Software and Software Engineering: The nature of Software, The unique nature of WebApps, Software Engineering, The software Process, The software Engineering practice, The software myths, How it all starts

**What is software?**

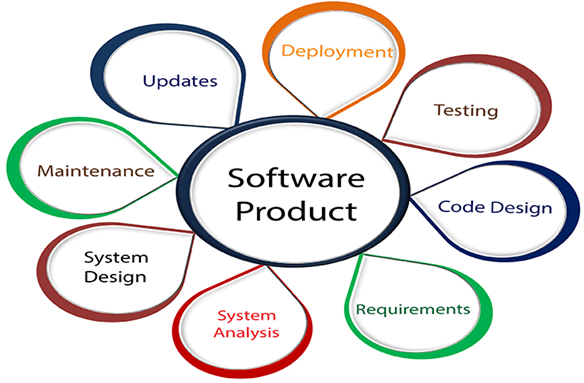
Software is: (1) instructions (computer programs) that when executed provide desired features, function, and performance; (2) data structures that enable the programs to adequately manipulate information, and (3) descriptive information in both hard copy and virtual forms that describes the operation and use of the programs.

##### **What is Software Engineering?**

The term **software engineering** is the product of two words, **software**, and **engineering**. The **software** is a collection of integrated programs.

Software subsists of carefully-organized instructions and code written by developers on any of various particular computer languages.

Computer programs and related documentation such as requirements, design models and user manuals.

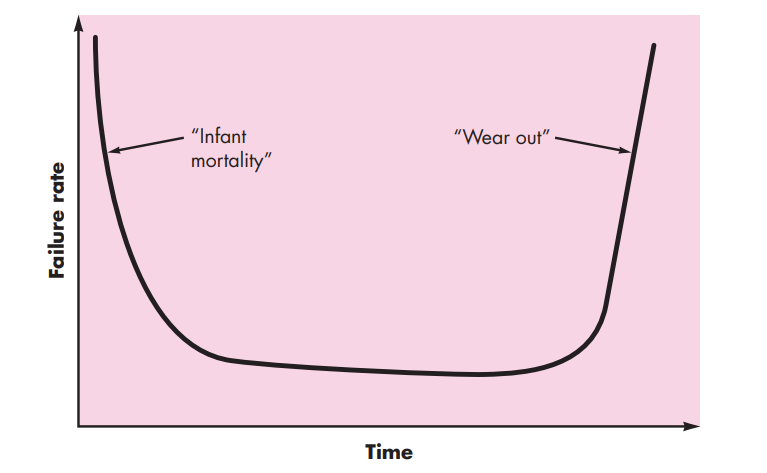
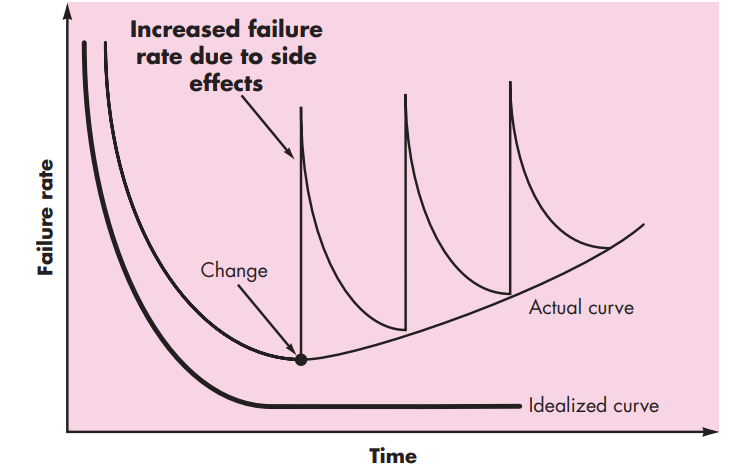
**Engineering** is the application of **scientific** and **practical** knowledge to **invent, design, build, maintain**, and **improve frameworks, processes, etc**.

**Software Engineering** is an engineering branch related to the evolution of software product using well-defined scientific principles, techniques, and procedures. The result of software engineering is an effective and reliable software product.

software has characteristics that are considerably different than those of hardware:

1. **Software is developed or engineered; it is not manufactured in the classical sense**.**Different approaches for creating software vs. hardware:**
   * Software is designed and programmed, focusing on good design practices to ensure quality. Hardware is manufactured, and quality can be affected by issues during production.
   * Software development costs are primarily in the engineering stage. This means managing software projects differently from hardware projects, which involve manufacturing costs, is essential.
2. **Software doesn't wear out physically, but it can degrade over time:**
   * Hardware components are susceptible to wear and tear from everyday use and environmental factors. This results in a higher failure rate over time, like a bathtub curve.
   * Ideally, software shouldn't wear out because it's not physical. However, adding new features or making changes can introduce bugs, causing the failure rate to increase over time (unlike the idealized flat curve). This is because there are no "spare parts" for software like there are for hardware. Fixing software problems requires revisiting the code and potentially rewriting parts, making it more complex than replacing a faulty hardware component.
3. **Although the industry is moving toward component-based construction, most software continues to be custom built,** **Software development is catching up on reusability:**
   * Hardware engineers can leverage pre-built components like screws and circuits to design new systems faster. This reusability has been a standard practice in hardware engineering for a long time.
   * Software development is only recently adopting a component-based approach. Reusable software components encapsulate both data and the way it's processed, allowing developers to build new applications by assembling these pre-built blocks. This approach can significantly speed up software development.

In essence, software and hardware are built differently, and this difference is reflected in how they function, wear out, and are created.



**Software Application Domains**

Today, seven broad categories of computer software present continuing challenges for software engineers:

**System software**—Infrastructure software come under this category like compilers, operating systems, editors, drivers, etc. Basically system software is a collection of programs to provide service to other programs.

**Application software**— Application software comprises various programs designed to perform specific tasks that aid users in executing tasks, solving problems, or managing operations. These programs are created to help with specific activities. Whether it's drafting a document, managing data, or editing a video, each application software is tailored to meet these individual needs. In some industries, real-time data processing is crucial. Application software helps in managing operations that need immediate attention.

