1 Software Development Lifecycle

In normal usage, the term *life cycle* means “the changes that happen in the life of an animal or plant” (Cambridge, 2003). In software engineering, the term *lifecycle* (normally written as a single word) is applied to artificial software systems to mean the changes that happen in the “life” of a software product. Various identifiable phases between the product’s “birth” and its eventual “death” are known as *lifecycle phases*.

The changes, and therefore the phases, are gradual. The product is *phased in* – introduced in stages. Hence, iterative and incremental development. Eventually, the product is *phased out* – gradually stopped being used. Hence, stepwise retirement. A reasonable way of thinking is that software is, at any time except deployment stage, either in phase-in or phase-out period. Software maintenance, despite its evolutionary character, begins the phasing-out process.

Figure 1-1 shows typical software *lifecycle phases* (explained in more detail in Section 1.2). The phases are:

1. requirements analysis
2. system design
3. implementation
4. integration and deployment
5. operation and maintenance
Figure 1-1 demonstrates that once a software product is introduced into an organization, it stays there for ever, although under different “reincarnations”. There is no possibility for an organization to return to a manual way of conducting business. Once in operation, the software product is maintained “to death”.

Maintenance, even if it evolves the system, leads eventually to deterioration of its original architectural design. The system becomes a legacy system – it cannot be “perfected” any more and even housekeeping and corrective maintenance become a major challenge. The entire system, or major components of it, must be phased out. The realization that the system is a legacy results in a decision to develop a new system. This starts a new lifecycle shown at the bottom of Figure 1-1. The phasing out of the old system and the phasing in of a new system are conducted in parallel until the new system is deployed to the users. Even after deployment, the old system may stay operational for some time until the new system can demonstrate its production usefulness.

Characteristic to Figure 1-1 is the absence of testing as a lifecycle phase. Testing – like project management activities, including the collection of project metrics – is an all encompassing activity that applies to all phases of the lifecycle.

1.1 Software Engineering Quintessence

Understanding of the software lifecycle is conditional on the understanding of the quintessence of software engineering – its fundamental nature, the context of software production. The quintessence of software engineering is captured in the following key observations:

- software system is less than enterprise information system
- software process is part of business process
1. SOFTWARE DEVELOPMENT LIFECYCLE

- software engineering is different from traditional engineering
- software engineering is more than programming
- software engineering is about modeling
- software system is complex

1.1.1 Software System Is Less Than Enterprise Information System

A software system is merely a part of a much larger enterprise information system. An implication is that the development of a software system is just an activity (albeit a fundamental one) in the development of an enterprise information system. A Venn diagram in Figure 1-2 demonstrates the inclusion of a software system in an enterprise information system. It shows also that an enterprise information system is a component of the enterprise as the whole and that the enterprise is a part of the business environment.

An information system is concerned with generating and managing information for people. Some of this information is generated automatically by computer systems. Other information is generated manually by people. The point is that information systems are social systems that encompass and use software and other components for the benefit of the enterprise. Benson and Standing (2002) list the following components of an information system:

- people
- data/information
- procedures
- software
- hardware
- communications

For example, a bank account management system consists of bank tellers, data/information about customers and their accounts, procedures governing the withdrawal and depositing of money, software able to process data/information, hardware on which the software can run (including automatic teller machines), and personal and automated communication channels that shackle all these components together.
1.1.2 Software Process Is Part Of Business Process

A *process* refers to the manner in which work activities are scheduled, organized, coordinated and performed at a certain period of time in a given place in order to produce product or service. The difference between software process and business process stems from and relates to a product or service expected from these processes. The result of a *software process* is software. The result of a *business process* is business.

There is a clear relationship between software and business. Software is potentially a major contributor to business success. Software is part of business, but not vice versa. In fact, this subset/superset relationship has been depicted in Figure 1-2. Enterprise in Figure 1-2 is another term for business. The purpose of enterprise is to realize a value creation chain, which serves the realization of business mission, objectives and goals.

The difference between software process and business process is akin of the difference between process efficiency and effectiveness. *Efficiency* means doing something right. *Effectiveness* means doing a right thing. In organizational terms, effectiveness implies attainment of business mission, objectives and goals. These are all deliverables of strategic planning processes conducted by the enterprise. Part of strategic planning is *business modeling*. Hence, business processes aim at delivering effectiveness.

By contrast, software processes aim at delivering efficiency. It is, therefore, possible for a software process to deliver a very *efficient* product or service, which will be *ineffective* to the business. At best, ineffective can mean neutral to the business. At worst, it can make the business vulnerable to competition and even bring it to bankruptcy.

Clearly, a software process is an inherent part of a business process, vital to the success of an enterprise. To deliver effectiveness as well as efficiency, a software process must be a contributor to a business process. After all, a decision to develop a software product or service is, in the first place, an outcome of strategic planning and business modeling.

The discipline of software engineering realizes the alignment of software and business processes. On one hand, software development is increasingly placed in the context of business modeling. Chapters 6 and 7 of this book are a clear manifestation of this trend. On the other hand, software development strives to deliver products and services of increasing business value to an enterprise. This has to do with three *management levels* that business processes service – operational, tactical and strategic.

Placing software development in the context of business modeling means that a software process is derived from a wider business model and it tries to support and implement a particular business process in that model. This means that a software product/service cannot be just an information service. It should also implement and assist in business actions. The design of an information system should either explicitly identify a business process it serves or, better, it should be a part of a (business) *knowledge management system*. One aspect of such design is a coordination between automated informational actions, manual supportive actions, and creative decision making actions.

Usually, a software system services a single management level – operational, tactical or strategic (Figure 1-3). The *operational level* processes business operational data and documents, such as orders and invoices. This is the realm of *OnLine Transaction Processing* (OLTP) systems assisted by conventional *database* technology. The *tactical level* processes information obtained from the analysis of data, such as monthly trends in product orders. This is the realm of *OnLine Analytical Processing* (OLAP) systems assisted by
data warehouse technology. The strategic level processes the organizational knowledge, such as rules and facts behind a highly profitable product selling. This is a realm of knowledge systems assisted by knowledge base technology.

Software products/services at operational management level are indispensable to the enterprise. Without them, a modern enterprise cannot function. However, operational software does not provide to the enterprise any competitive edge. Competitors work with similar software systems. The business value of software increases with higher management levels that the software applies to.

Interestingly, software products/services that are potentially of highest business value to the enterprise are the most difficult to automate. Understandably so. Strategic management is about organizational knowledge and wisdom. As noted by Benson and Standing: “Wisdom and knowledge exist only in the minds of people. When people talk about knowing something such as a telephone number, they are really talking about data. Understanding how to use that piece of data is knowledge. Deciding not to call someone at 3.00 a.m. is an example of wisdom.” (Benson and Standing, 2002, p.77). Processing and transfer of knowledge (not to mention wisdom) is and will remain mostly a social phenomenon, motivational and intuitive, not technical and predictable.

1.1.3 Software Engineering Is Different From Traditional Engineering

The fact that a software system is a component of an information system implies that software engineering is an aspect of a broader discipline of systems engineering. Consequently, a software engineer must understand the requirements of the whole system and must be competent in the system’s application domain to engineer the interfaces that the software must supply to its environment. A software engineer must also understand that some data/information processing can be better done in hardware than in software and that some processing may not need (or cannot be) automated at all.

