing circumstances as the project and product evolve. In particular, they must adjust to changing requirements.

Figure 25 is a depiction of the agile development process in the diagram notation used earlier in this chapter. In keeping with the highly iterative development strategy, requirements are analyzed in very small increments, called *user stories*. A story is a simple informal description of a requirement, written in only a few words. Rather than elaborating the requirements into a larger document and software model, the process proceeds directly to the testing and implementation steps.

A key part of Agile development is a *test-first* principle. This means that as requirements are gathered, test cases are defined that will be used to ensure that the implementation of each requirement is correct. This helps solidify the understanding of the requirements and ensure the fully pervasive development of

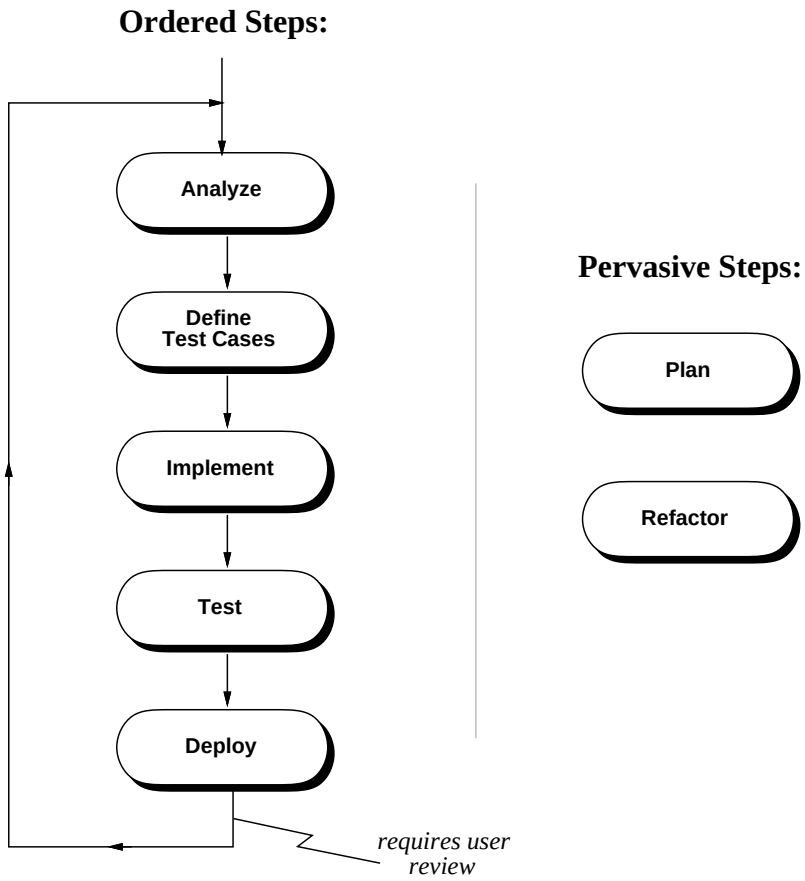


Figure 25: The agile process illustrated.

testing. Implementation and testing then proceed, with the goal of deploying a small, but working program as quickly as possible. Deployment increments can be as short as a few weeks.

The users examine each incremental deployment, and comment on whether it meets their needs and expectations. The users then modify their previous requirements if appropriate, define new requirements, and the process proceeds with another iteration. One of the tenets of the Agile development is that users can better understand and evolve their requirements if they can see working versions of a software product as it evolves. This is the same concept behind building a rapid prototype in a more traditional process. In the case of Agile development, the potentially throw-away prototype is replaced with an actual working version of the product, produced in small increments.

In order for the Agile process to proceed smoothly, pervasive planning is necessary. The planning entails organizing project personnel, prioritization of requirements, and determination of the work to accomplish in each process iteration. The pervasive planning step is therefore instantiated at the beginning of each process iteration, and at other points during the process as necessary.

The other major pervasive step is *refactoring*. This is the step in which software design activities are conducted. Rather than producing a design up front as in a more traditional process, the design is derived bottom-up from the implementation. Many of the same design principles are employed in Agile development as are employed in a traditional process. These include the use of design patterns and other well-established design techniques.