**Engineering/scientific software**— Engineering and scientific software used to focus mainly on complex calculations, solving problems with a lot of mathematical processing. These software tools were used in various fields like astronomy, studying volcanoes, testing car strength, planning space shuttle paths, studying molecules, and automated manufacturing.

**Embedded software:** This type of software is placed in “Read-Only- Memory (ROM)”of the product and control the various functions of the product. The product could be an aircraft, automobile, security system, signalling system, control unit of power plants, etc. the embedded software handles hardware components and is also termed as intelligent software .

**Product-line software**— Product-line software is created to do specific tasks and can be used by many different people. Some of this software is made for special groups, like tools to help manage inventory in stores. Other types are made for lots of people to use, like word processors, spreadsheets, drawing programs, games, and software to manage databases or finances for both personal and business use.

**Web applications**— Web applications, or "WebApps," are software programs you use over the internet. At their simplest, WebApps can be basic websites with linked pages that show text and pictures. However, with the development of Web 2.0, WebApps are becoming much more advanced. Now, they can do a lot more than just show information. They can offer various features and functions, work like computer programs, and connect with business systems and databases to provide more integrated and powerful services to users.

**The unique nature of WebApps**

The following attributes are encountered in the vast majority of WebApps.

**Network intensiveness**. A WebApp resides on a network and must serve the needs of a diverse community of clients. The network may enable worldwide access and communication (i.e., the Internet) or more limited access and communication (e.g., a corporate Intranet).

**Concurrency**. A large number of users may access the WebApp at one time. In many cases, the patterns of usage among end users will vary greatly.

**Unpredictable load.** The number of users of the WebApp may vary by orders of magnitude from day to day. One hundred users may show up on Monday; 10,000 may use the system on Thursday.

**Performance.** If a WebApp user must wait too long (for access, for serverside processing, for client-side formatting and display), he or she may decide to go elsewhere.

**Availability**. Although expectation of 100 percent availability is unreasonable, users of popular WebApps often demand access on a 24/7/365 basis.

**Data driven**. The primary function of many WebApps is to use hypermedia to present text, graphics, audio, and video content to the end user. In addition, WebApps are commonly used to access information that exists on databases that are not an integral part of the Web-based environment (e.g., e-commerce or financial applications).

**Content sensitive**. The quality and aesthetic nature of content remains an important determinant of the quality of a WebApp.

**Continuous evolution.** Unlike conventional application software that evolves over a series of planned, chronologically spaced releases, Web applications evolve continuously. It is not unusual for some WebApps (specifically, their content) to be updated on a minute-by-minute schedule or for content to be independently computed for each request.

**Security**. Because WebApps are available via network access, it is difficult, if not impossible, to limit the population of end users who may access the application. In order to protect sensitive content and provide secure modes of data transmission, strong security measures must be implemented throughout the infrastructure that supports a WebApp and within the application itself.

**Aesthetics.** An undeniable part of the appeal of a WebApp is its look and feel. When an application has been designed to market or sell products or ideas, aesthetics may have as much to do with success as technical design.

**Software Engineering**

## **What is Software Engineering?**

**Software Engineering** is the process of designing, developing, testing, and maintaining software. It is a systematic and disciplined approach to software development that aims to create high-quality, reliable, and maintainable software.

IEEE defines software engineering as:

(1) The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is, the application of engineering to software.

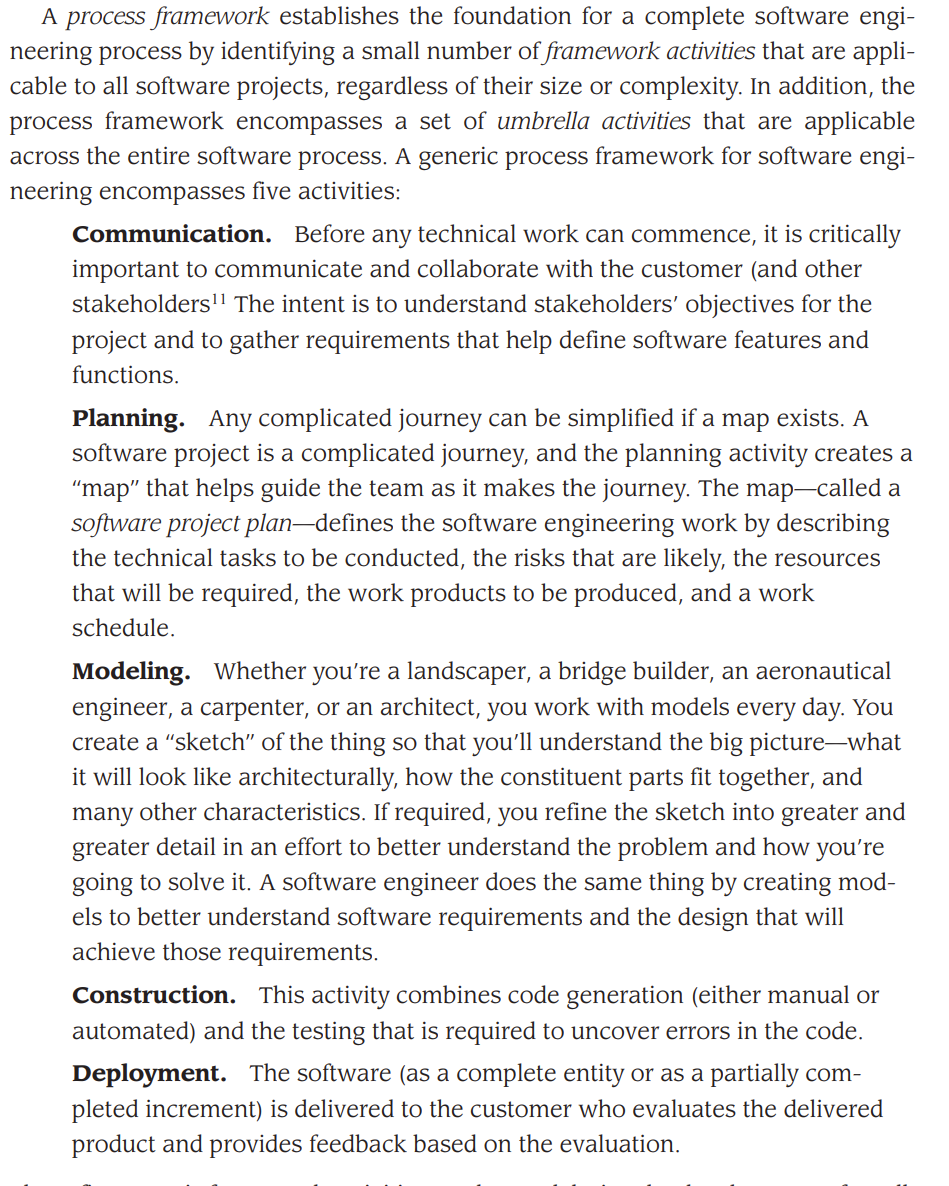
(2) The study of approaches as in the above statement.