System engineering is concerned with studying principles that govern the internal workings of complex systems. There is a long history of using systems engineering in traditional engineering disciplines, such as mechanical or electrical engineering. The discovered principles of systems engineering are formalized in mathematical models. The models are validated and applied in engineering products. These products are material in nature – bridges, buildings, power stations.
Not so with software products. As observed in the seminal work of Brooks (1987), software is immaterial in nature. Classical mathematical models apply to some but not to all aspects of software. Software is defined in fuzzy terms – “good”, “bad”, “acceptable”, “satisfying user requirements”, etc. Similar qualities are used in the service sector where quality is associated with fuzzy terms such as “good service”, “customer convenience”, “competence”, “job knowledge”, etc. Software engineering may be tackling “fuzzy” problems but this does not mean that it must be less rigorous or not provable. It is just that software engineering should use different branches of mathematics, such as fuzzy logic or rough sets, to provide rigor and proof.

In this context it is worth to observe that rigor is not the same as formality. A software process may be rigorous even though not formally proven by mathematical laws. Indeed, this is the case even in classical mathematics. As pointed out insightfully by Ghezzi et al. (2002, p.43): “Textbooks on functional calculus are rigorous, but seldom formal: Proofs of theorems are done in a very careful way, as sequences of intermediate deductions that lead to the final statement; each deductive step relies on an intuitive justification that should convince the reader of its validity. Almost never, however, is the derivation of a proof stated in a formal way, in terms of mathematical logic.” Interested readers are advised to reach for a seminal paper on social processes and proofs of theorem and programs by Millo De et al. (1979).

Software engineering does not need to be a poor cousin of traditional engineering. It is just different. Nobody in traditional engineering expects that a bridge built to mathematical models will collapse. Similarly, a “good” software “satisfying user requirements” should not fail. But, there is a but – provided that the user requirements and expectations or external circumstances have not changed drastically in the meantime.

Nobody expects a bridge to be moved by ten meters after it has been built. Similarly, nobody should expect a software product to happily perform different tasks after it has been built. If this is what is expected then the software has not failed. The software has only become unusable or unacceptable because the business conditions or external environment changed. If a river corridor moved by ten meters because of the recent flooding, a civil engineer cannot be blamed and must not be expected to effortlessly move the existing bridge to span the new corridor.

This said, a software engineer must be prepared to build software that can accommodate change. That is the demanded nature of software. Software must be supportable – understandable, maintainable and scalable. This is what makes software different from a bridge and makes software engineering different from traditional engineering.

Each software system is unique and its production process is unique. Unlike in traditional engineering disciplines, an application software product is not manufactured – it is implemented. It is not a car or refrigerator. It must be implemented to fit its environment. Each instance of a software system is unique - whether built from scratch or customized from a Commercial Of The Shelf (COTS) software package. Only system software and software tools, such as operating systems and word processors, are massively manufactured once engineered. Application software, which is the subject of this book, is implemented not manufactured.

1.1.4 Software Engineering Is More Than Programming

In introductory pages to Part A of this book, software engineering has been defined as “…the field of computer science that deals with the building of software systems that are so large or so complex that they
1. SOFTWARE DEVELOPMENT LIFECYCLE

are built by a team or teams of engineers.” (Ghezzi et al., 2003, p.1). The definition emphasizes teams of people and complex systems.

Programming refers to “code cutting” — writing a serious of instructions to make a computer perform a particular task. If the task is large, programming may involve a team of programmers but each act of programming is primarily a personal activity. Programming is a skill. Given a definition and specification for the problem, a programmer applies the skills to express the problem in a programming language.

Software engineering is more than programming. Software engineering applies to complex problems that cannot be solved by programming alone. Complex systems must be designed before they can be programmed. Like in the building industry, a complex system must be architected before it can be built. It must be modularized using abstraction and “divide-and-conquer” methods. Each module must be then carefully specified and its interfaces to other modules defined, before it can be given to programmers for coding.

A programmer has a limited understanding of the entire system. S/he codes one programming module at a time — a software component that needs to be integrated (by a software engineer) with other components to configure a working system. (Of course, this distinction between programmer and software engineer is only to illustrate the issue. In practice, the distinction may or may not apply.)

Frequently, multiple versions of the same component are available to a software engineer. A software configuration is built by assembling specific versions of different components. It is, therefore, possible to have multiple configurations of the same system.

Before a system can be designed, a software engineer must understand its requirements. This means that the requirements analysis must be done and specified in some modeling language. A standard modeling language in contemporary practice is Unified Modeling Language (UML). Both analysis and design models are expressed in UML.

A software engineer is responsible for bringing UML models for a system to the point where the initial programming code can be generated from these models. Programmers can take over from that point but the software engineer remains in charge of roundtrip engineering between the design and the code. Roundtrip engineering is an iterative cycle of forward (from design to code) and reverse (from code to design) engineering activities.

Finally, software engineering is a team activity. Teams must be managed. Consequently, software engineering is affected by and contributes to project management. This includes planning, budgeting and scheduling, quality control and risk management, and change and configuration management.

To recap, software engineering is concerned with providing an architectural solution for the system, with designing architectural components, with integrating components into an operational system, with roundtrip engineering, with project management, etc. Software engineering is an elaborate knowledge within which programming is a useful skill.

1.1.5 Software Engineering Is About Modeling

Software engineering is about modeling (Lee and Tepfenhart, 2002). Models are abstractions from reality. They are abstract representations of reality. Is computer program a model or reality? Well, one can argue that a program stored in computer memory or printed on paper is a reality. But the purpose of
programming is not the program code per se; it is rather the functionality it provides. Is a footbal game played on a computer a reality? Is it really for real? These are clearly rhetorical questions.

**Abstraction** is a powerful technique in software engineering. By allowing to concentrate on important aspects of a problem and by ignoring aspects that are currently not relevant, abstraction allows to systematically conquer the problem’s complexity (Section 1.1.6). Abstraction applies to both software products and to software processes. A **software process model** is an abstract representation of the software process. In practical terms, a software process model defines the lifecycle phases and how they interact (Section 1.2). A **software product model** is an abstract representation of a software product. A software product model identifies a discrete product in a discrete stage of the lifecycle.

A software process model determines what software products, at various levels of abstraction, need to be produced by lifecycle phases. The following is a list of generic software product models:

- **Requirements model** (a relatively informal model that captures user requirements and describes the system in terms of its business value).
- **Specifications model** (a model that specifies requirements in more formal terms using a modeling language such as UML).
- **Architectural model** (a model that defines the desired architecture of the system).
- **Detailed design model** (a model that defines the software/hardware details of the demanded programming solution).
- **Program model** (an implementation model that constitutes an ultimate executable software model).

Each of these product models can be further divided to identify more specific models. For example, a detailed design model can include user interface model, database model, program logic model, etc.

Finally, the software engineering paradigm used in system development influences the modeling abstractions. The two main paradigms are the old-style functional (procedural, imperative, structured) development and the modern object-oriented development.

The **functional paradigm** breaks a complex system down to manageable units using the approach known as functional decomposition. The technique called data flow modeling is used for this purpose. The software model is successively divided into processes (at decreasing levels of abstraction) linked by data flows.