Refactoring also involves the refinement and reorganization of the incrementally-developed implementation. When code is developed rapidly in response to user requirements, it may not be well organized in its initial form. Refactoring employs standard practices of good code development to improve the structure and efficiency of the implementation as it evolves. Refactoring at both the design and implementation levels is performed pervasively, at regularly scheduled points in the process.

The agile style of development is not without controversy. It’s detractors see it as throw-back to the "bad old days" when programmers built programs in an undisciplined and unmanageable manner. Its supporters see it as highly effective means to deliver quality products to satisfied users in less time than traditional methods. While proponents have reported substantial success, there are at present few empirical studies that support the effectiveness of agile development [Abrahamsson 03].

As a form of highly evolutionary development, agile processes are subject to the people issues discussed earlier in Section 2.5.1. Namely, users and implementors must be willing and able to work together in a tight-knit team, and implementors must be highly skilled in delivering incremental product versions in a timely manner. A significant unanswered question about agile development is whether it can scale up to large-scale development projects. Further experience and study are necessary to address this question.

Many view agile development as fundamentally at odds with more traditional processes, such as the one followed in this book. There is in fact much in common between an agile processes and a traditional process enacted in a highly iterative manner. In particular, the following simple adaptations of agile development would bring it substantially in line with the iteratively-enacted traditional process shown in Figure 19:

- instead of discarding user stories gathered during requirements analysis, add them incrementally into a surviving requirements document
- during the test-definition step, develop formal specifications in conjunction with the test cases
- instantiate the refactor step before implementation, having it become an incremental design step

While agile proponents might balk at these suggestions, they serve to illustrate that seemingly disparate

process models are really not irreconcilably different.

2.7. Postscript

Most practitioners and academics agree on the fundamental steps in the software process. By one name or another, the top-level steps illustrated in Figure 6 are widely accepted. Different process models vary in how and when the steps are enacted, what emphasis is placed on each, and what concrete deliverables are produced.

Much of what this book has to offer can be readily adapted to other types of processes than the one laid out in this chapter. If you don't like this process, define one of your own. You can take the book's process as a starting point and change some things. Or you can scrap it entirely and start from scratch. While defining a process may be somewhat time consuming, the basics for doing it are straightforward:

a. define the steps and substeps of the process, i.e., what people do;

b. define the style(s) of enactment, i.e., what order things get done in;

c. define the artifacts, i.e, what gets produced (this topic being covered in the next chapter).

Taking the time to define and use a software process is time well spent. For small projects, with a tight-knit team, it may be possible to build quality software without a well-defined process. In just about any other situation, not having a good process is a bad idea.

References

[Abrahamsson 03]
Abrahamsson, P., J Warsta, M. T. Siponen, and J Ronkainen. New Directions on Agile Methods: A comparative Analysis, *Proceedings of the 25th International Conference on Software Engineering*, p. 244-254 (May 2003).

[Beck 99]
Beck, K.. Embracing Change with Extreme Programming, *IEEE Computer* **32**(10) p. 70-77 (October 1999).

[Boehm 88]
Boehm, B. W. A Spiral Model of Software Development and Enhancement, *IEEE Computer* **21**(5) p. 61-72 (May 1988).

[German Federal Ministries of Defense 93]
German Federal Ministries of Defense. *V-Model Lifecyle Process Model, Brief Description*, General Reprint No. 250 (February 1993).

[Linger 94]
Linger, R. C.. Cleanroom Process Model, *IEEE Software* **11**(2) p. 50-58 (March 1994).

[Martin 03]
Martin, R. C. *Agile Software Development*, Prentice Hall (2003).

[Mills 87]
Mills, H. D., M. Dyer, and R. Linger. Cleanroom Software Engineering, *IEEE Software* **4**(5) p. 19-25 (September 1987).

[Royce 70]
Royce, W. W. "Managing the Development of Large Software Systems", *Proceedings, IEEE Wescon*, p. 1-9 (August 1970). Reprinted in *Proceedings of the Ninth International Conference on Software Engineering*, p. 328-338, (1989).