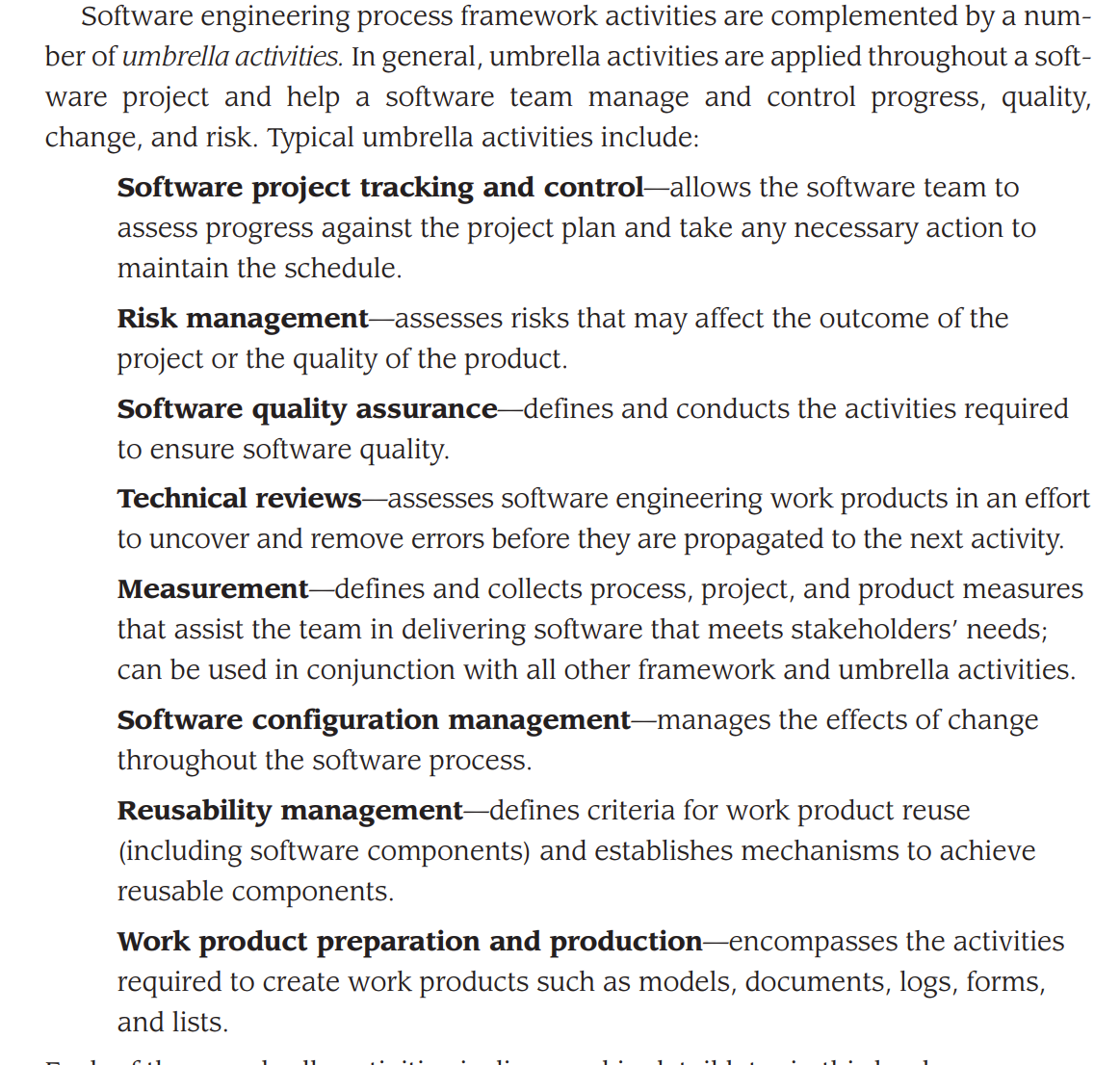
1. Software engineering includes a variety of techniques, tools, and methodologies, including requirements analysis, design, testing, and maintenance.
2. It is a rapidly evolving field, and new tools and technologies are constantly being developed to improve the software development process.
3. By following the principles of software engineering and using the appropriate tools and methodologies, software developers can create high-quality, reliable, and maintainable software that meets the needs of its users.
4. Software Engineering is mainly used for large projects based on software systems rather than single programs or applications.
5. The main goal of Software Engineering is to develop software applications for improving quality,  budget, and time efficiency.
6. Software Engineering ensures that the software that has to be built should be consistent, correct, also on budget, on time, and within the required requirements.