The **object-oriented paradigm** breaks a system down into packages/components of classes linked by various relationships. Abstraction can be applied by allowing nested structures, i.e. a package/component can contain multiple levels of other packages/components. This book concentrates on object-oriented software engineering.

### 1.1.6  Software System Is Complex

Software systems are **complex**. In the past, software was monolithic and procedural in nature. A typical Cobol program of the past was a single entity with subroutines called as required. The logic of the program was sequential and predictable. The complexity of such software was a consequence of its mere size.
Modern object-oriented software is distributed (it can reside on many computer nodes) and its execution is random and unpredictable. The size of modern software is the sum of the sizes of its components. Each component is designed to be of limited manageable size. As a result, the size is not the main factor in the complexity of modern software.

The complexity of modern software is in the “wires” – in the linkages and communication paths between components. The inter-component linkages create dependencies between distributed components that may be difficult to understand and manage. The difficulty is inflated by the fact that components are frequently developed and managed by people and teams not even known to each other.

Figure 1-4 shows a possible object-oriented system in which objects in various packages communicate indiscriminately. This creates a network of intercommunicating objects. In the diagram, the complexity within packages (components) is still manageable due to limited size of packages. However, the dependencies created by inter-package communication links will grow exponentially with the introduction of more packages. The responsibilities of managing such dependencies are not always clear as the responsibilities for packages remain with different teams. More importantly, the fact that any object in one package can communicate with any object in another package creates potential dependencies between all objects in the system. This means that a change in an object can potentially impact (can have a “ripple effect”) on any other object in the system.

More formally, the cumulative measure of object dependencies with unrestricted inter-package (inter-component) communication links is the number of different combinations of pairs of objects. Such measure can be computed using a probability theory method known as the combinations counting rule. The formula below calculates cumulative class dependency (CCD) in a system with \( n \) classes of objects. For 5 classes, CCD equals 10. For 57 classes (for example), CCD equals 1596. Such growth in complexity quickly becomes unsupportable.

\[
n \text{CCD}_n = \frac{n!}{2!(n - 2)!}
\]
The formula computes the worst complexity, where each object communicates with all other objects. Although the worst scenario is unlikely in practice, it must be assumed in any dependency impact analysis conducted on the system (simply because real dependencies are not known beforehand). If a change in a class can potentially impact on any other class, then this fact must be checked to ensure that the change has been conquered. Systems permitting an indiscriminate network of intercommunicating objects, like in Figure 1-4, are considered unsupported from software engineering perspective. They are not understandable, not maintainable, and not scalable.

The solution to the dilemma lies in replacing networks of objects with hierarchies of objects. All complex systems that are supportable take the form of a hierarchy. This topic is so important that a separate chapter is dedicated to it (Chapter 9). Figure 1-5 shows merely how the complexity of a system can be reduced by allowing only single channels of communication between packages. Each package defines an interface object (this could be a Java-style interface or so called dominant class) through which all communication with the package is channeled. Despite the introduction of three extra objects, the complexity of the system in Figure 1-5 is visibly reduced in comparison with the same system in Figure 1-4.

Note also that the design in Figure 1-4 makes a mockery of a principal tool in the hands of the software engineer – the abstraction mechanism. Abstraction allows to reason about selected parts of the system in a way that suppresses (abstracts from) irrelevant details. Although objects in Figure 1-4 are grouped in three packages, the complexity of the system (measured as a cumulative class dependency) is the same as for the similar system with no packages at all. The packages in Figure 1-4 do not introduce any useful layer of abstraction. They may well not exist.

1.2 Lifecycle Phases

Software lifecycle is an abstract representation of a software process. It defines the phases, steps, activities, methods, tools, as well as expected deliverables, of a software development project. It defines a software development strategy.
There exist a number of useful lifecycle models (Section 1.3), which are in relative agreement on lifecycle phases but differ on the importance of particular phases and on interactions between them. Lifecycle phases assumed in this book were identified at the beginning of this chapter. They are:

1. requirements analysis
2. system design
3. implementation
4. integration and deployment
5. operation and maintenance

### 1.2.1 Requirements Analysis

"**User requirements** are statements, in a natural language plus diagrams, of what services the system is expected to provide and the constraints under which it must operate" (Sommerville, 2001, p.98).

**Requirements analysis** are the activities of determining and specifying requirements. In contemporary practice, requirements analysis is assisted by a good degree of engineering rigor, and it is therefore sometimes identified with **requirements engineering**.

**Requirements determination** proves to be one of the greatest challenges of any software development lifecycle. Users are frequently unclear about what they require from the system. Frequently, they do not know real requirements, exaggerate them, provide requirements that conflict with requirements of fellow users, etc. There is also a risk, as in any communication between people, that the true meaning of a requirement is misunderstood. Developers are faced with the following anonymous observation: “I know you believe you understood what you think I said, but I’m not sure you realize that what you heard is not what I meant.”

There are many methods and techniques of eliciting requirements. They include (Maciaszek, 2001):

- interviewing users and domain experts,
- questionnaires to users,
- observation of users performing their tasks,
- study of existing system documents,
- study of similar software systems to learn about the domain knowledge,
- prototyping of working models of the solution to discover and confirm requirements,
- joint application development sessions between developers and customers.

**Requirements specification** follows requirements determination. In current practice, Unified Modeling Language (UML) is the standard modeling language for requirements specification (as well as for system design). Requirements are specified in graphical models as well as in textual descriptions. Because a complex system cannot be understood from a single viewpoint, the models provide complementing and by necessity overlapping viewpoints on the system.

Both graphics and text are stored in a repository of a **Computer Assisted Software Engineering (CASE)** tool. The tool facilitates making changes to the models as needed. It enables the integration of various models with overlapping concepts. It also enables transformations between analysis models (where this is possible) and assists in transformations to design models.
1. SOFTWARE DEVELOPMENT LIFECYCLE

Requirements analysis results in a requirements document (Maciaszek, 2001). Most organizations adopt some templates for requirements documents. A template defines the structure of the document and provides guidelines how to write it. The main body of a requirements document contains models and descriptions for system services and for system constraints. System services (what the system does) are frequently classified into functional requirements and data requirements. System constraints (how the system is constrained) include considerations related to the user interface, performance, security, operational conditions, political and legal restrictions, etc.

As mentioned in passing, the outcome of each lifecycle phase should be validated and tested. A professional approach to testing demands the creation of a Software Quality Assurance (SQA) group within the organization. The SQA team consists of professional system testers. The team is relatively independent from developers. To make the whole process work, the SQA team, not the developers, is made responsible for the quality of the software product (i.e. the SQA is blamed for any undetected inadequacies and defects in the software).

Testing of abstract models is difficult because, most of the time, it cannot be automated. Walkthroughs and inspections are two popular and effective techniques. The techniques are similar. These are pre-organized meetings of developers and users during which the requirements models and documents are “walked through”. The discussion that follows in the meetings is likely to uncover some problems. The essence of these techniques is that the problems are identified, but not attempted to solve during the meetings, and there is no finger-pointing at people potentially responsible for the problems.

1.2.2 System Design

“A software design is a description of the structure of the software to be implemented, the data which is part of the system, the interfaces between system components and, sometimes, the algorithms used.” (Sommerville, 2001, p.56). This definition is consistent with the definition of a software system as a union of data structures and algorithms. In enterprise information systems, data structures imply databases. Algorithms are not always fully described during design in order leave some level of implementation freedom to the programmers (and, to put it bluntly, designers are not programmers and they may not be in position to take knowledgeable algorithmic decisions).

Design begins where the analysis ends. As true and trivial as this statement is, the separation line between analysis and design is not that clear in many projects. Theoretically the issue is simple. Analysis is the modeling unconstrained by any implementational (hardware/software) considerations. Design is the modeling that takes into consideration the platform on which the system is to be implemented.

In practice the distinction between analysis and design is blurred. There are two main reasons for this. Firstly, modern lifecycle models are iterative with increments (Section 1.3.2). In most such models, at any point in time, there are multiple incremental versions of the software under development. Some of the versions are in analysis, others in design; some in development, others in production, etc. Secondly, and more significantly, the same modeling language (UML) is used for analysis and design. The movement from analysis to design is “by elaboration”, rather than by translation between different representations. The analysis model is elaborated into the design model by smooth addition of specification details. Drawing the line between analysis and design is very difficult in these circumstances.

The design discussed above is called, more precisely, the detailed design, i.e. the design that adds details to the analysis models. But there is another aspect of system design, namely the architectural design. The
**architectural design** is concerned with setting up an architectural framework for the system, the component structure that the detailed design must adhere to, and the principles and patterns of internal communications between components.

The architectural design decides about the “beauty” of the system. A major objective of architectural design is to lead to a system that is supportable – understandable, maintainable, and scalable. The detailed design must conform to architectural design. Because of the blurry separation line between analysis and detailed design, some early architectural decisions may need to be taken within or even prior to requirements specifications (but after requirements determination).

**Testing** of architectural design is two-faceted. Firstly, the superiority of an architectural framework offered to the designers must be demonstrated. It must be shown that the framework addresses software complexity, ensures supportability, streamlines development, etc. Secondly, testing of architectural design has to do with verifying if the design of components conforms to principles and patterns of the adopted architectural framework.

Testing of detailed design also has two aspects. Firstly, to be testable, the detailed design must be traceable. **Traceability management** is the whole branch of software engineering concerned with maintaining links between software artifacts at various stages of development. In the case of detailed design, each design artifact must be linked to requirements in the requirements document that motivated the production of that artifact. The existence of an artifact does not mean that it is acceptable. Hence, the second aspect of design testing employs walkthrough and inspections to assess the quality of the design product so far.

### 1.2.3 Implementation

**Implementation** is mostly programming. But programming does not just imply a bunch of people sitting in a common area and coding in some programming language from design specifications. Programming is much more intellectual than that. As indicated in the previous sections, the designs will be “underspecified” in some areas when they reach programmers, in particular in the area of algorithm design. Completing the specifications requires extra designing before coding can take place. In this sense, a programmer is also a designer.

A programmer is a *component engineer*. Today’s programming is rarely done from scratch. Much of programming is based on re-using preexisting components. This means that the programmer must have the knowledge of component software and must know how to find this software in order to plug it the newly coded application components. This is a tough ask.

A programmer is a *roundtrip engineer*. Programming begins as a transformation of design into code. Initial code does not have to be manually programmed. Using CASE tools and **Integrated Development Environments** (IDEs), the code can be generated (forward-engineered) from design models. Once generated, the code must be programmed manually to fill in the missing bits (these “bits” are substantial and most difficult bits to program). Once modified by the programmer, the code can be reverse-engineered back to update the design models. These forward- and reverse-engineering cycles are called **roundtrip engineering**.

If all this sounds simple, it is not. Roundtrip engineering is not perfect. The existing tools are powerful and clever, but they still fail in properly doing some aspects of the job. To keep the design and
implementation in sync, both the designers and programmers must know the limitations and make manual corrections and additions as needed. Much of the responsibility in this task rests on project managers. They must schedule and monitor the work of designers and programmers so that they do not step on each other toes. The practical requirement is that forward-engineering and reverse-engineering cannot overlap in time (Maciaszek, 2002).

In many projects, implementation is the longest of the development phases. In some lifecycle models, such as in the agile software development (Section 1.3.2.4), implementation is a dominant development phase. Implementation is an error-prone activity. The creative time to write programs may be less than the time spent on program debugging and testing.

**Debugging** is an act of removing software “bugs” – errors in programs. Errors in program syntax and some logic errors can be spotted and rectified by commercial debugging tools. Other errors and defects need to be discovered during program tests. **Testing** can take the form of code reviews (walkthroughs and inspection) or it can be execution-based (observing the program behavior during its execution). Traceability management supports testing in establishing if the programs satisfy user requirements.

There are two kinds of **execution-based testing**: testing to specs (black-box testing) and testing to code (white-box testing). Both kinds use the same strategy of feeding the program with input data and observing if the output is as expected. The difference lies in that testing to specs feeds the program with data without any consideration given to the program logic. The assumption is that the program should be behaving reasonably for any input data. In testing to code, the input data is provided to test specific execution paths in the program; as many of them and as diverse as possible. Because testing to specs and testing to code tend to uncover different kinds or errors and defects, both of them should be used.

### 1.2.4 Integration and Deployment

“The whole is more than the sum of the parts”. This seminal truth by Aristotle(384-322 B.C.) captures the essence of system integration and deployment. **Integration** assembles the application from the set of components previously implemented and tested. **Deployment** is the handing over of the system to customers for production use.

Software integration signifies the movement from the “programming in the small” to the “programming in the large” (Ghezzi et al., 2003). Enterprise information systems are all sufficiently large and complex systems (Section 1.1.6) to make integration a very significant phase in the lifecycle. As another saying goes: “For every complex problem there is a simple solution - that won't work” (H.L. Mencken). Integration cannot be swept under the carpet. Integration is a stage in its own right, even if it is sometimes difficult to disassociate it from implementation, such as in the continuous integration of the agile development (Section 1.3.2.4).

Integration is also difficult to disassociate from testing. In fact, the integration phase of the lifecycle is frequently referred to and discussed under the term of **integration testing**. With the omnipresent acceptance of iterative lifecycle models (Section 1.3.2), software is produced as a sequence of frequent incremental releases. Each increment is an integration of components, individually tested before, but in need of integration testing prior to deployment.

To a large degree, integration is driven by the architectural design of the system. The architecture of the system identifies its components and dependencies between them. It is of paramount importance that the
architectural solution is in the form of a hierarchy (Section 1.1.6). A hierarchy means elimination of any circular dependencies between components. With circular dependencies, integration testing of some software increments (builds) may be impossible.

Consider a component dependency structure where component $C_i$ uses (depends on) $C_j$ and $C_j$ uses $C_k$. Suppose that $C_i$ and $C_j$ have been implemented and individually tested but $C_k$ is yet to be developed. The task is to integrate $C_i$ and $C_j$. This task requires to program a stub for $C_k$, i.e. a piece of code that simulates the behavior of the missing component $C_k$. The stub provides an integration context for the execution of $C_j$. A usual way to implement the stub is by allowing it to take the same input/output parameters as in the ultimate $C_k$ and to produce the results expected by $C_j$ by hard-coding them or reading from a file.