## **Key Principles of Software Engineering**

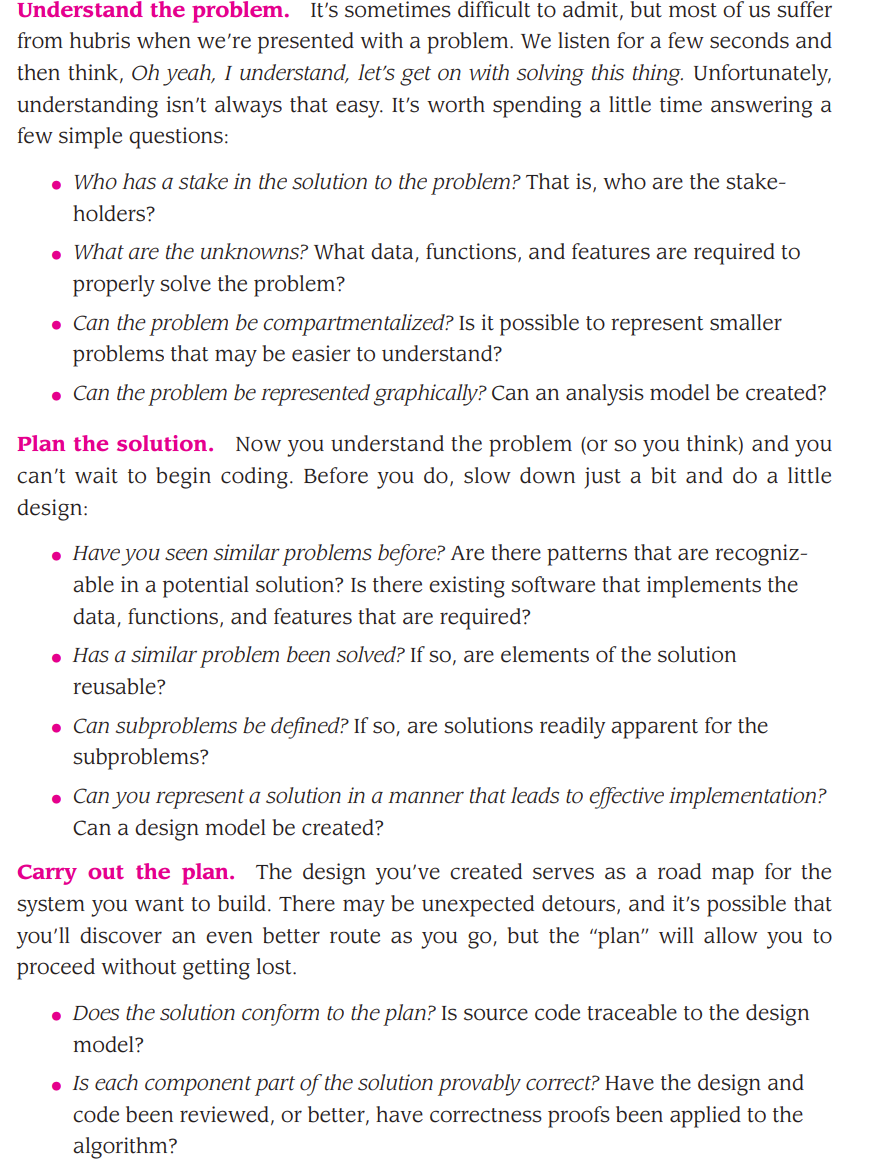
1. **Modularity**: Breaking the software into smaller, reusable components that can be developed and tested independently.
2. **Abstraction**: Hiding the implementation details of a component and exposing only the necessary functionality to other parts of the software.
3. **Encapsulation**: Wrapping up the data and functions of an object into a single unit, and protecting the internal state of an object from external modifications.
4. **Reusability**: Creating components that can be used in multiple projects, which can save time and resources.
5. **Maintenance**: Regularly updating and improving the software to fix bugs, add new features, and address security vulnerabilities.
6. **Testing**: Verifying that the software meets its requirements and is free of bugs.
7. **Design Patterns**: Solving recurring problems in software design by providing templates for solving them.
8. **Agile methodologies:**Using iterative and incremental development processes that focus on customer satisfaction, rapid delivery, and flexibility.
9. **Continuous Integration & Deployment:** Continuously integrating the code changes and deploying them into the production environment.

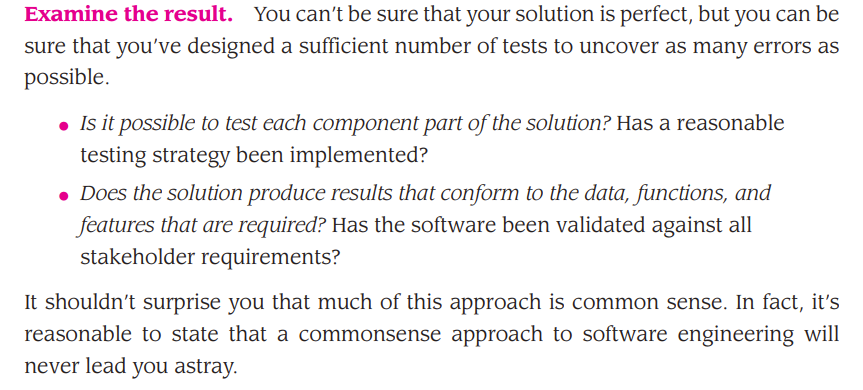
**The software Process**

****



**The software Engineering practice**

****

****

**GENEARAL PRINCIPLES**

## **The First Principle: *The Reason It All Exists***

A software system exists for one reason: *to provide value to its users*. All decisions should be made with this in mind. Before specifying a system requirement, before noting a piece of system functionality, before determining the hardware platforms or development processes, ask yourself questions such as: “Does this add real value to the system?” If the answer is “no,” don’t do it. All other principles support this one.

## **The Second Principle: KISS (Keep It Simple, Stupid!)**

Software design is not a haphazard process. There are many factors to consider in any design effort. *All design should be as simple as possible, but no simpler*. This facilitates having a more easily understood and easily maintained system. This is the more simple ones. Simple also does not mean “quick and dirty.” In fact, it often takes a lot of thought and work over multiple iterations to simplify. The payoff is software that is more maintainable and less error-prone.