All this works provided there are no circular dependencies between $C_j$ and $C_k$. In the presence of circular dependencies both components must be fully implemented and individually tested prior to integration. Even then, the circular dependencies will create a testing nightmare. With extensive cycles in architectural design, integration testing would have to be done as a single activity called a big-bang testing. Big-bang testing can only be successfully performed for small programming solutions. It is not a reasonable proposition in modern software development.

Coming back to the example, suppose now that $C_j$ and $C_k$ have been implemented and individually tested but $C_i$ is yet to be developed. How do we integrate $C_j$ and $C_k$ so that the integrated increment (build) gets the data and other context that would be normally given to it by $C_i$ and so that we can assert that the increment does the job expected by $C_i$? To do this, a driver for $C_i$ must be programmed to “drive” the integration.

All in all, integration requires writing of extra software, stubs and drivers, which are useful only during integration. This supporting test software is called test harness or test scaffolding, to refer to temporary scaffolding used in the building industry.

Integration can be conducted top-down (from the root of the dependency hierarchy) or bottom-up (from the components at the leaves of the hierarchy). The top-down approach requires the implementation of stubs. The bottom-up approach demands drivers. In reality, the integration rarely follows exclusively one of this approaches. A mixed approach, called sometimes middle-out, prevails.

Like integration, deployment is not once-off activity. Software is deployed in releases. Each release combines a number of increments (builds) that offer coherent and usable functionality to the users. Prior to deployment, software is system tested by developers under realistic conditions. This is sometimes called alpha testing. Alpha-testing is followed by acceptance testing by user-testers. This is sometimes called beta-testing (alpha and beta testing are terms accepted more in tests conducted on system software on behalf of software vendors, as opposed to the application software development).

Apart from system and acceptance testing, deployment includes a range of other activities. The most important of these activities is user training. In practice, user training may start well before the system is ready for release. The training coincides with the production of user documentation.
1.2.5 Operation and Maintenance

Operation signifies the lifecycle phase when the software product is used in day-to-day operations and the previous system (manual or automated) is phased out. Phasing out is usually a staged process. In situations where this is possible and feasible, the organization runs the new and old system in parallel for some time. This provides a fallback possibility if the new product fails the business demands.

Operation coincides with the start of product maintenance. In software engineering, maintenance has a slightly different meaning to the normal use of this word. Firstly, maintenance is not just an unplanned fixing of arising problems. Maintenance is planned and costed in the early stages of the lifecycle. Secondly, maintenance includes product evolution. In some iterative lifecycle models (Section 1.3.2), it may be even difficult to distinguish between development and maintenance.

In literature (Maciaszek, 1990; Ghezzi et al., 2003), maintenance is usually divided into:

- **corrective** (housekeeping) – fixing defects and errors discovered in operation,
- **adaptive** – modifying the software in response to changes in the computing or business environment,
- **perfective** – evolving the product by adding new features or improving its quality.

Maintenance cost is considerable in the life of software. It is considerable because the product remains in operation for a long time. Large enterprise systems are so fundamental to organizations that they are kept alive using any available “life support” technologies. Such systems are called legacy systems. They should be retired but there is no replacement for them. Indeed, when the decision of system retirement is eventually made, it is not made because the legacy system is not useful to the organization any more. It is because it is not technically possible to keep it alive much longer. It should not come as a surprise that maintenance is the main reason why this is not technically possible – over time maintenance destroys the architectural clarity of software to the point that it cannot be supported any more.

1.3 Lifecycle Models

Software lifecycle determines the “what”, but not the “how”, of software engineering. Software engineering is to a considerable degree a social phenomenon constrained by a particular organizational culture of an enterprise. An enterprise may elect a generic lifecycle model but the specifics of the lifecycle, how the work is done, is unique for each organization and may even differ considerably from project to project. This is consistent with the observations, made in Section 1.1.3, that a software product is not manufactured – it is implemented. Software process is not an experiment that can be repeated over and over again with the same degree of success.

The following is a short list of reasons why lifecycle specifics must be tailored to organizational cultures and why they differ from project to project:

- software engineering experience, skills and knowledge of the development team (if not sufficient, the time for the “learning curve” must be included in the development process),
- business experience and knowledge (this is much more troublesome than the previous point because business experience and knowledge is not acquired easily),
- kind of application domain (different processes are needed to develop an accounting system and a power station monitoring system),
1. SOFTWARE DEVELOPMENT LIFECYCLE

- business environment changes (changes in external political, economic, social, technological, and competitive factors),
- internal business changes (changes to management, working conditions, enterprise financial health, etc.),
- project size (a large project demands different processes than a small one; a very small project may even not need any processes as the developers can cooperate and exchange information informally).

Under the proviso of the above discussion, the approaches to software lifecycle can be broadly divided into two main groups:

- waterfall with feedback
- iterative with increments

1.3.1 Waterfall Lifecycle With Feedback

The waterfall model is a traditional lifecycle introduced and popularized in the 1970s. The model has been reported as used with great success on many large projects of the past. Most of these projects were for batch (i.e. not interactive) systems implemented in Cobol language. Today, the waterfall lifecycle is used less frequently.

Using the lifecycle phases assumed in this chapter, the waterfall lifecycle with feedback can be visualized as in Figure 1-6. It is a linear sequence of phases, in which the previous phase must be completed before the next one can begin. The completion of each phase is marked with signing off of a project document for that phase. Feedbacks (back arrows in Figure 1-6) between phases are possible, and indeed likely. A feedback signifies an undocumented but necessary change in a later phase, which should result in a corresponding change in the previous phase. Such backtracking should, but rarely does, continue to the initial phase of Requirements Analysis.

![Figure 1-6: Waterfall lifecycle with feedback](image)

There are many variants of the waterfall lifecycle model. One such variant, depicted in Figure 1-7, allows for overlaps between phases, i.e. the next phase can begin before the previous one is fully finished, documented and signed off. This is indicated in Figure 1-7 by arrowed circles between phases. Another popular variant, called sometimes the prototyping model, allows for construction of software prototypes in phases preceding the implementation phase. This is also indicated in Figure 1-7.
A crucial point about the waterfall approaches is that they are **monolithic** – they are applied “in one go” to the whole system under development and they aim at a single delivery date for the system. The user is involved only in early stages of requirements analysis and signs off the requirements specification document. Later in the lifecycle, the user is in the dark until the product can be user-tested prior to deployment. Because the **time lag** between project commencement and software delivery can be significant (in months or even years), the trust between users and developers is put to the test and the developers find it increasingly difficult to defend the project to the management and justify expended resources.

The introduction of **overlaps** between phases can address another drawback of the lifecycle, namely stoppages in some parts of the project because developers from various teams wait for other teams to complete dependent tasks. The overlaps allow also for greater feedback between neighboring phases.

**Prototyping** in any lifecycle has a useful purpose, but it does offer special advantages to the waterfall model by introducing some flexibility to its monolithic structure and by cushioning against the risk of delivering product not meeting user requirements. A **prototype** is a partial “quick and dirty” example solution to the problem. From that example solution, successive forms of the software product can be developed.

In software engineering, prototyping has been used with a great deal of success to elicit and clarify user requirements for the product. Once used in this capacity, the question arises what to do with the prototype software. One possibility is to throw it away once its requirements validation purpose has been achieved. The justification for “throw-away” prototyping is that retaining the prototype can introduce “quick and dirty” solutions into the final product. After all, people don’t drive in prototype cars.

However, software implementation is not car manufacturing. With component-based dynamic software development environments, the conversion of a prototype to a good final product without any traces of “quick and dirty” solutions is achievable. This is depicted with forward arrows in Figure 1-7. A prototype created in requirements analysis can serve to refine system design and an improved design prototype can be refined into an implemented final product.

Table 1-1 classifies the main characteristics of waterfall lifecycles into advantages and disadvantages (Ghezzi et al., 2002; Schach, 2002). Sometimes, a characteristic is both advantage and disadvantage. This is shown by comments placed in both cells of a table row.
Table 1-1

<table>
<thead>
<tr>
<th>Waterfall lifecycle with feedback</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Advantages</strong></td>
<td><strong>Disadvantages</strong></td>
</tr>
<tr>
<td>• Enforces disciplined approach to software development. Defines clear milestones in lifecycle phases, thus facilitating project management.</td>
<td>• Completion criteria for requirements analysis and for system design are frequently undefined or vague. Difficult to know when to stop. Danger to deadlines.</td>
</tr>
<tr>
<td></td>
<td>• A monolithic approach, applying to the whole system, that may take a very long time to final product. This may be outright unacceptable for a modern enterprise demanding short “return on investment” cycles.</td>
</tr>
<tr>
<td></td>
<td>• No scope for abstraction. No possibility to “divide-and-conquer” the problem domain to handle the system complexity.</td>
</tr>
<tr>
<td>• Produces complete documentation for the system.</td>
<td>• Documentation can give a false sense of confidence about the project progress. Its dry inanimate statements can be easily misinterpreted. Also, there is a risk of bureaucratising the work.</td>
</tr>
<tr>
<td>• Signing off the project documents before moving to successive phases clarifies the legal position of development teams.</td>
<td>• Freezing the results of each phase goes against software engineering as a social process, in which requirements change whether we like it or not.</td>
</tr>
<tr>
<td>• Requires careful project planning.</td>
<td>• Project planning is conducted in early stages of the lifecycle when only limited insight into the project is available. Risk of misestimating of required resources.</td>
</tr>
</tbody>
</table>

1.3.2 Iterative Lifecycle With Increments

**Iteration** in software development (as contrasted with iteration in a program) is a repetition of some process with an objective to enrich the software product. Every lifecycle has some elements of an iterative approach. For example, feedbacks and overlaps in the waterfall model introduce a kind of iteration between phases, stages or activities. However, the waterfall model cannot be considered iterative because an iteration means movement from one version of the product to the next version of it. The waterfall approach is monolithic with only one final version of the product.

An **iterative lifecycle** assumes **increments** – an improved or extended version of the product at the end of each iteration. Hence, the iterative lifecycle model is sometimes called evolutionary or incremental.

An iterative lifecycle assumes **builds** – executable code that is a deliverable of an iteration. A build is a vertical slice of the system. It is not a subsystem. The scope of a build is the whole system, but with some
functionality suppressed, with simplified user interfaces, with limited multi-user support, inferior performance, and similar restrictions. A build is something that can be demonstrated to the user as a version of the system, on its way to the final product. Each build is in fact an increment over the previous build. In this sense the notions of build and increment are not different.

An iterative lifecycle assumes short iterations between increments, in weeks or days, not months. This permits continual planning and reliable management. The work done on previous iterations can be measured and can provide valuable metrics for project planning. Having a binary deliverable at the end of each iteration, that actually works, contributes to reliable management practices.

Figure 1-8 demonstrates that each iteration is a small waterfall of typical lifecycle phases. The differences are in the “small print”, as discussed above. The uppermost loop from Operation and Maintenance to the user is now fully and frequently employed. Lessons of previous iteration are taken an immediate advantage in the next iteration. The user, equipped with the new experience of using the previous build, can be very helpful in refining requirements for the next iteration. The current design of the build is a starting point for System Design in the next iteration. Deployment of Iteration 2 Product is a start to Iteration 3, etc.

A classic iterative lifecycle model is the spiral model (Boehm, 1988). A modern representative of the iterative lifecycle is the IBM Rational Unified Process® (RUP®) (RUP, 2003), which originated from Rational Unified Process (RUP) (Kruchten, 1999). More recent representatives of the iterative lifecycle model are Model Driven Architecture (MDA®) (Kleppe et al., 2003; MDA, 2003) and the agile development (Agile, 2003; Martin, 2003).
1. SOFTWARE DEVELOPMENT LIFECYCLE

1.3.2.1 Spiral Model

As rightly noticed by Ghezzi et al. (2002), the spiral model (Boehm, 1988) is really a metamodel in which all other lifecycle models can be contained. The model consists of four quadrants of the Cartesian diagram (Figure 1.9). The quadrants are: planning, risk analysis, engineering, and customer evaluation. The first loop around the spiral starts in the planning quadrant and is concerned with initial requirements gathering and project planning. The project then enters the risk analysis quadrant, which conducts cost/benefit and threats/opportunities analyses in order to take a “go, no-go” decision of whether to enter the engineering quadrant (or kill the project as too risky). The engineering quadrant is where the software development happens. The result of this development (a build, prototype or even a final product) is subjected to customer evaluation and the second loop around the spiral begins.

The spiral model treats the software development per se as just one of the four quadrants – the engineering quadrant. The emphasis on repetitive project planning and customer evaluation gives it a highly iterative character. Risk analysis is unique to the spiral model. Frequent risk analyses allow an early identification of any emerging risks to the project. The risks can be internal (under the organization’s control) or external (risks that the organization cannot control). Either way, the task of risk analysis is to look in the future and if it is clear that the risks are not sustainable, the project must be killed without regard to the costs expended so far.

Each loop fragment within the engineering quadrant can be an iteration resulting in a build. Any successive loop fragment in the outward direction is then an increment. This said, other interpretations of
engineering loop fragments are possible. For example, the whole loop around the spiral may be concerned with requirements analysis. In such case, the engineering loop fragment may be concerned with requirements modeling and building an early prototype to solicit requirements. Alternatively, the spiral model can contain the monolithic waterfall model. In such case, there will be only one loop around the spiral.

The spiral model is rather a reference model or a metamodel for other models. Attractive and realistic as it may be, it does not translate to a specific documented lifecycle that organizations use in software development. The spiral model is a “way of thinking” about software development principles.

### 1.3.2.2 Rational Unified Process (RUP)

The *IBM Rational Unified Process® (RUP®)* (RUP, 2003) is more than a lifecycle model. It is also a support environment (called the RUP platform) to assist developers in using and conforming to the RUP lifecycle. This support comes in the form of HTML and other documentation providing online help, templates and guidance. The RUP support environment constitutes an integral part of IBM (previously Rational) suite of software engineering tools, but it can be used as a lifecycle model for any software development.