**The Third Principle: *Maintain the Vision***

*A clear vision is essential to the success of a software project*. Without one, a project almost unfailingly ends up being “of two [or more] minds” about itself. Without conceptual integrity, a system threatens to become a patchwork of incompatible designs, held together by the wrong kind of screws. . . . Compromising the architectural vision of a software system weakens and will eventually break even the well-designed systems. Having an empowered architect who can hold the vision and enforce compliance helps ensure a very successful software project.

### **The Fourth Principle: *What You Produce, Others Will Consume***

Industrial-strength software systems are rarely created or used in isolation. Other people will need to use, maintain, document, or understand your system in some way. Therefore, when you create software, always remember that someone else will need to understand it.

The audience for software is often large and varied. When you specify requirements, think about the users. When you design, consider the people who will build the system. When you write code, think about those who will maintain and expand it. Someone will eventually need to fix any bugs in your code, making them a user of your code too. Making their job easier adds value to the system.

**The Fifth Principle: *Be Open to the Future***

A system with a long lifetime has more value. In today’s computing environments, where specifications change on a moment’s notice and hardware platforms are obsolete just a few months old, software lifetimes are typically measured in months instead of years. However, true “industrial-strength” software systems must endure far longer. To do this successfully, these systems must be ready to adapt to these and other changes. Systems that do this successfully are those that have been designed this way from the start. *Never design yourself into a corner*. Always ask “what if,” and prepare for all possible answers by creating systems that solve the general problem, not just the specific one.[[1]](#footnote-1) This could very possibly lead to the reuse of an entire system.

**The Sixth Principle: *Plan Ahead for Reuse***

Reuse saves time and effort.Achieving a high level of reuse is arguably the hardest goal to accomplish in developing a software system. The reuse of code and designs has been proclaimed as a major benefit of using object-oriented technologies. However, the return on this investment is not automatic. To leverage the reuse possibilities that object-oriented [or conventional] programming provides requires forethought and planning. There are many techniques to realize reuse at every level of the system development process. . . . *Planning ahead for reuse reduces the cost and increases the value of both the reusable components and the systems into which they are incorporated*.

**The Seventh principle: *Think!***

This last principle is probably the most overlooked. *Placing clear, complete thought before action almost always produces better results*. When you think about something, you are more likely to do it right. You also gain knowledge about how to do it right again. If you do think about something and still do it wrong, it becomes a valuable experience.

**The software myths, How it all starts**

**Management myths.** Managers with software responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep schedules from slipping, and improve quality. Like a drowning person who grasps at a straw, a software manager often grasps at belief in a software myth, if that belief will lessen the pressure (even temporarily).

|  |  |
| --- | --- |
| **Myth:** | *We already have a book that’s full of standards and procedures for building software. Won’t that provide my people with everything they need to know?* |
| **Reality:** | The book of standards may very well exist, but is it used? Are software practitioners aware of its existence? Does it reflect modern software engineering practice? Is it complete? Is it adaptable? Is it streamlined to improve time-to-delivery while still maintaining a focus on quality? In many cases, the answer to all of these questions is “no.” |
| **Myth:** | *If we get behind schedule, we can add more programmers and catch up (sometimes called the “Mongolian horde” concept).* |
| **Reality:** | Software development is not a mechanistic process like manufacturing. In the words of Brooks [Bro95]: “adding people to a late software project makes it later.” At first, this statement may seem counterintuitive. However, as new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive development effort. People can be added but only in a planned and wellcoordinated manner. |
| **Myth:** | *If I decide to outsource the software project to a third party, I can just relax and let that firm build it.* |
| **Reality:** | If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it outsources software projects. |

**Customer myths.** A customer who requests computer software may be a person at the next desk, a technical group down the hall, the marketing/sales department, or an outside company that has requested software under contract. In many cases, the customer believes myths about software because software managers and practitioners do little to correct misinformation. Myths lead to false expectations (by the customer) and, ultimately, dissatisfaction with the developer.

**Myth:** *A general statement of objectives is sufficient to begin writing programs—we can fill in the details later.*

**Reality:** Although a comprehensive and stable statement of requirements is not always possible, an ambiguous “statement of objectives” is a recipe for disaster. Unambiguous requirements (usually derived through effective and continuous communication between customer and developer.

|  |  |
| --- | --- |
| **Myth:** | *Software requirements continually change, but change can be easily accommodated because software is flexible.* |
| **Reality:** | It is true that software requirements change, but the impact of |

change varies with the time at which it is introduced. When requirements changes are requested early (before design or code has been started), the cost impact is relatively small.[[2]](#footnote-2) However, as time passes, the cost impact grows rapidly—resources have been committed, a design framework has been established, and change can cause upheaval that requires additional resources and major design modification.

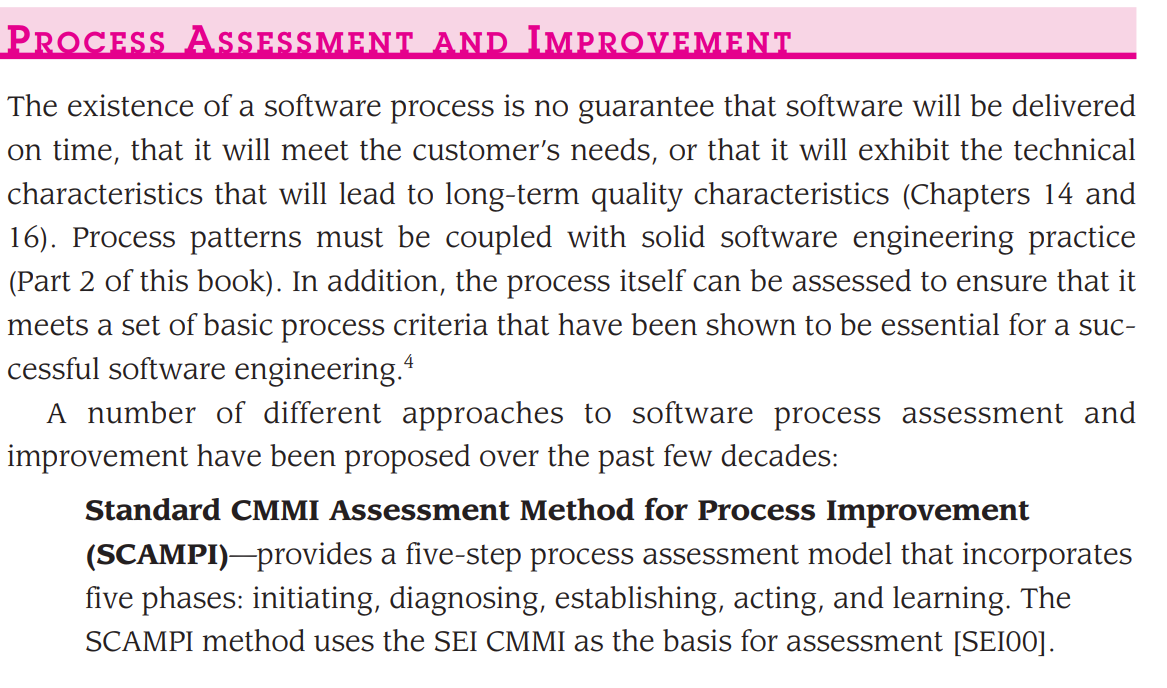
**Practitioner’s myths.** Myths that are still believed by software practitioners have been fostered by over 50 years of programming culture. During the early days, programming was viewed as an art form. Old ways and attitudes die hard.

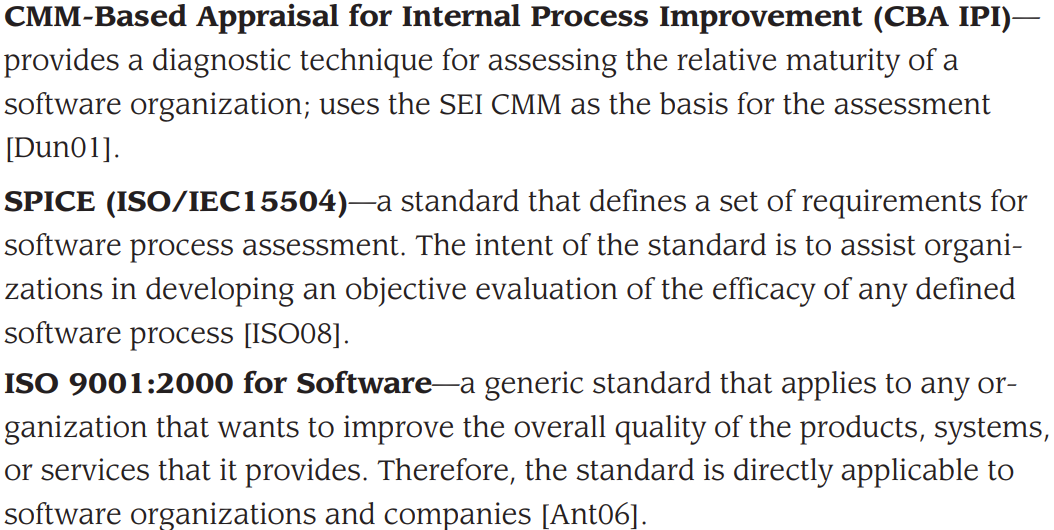
|  |  |
| --- | --- |
| **Myth:** | *Once we write the program and get it to work, our job is done.* |
| **Reality:** | Someone once said that “the sooner you begin ‘writing code,’ the longer it’ll take you to get done.” Industry data indicate that between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer for the first time. |
| **Myth:** | *Until I get the program “running” I have no way of assessing its quality.* |
| **Reality:** | One of the most effective software quality assurance mechanisms can be applied from the inception of a project—*the technical review.* Software reviews (described in Chapter 15) are a “quality filter” that have been found to be more effective than testing for finding certain classes of software defects. |
| **Myth:** | *The only deliverable work product for a successful project is the working program.* |
| **Reality:** | A working program is only one part of a software configuration that includes many elements. A variety of work products (e.g., models, documents, plans) provide a foundation for successful engineering and, more important, guidance for software support. |
| **Myth:** | *Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down.* |
| **Reality:** | Software engineering is not about creating documents. It is about creating a quality product. Better quality leads to reduced rework.  And reduced rework results in faster delivery times. |

Process Models: A generic process model, Process assessment and improvement, Prescriptive process models, Waterfall model, Incremental process models, Evolutionary process models, Concurrent models, Specialized process models.

**A generic process model**

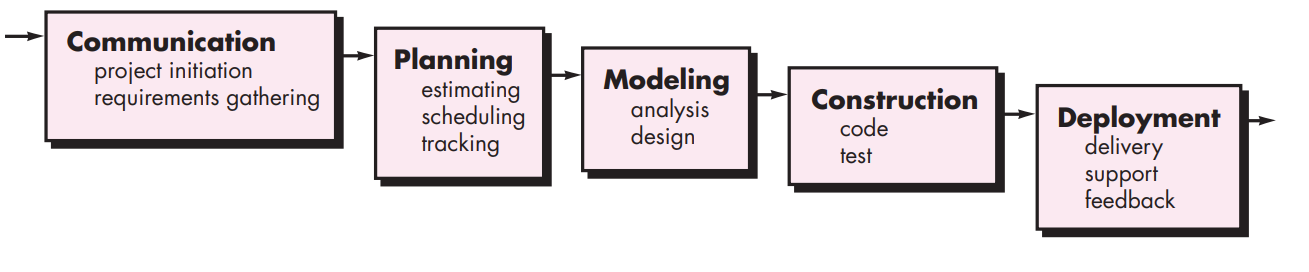
**Process assessment and improvement**





**Prescriptive models**

**1 waterfall model**

****

This model has five phases: Requirements analysis and specification, design, implementation, and unit testing, integration and system testing, and operation and maintenance. The steps always follow in this order and do not overlap. The developer must complete every phase before the next phase begins. This model is named "**Waterfall Model**", because its diagrammatic representation resembles a cascade of waterfalls.