The level of RUP acceptance has suffered from an obscure RUP process structure, which has been presented as two-dimensional. The *horizontal dimension* represents the dynamic aspect of the lifecycle – time passing in the process. This dimension has been divided into four unfolding lifecycle aspects: inception, elaboration, construction and transition. The *vertical dimension* represents the static aspect of the lifecycle – software development disciplines. This dimension has been divided into lifecycle activities: business modeling, requirements, analysis and design, implementation, test, deployment, as well as supporting activities of configuration and change management, project management, and environment.

Although appealing at first, the process explained as the combination of the two RUP dimensions lacks clarity. What is the difference between construction and implementation or between inception and business modeling or between transition and deployment? These questions are difficult to answer. To rectify these problems (it seems), a simplified visualization of the RUP® process has been offered. Figure 1-10 is this simplified representation (RUP, 2003)
Apart from some small variations in lifecycle phases, the RUP process aligns very well with the generic iterative lifecycle in Figure 1-8. The main difference is an explicit Test phase after Implementation. As mentioned few times in passing, the lifecycle phases adopted in this book consider testing as an all-encompassing activity to be applied to each lifecycle phase, not just to implementation. In RUP, testing results in the Deployment of an iteration (increment).

Following the spiral model, RUP tries to be explicit about risk management. One aspect of risk management in RUP is the explicit Evaluation phase. More importantly though, and consistently with the spiral model, RUP advocates that a “go, no-go” decision should be taken at the end of each lifecycle phase.

### 1.3.2.3 Model Driven Architecture (MDA)

The Model Driven Architecture® (MDA®) (MDA, 2003; Kleppe et al., 2003) is a lifecycle framework from Object Management Group (OMG). MDA attempts to take the Unified Modeling Language (UML) to its next natural stage – executable specifications. The idea of executable specifications, i.e. turning specifications models into executable code, is not new, but MDA takes advantage of existing standards and modern technology to make the idea happen.

The MDA is a modern representative of the transformation model (Ghezzi et al., 2003), which in turn has its origins in formal systems development (Sommerville, 2001). The transformation model treats systems development as a sequence of transformations from the formal specifications for the problem, via more detailed (less abstract) design stages, to an executable program. Each transformation step is carefully verified to guarantee that each output is a true representation of the input.

The transformation model assumes extensive automatization of transformations, but recognizes that some transformations can only be performed manually. Accordingly, the machine-readable representations of models, created and stored within Computer Assisted Software Engineering (CASE) environments, are assumed. Machine-readability supports iterative nature of the transformation model. Unlike in the waterfall lifecycle, the transformation model supports evolution of models through multiple iterations.
The MDA lifecycle is depicted in Figure 1-11. MDA distinguishes between informal Requirements and more formal Analysis specifications. This separation allows to exclude Requirements from the transformations of the MDA process. The remaining artifacts (models) of the process are in the machine-readable form susceptible to transformations.

The result of Analysis is a Platform Independent Model (PIM), highly abstract and detached from any software/hardware constraints. The result of Design is a Platform Specific Model (PSM), less abstract and constrained by the implementation software/hardware platform. For each platform, a separate PSM is generated. If the system under development spans multiple platforms, then additional PSMs are provided and linked by MDA interoperability bridges. Multiple PSMs transform to multiple codes, also linked by bridges.

The MDA promise extends into the component technology and the whole area of constructing systems from reusable building blocks – models and programs (this is called sometimes the component-based lifecycle (Pressman, 2001)). The architecture goes beyond executable UML models and encompasses a range of OMG-specified services, such as repository services, persistence, transactions, event handling, and security. The aim is to create reusable and transformable models for specific vertical industries, such as telecommunications or hospitals. Time will show if MDA will prove itself in practice. For this to happen, MDA must be able to demonstrate that it scales up to large and complex systems.
1.3.2.4 Agile Lifecycle With Short Cycles

The agile software development process, proposed in 2001 by Agile Alliance — a non-profit organization of enthusiasts, is a daring new approach to software production. In the Manifesto of Agile Alliance (Agile, 2003), the spirit of the agile development is captured in four recommendations:

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

The agile development stresses that software production is a creative activity that depends on people and team collaboration far more than on processes, tools, documentation, planning and other formalities. Agile development subscribes to the maxim that “great software comes from great people”, everything else is secondary. “People” includes all project stakeholders – developers and customers. Unlike in other software processes, in agile development, customers (users and system owners) work closely with the development team throughout the lifecycle, not just at the beginning of it. Constant customer feedback alleviates the need for signing a formal contract for the entire product. Intense collaboration alleviates the need for these parts of documentation that serve the purpose of knowledge transfer.

Despite all these “revolutionary” propositions, agile development sits well among other iterative lifecycles. As shown in Figure 1-12, an agile lifecycle may not be using the terminology of typical lifecycle phases, but it does in effect follow the normal cycle of analysis, design, implementation, and deployment.

![Figure 1-12 Agile software development](image-url)
“Conventional” requirements analysis is replaced in agile development by user stories – features that the customer would like the system to support. The development team estimates how long it will take to implement each story, and how much each story will cost. The customer can then select stories to be implemented in the first and successive iterations.

“Conventional” system design and implementation are replaced in agile development by a combination of acceptance tests, refactoring, and test-driven development. Acceptance tests are programs that an application program must pass to be accepted by customers. This process is called test-driven development or intentional programming – the developer programs his/her intent in an acceptance test before s/he implements it. The whole approach is facilitated by frequent refactorings – improvements to the structure of the system without changing its behavior.

Agile development encourages other practices, such as pair programming and collective ownership. All programming is done by pairs of programmers – two programmers working together at a single workstation. One programmer writes the code and the other observes the progress and raises questions. The roles change whenever one of the programmer wants to “prove the point by driving the keyboard”. The programming pairs also change at least once during a day. The collective ownership is an outcome. No individual owns the code. The assumption is that a high spirit in the team and desire to ship the product will outweigh any demand for individual responsibility.

“Conventional” integration and deployment is replaced in agile development by continuous integration and short cycles. Programming pairs can check-in their code and integrate with the rest of the code at will. Moreover, more than one programming pair can check-out and work on the same piece of code. This means a possibility of conflict when a team which wants to check-in and merge their code discovers that another team has done the merge before with the conflicting code. Such conflicts must be negotiated between the teams involved.

Agile development does not mean lack of planning. In fact, the deployment dates are carefully planned. Each iteration is normally planned to complete in short cycles of two-week duration. The product at the end of two-week cycle is a minor delivery for customer evaluation. A major delivery, a product put into production, is a result of about six two-week cycles.

Agile development differs more in practices than in approach to iterative development. Its main representative is eXtreme Programming (XP) (Beck, 1999). As with the MDA approach, the future will show if agile development can scale up to large and complex systems. The main danger facing adopters of the agile lifecycle is the risk of ending up with the discredited build-and-fix model (Schach, 2002), in which the software is hack-coded with no specified requirements or design.

Summary

1. Software engineering is concerned with development of large software systems. Software engineering is typically the central activity of a more generic notion of system engineering.

2. The software engineering pentagon consists of software development lifecycle, software modeling language, software engineering tools, software project plan, and software process management.
3. The stages of software development process are referred to as software lifecycle phases. The lifecycle phases assumed in the book are requirements analysis, system design, implementation, integration and deployment, and operation and maintenance.

4. A software system is merely a part of a much larger enterprise information system.

5. Software process is part of a business process. The result of a software process is software. The result of a business process is business.

6. A software system can service any of the three management levels – operational, tactical or strategic.

7. The immaterial and changeable nature of software are but two factors that make software engineering different from traditional engineering.

8. Software engineering is more than programming. Software engineering applies to complex problems that cannot be solved by programming alone.

9. Software engineering is about modeling. All products of software engineering, including programs, are models of reality. Modeling uses abstraction to represent concepts with various levels of detail.

10. Software systems are complex. The complexity of modern software is in the “wires” – in the linkages and communication paths between components.

11. Requirements analysis are the activities of determining and specifying user requirements. Accordingly, requirements analysis is divided into requirements determination and requirements specification. Requirements engineering are any more rigorous and formal tasks in support of requirements analysis.

12. System design follows requirements analysis and it is the modeling that takes into consideration the platform on which the system is to be implemented. There are two different aspects of system design – architectural design and detailed design. A major objective of architectural design is to lead to a system that is supportable – understandable, maintainable, and scalable. The detailed design must conform to architectural design.

13. Implementation is mostly programming, but it includes other engineering activities such as component engineering and roundtrip engineering. Debugging and testing are an integral part of implementation.

14. Integration assembles the application from the set of components previously implemented and tested. Deployment is the handing over of the system to customers for production use. Integration testing is an important contributor to successful integration and acceptance testing must be done prior to deployment.

15. Operation signifies the lifecycle phase when the software product is used in day-to-day operations and the previous system (manual or automated) is phased out. Phasing out is usually a staged process. Operation coincides with the start of product maintenance. The maintenance can be corrective, adaptive or perfective.
16. There are various lifecycle models that can be adopted for software development. They are broadly divided into the waterfall models with feedback and the iterative models with increments.

17. The waterfall models are characterized by a linear sequence of phases, in which the previous phase must be completed before the next one can begin. Waterfall models are not suitable for modern software production.

18. The book discusses four main representatives of iterative models: spiral, Rational Unified Process (RUP), Model Driven Architecture (MDA), and the agile model.

19. The spiral model is really a generic metamodel that encompasses all iterative models. The model consists of four lifecycle quadrants: planning, risk analysis, engineering, and customer evaluation. Risk analysis is the most characteristic feature of the spiral model.

20. RUP is more than a lifecycle model. It is also a support environment (called the RUP platform) to assist developers in using and conforming to the RUP lifecycle. Like the spiral model, RUP is explicit about risk management.

21. The MDA model is based on the idea of executable specifications. It is a modern representative of the transformation model, which in turn has its origins in formal systems development. The component technology is at the heart of the MDA model.

22. The agile development stresses that software production is a creative activity that depends on people and team collaboration far more than on processes, tools, documentation, planning and other formalities.

**Key Terms**

- abstraction .............................................. 10, 12
- acceptance testing .......................................... 17
- agile software development ................................. 27
- architectural design ........................................ 15
- architectural model ........................................ 10
- build ......................................................... 21
- business process ........................................... 6
- CASE ....................................................... 26
- Computer Assisted Software Engineering component technology ........................................... 26
- configuration ................................................ 13
- deployment .................................................. 16
- detailed design ............................................. 14
- detailed design model ...................................... 10
- effectiveness .................................................. 6
- efficiency ....................................................... 6
- enterprise information system ............................ 1, 5
- executable specifications .................................... 25
- eXtreme Programming ...................................... 28
- formal systems development ............................. 23
- functional paradigm ....................................... 10
- IDE .......................................................... 15
- implementation ............................................ 15
- increment ...................................................... 21
- information system ......................................... 5
- Integrated Development Environment ................. 16
- integration ..................................................... 16
- integration testing .......................................... 16
- iteration ....................................................... 21
- lifecycle ...................................................... 4, 18
- legacy system .............................................. 4, 18
- lifecycle model .............................................. 18
- lifecycle phases ............................................. 3
- maintenance .................................................. 18
- management levels ......................................... 6
- MDA ........................................................... 9
- Model Driven Architecture ................................ 25
- object-oriented paradigm .................................... 10
- OLAP ........................................................ 10
- OnLine Analytical Processing ............................ 6
- OnLine Transaction Processing .......................... 6
- operation ..................................................... 18
- pair programming .......................................... 28
- program model ............................................ 10
- programming ............................................... 28
- project management ........................................ 9
- prototyping model .......................................... 19
- Rational Unified Process ................................ 24
- refactoring .................................................... 28
- requirements analysis ....................................... 13
- requirements determination ............................... 13
- requirements document ..................................... 14
- requirements engineering .................................. 13
- requirements model ........................................ 10
- requirements specification ................................ 13
Review Questions

1. Explain how software engineering and system engineering relate to each other. Is this a containment or intersection relationship? Is it possible that the two concepts may not relate at all?

2. What are the five main facets of software engineering? Can you think of software engineering concerns not obviously covered by these facets?

3. What factors decide that a system is labeled as legacy system? Can a legacy system be turned into a modern system? How can this be done, if at all?

4. What do we mean by saying that information systems are social systems? Can a system be social? Consider the following explanation of the term ‘social’ – “relating to activities in which you meet and spend time with other people and which happen during the time when you are not working” (Cambridge, 2003).

5. In the management science there is a strong distinction made between efficiency and effectiveness of systems and people. In normal usage, the term ‘efficiency’ means “when someone or something uses time and energy well, without wasting any” (Cambridge, 2003). The term ‘effectiveness’ means “how successful it is … in achieving the results you want” (Cambridge, 2003). How do these two terms apply to software engineering? Is software engineering about efficiency or effectiveness, or both? Explain. Give examples.

6. What are the principal differences between a database and a data warehouse? How do these two concepts relate to management levels in an organization?

7. What is a supportable system? Can a supportable system become a legacy system? Explain.

8. What do we mean by roundtrip engineering? How does it relate to programming?

9. Software engineering is concerned with system modeling. What is modeling in software engineering? How does it relate to the notion of abstraction? Does it make sense to talk about program modeling?
1. SOFTWARE DEVELOPMENT LIFECYCLE

10. How do you understand the distinction between the functional and object-oriented paradigms in system development?

11. Which factor is most important in deciding about the complexity of modern object-oriented system? What is the main technique of controlling and minimizing the complexity? Explain.

12. Explain the relationship between requirements analysis and requirements engineering.

13. What is the distinction between requirements determination and requirements specification? Explain.

14. Draw a line between requirements analysis and system design. When analysis becomes designing?

15. How detailed design and architectural design relate to each other?

16. What are the main approaches and techniques of testing?

17. Explain the use of stubs and drivers in integration testing.

18. In modern software engineering, the waterfall lifecycle model is replaced by iterative models. Are there any aspects of the waterfall model, missing or not feasible in iterative models, that would benefit iterative approaches? Discuss.

19. What is risk management? Which lifecycle models are most explicit about risk management? Explain.

20. What are executable specifications? Which lifecycle model uses executable specifications as its focal point? Explain.

21. What are the most uncommon aspects of the agile development? (Uncommon in comparison with other iterative approaches). Explain.