**1. Requirements analysis and specification phase:** The aim of this phase is to understand the exact requirements of the customer and to document them properly. Both the customer and the software developer work together so as to document all the functions, performance, and interfacing requirement of the software. It describes the "what" of the system to be produced and not "how."In this phase, a large document called **Software Requirement Specification (SRS)** document is created which contained a detailed description of what the system will do in the common language.

**2. Design Phase:** This phase aims to transform the requirements gathered in the SRS into a suitable form which permits further coding in a programming language. It defines the overall software architecture together with high level and detailed design. All this work is documented as a Software Design Document (SDD).

**3. Implementation and unit testing:** During this phase, design is implemented. If the SDD is complete, the implementation or coding phase proceeds smoothly, because all the information needed by software developers is contained in the SDD.

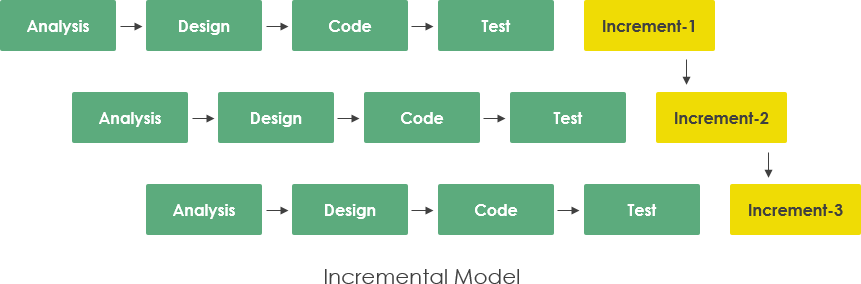
During testing, the code is thoroughly examined and modified. Small modules are tested in isolation initially. After that these modules are tested by writing some overhead code to check the interaction between these modules and the flow of intermediate output.

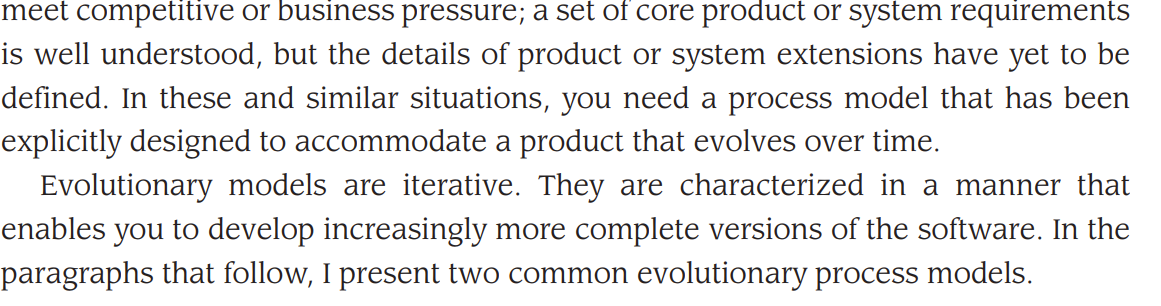
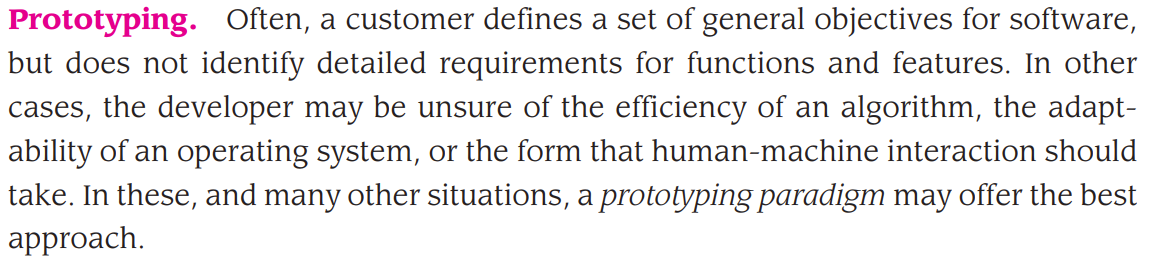
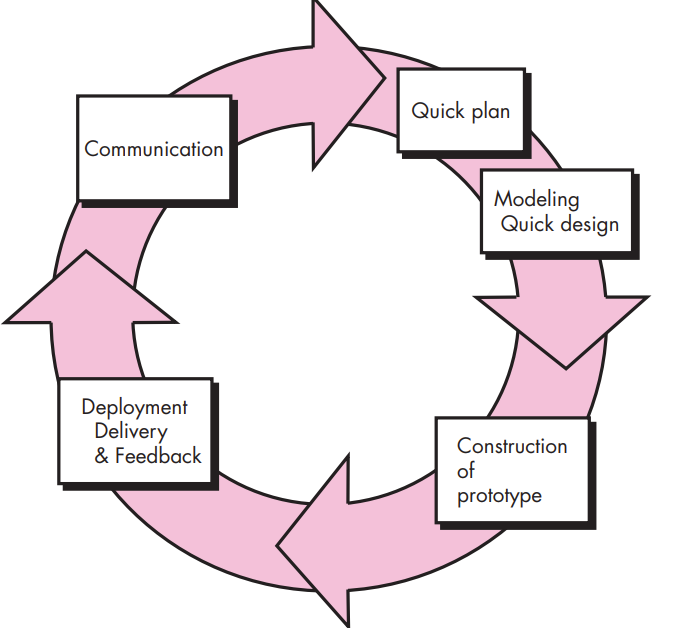
**4. Integration and System Testing:** This phase is highly crucial as the quality of the end product is determined by the effectiveness of the testing carried out. The better output will lead to satisfied customers, lower maintenance costs, and accurate results. Unit testing determines the efficiency of individual modules. However, in this phase, the modules are tested for their interactions with each other and with the system.

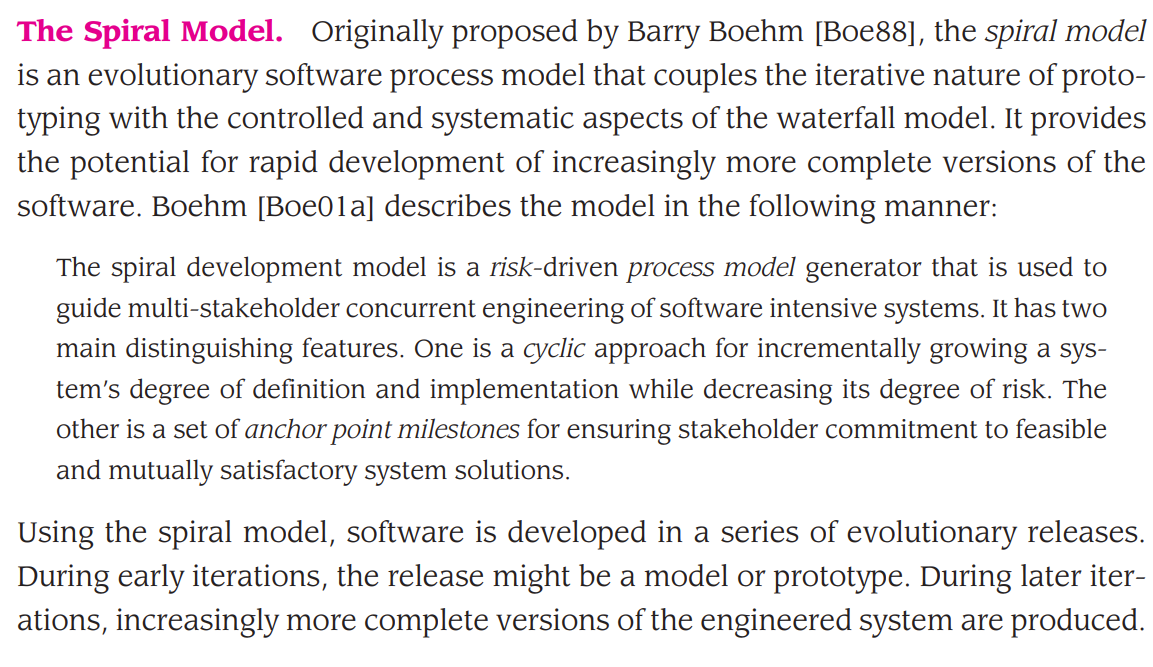
**5. Operation and maintenance phase:** Maintenance is the task performed by every user once the software has been delivered to the customer, installed, and operational.

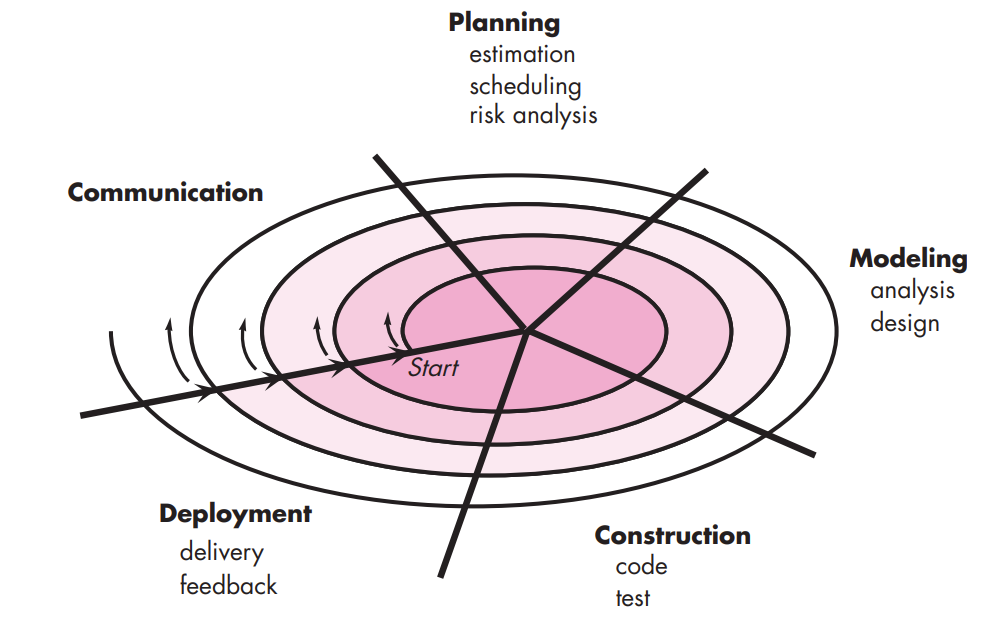
**Incremental model**

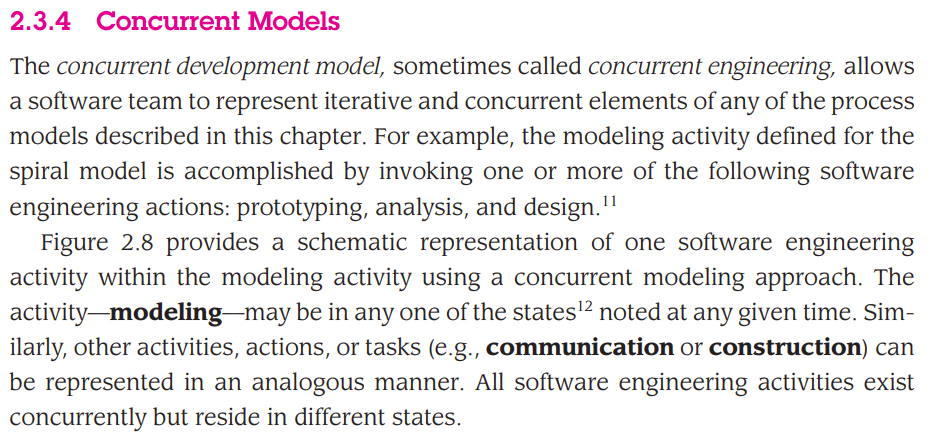
The incremental build model is a method of software development where the model is designed, implemented and tested incrementally (a little more is added each time) until the product is finished. It involves both development and maintenance. The product is defined as finished when it satisfies all of its requirements. Each iteration passes through the requirements, design, coding and testing phases. And each subsequent release of the system adds function to the previous release until all designed functionally has been implemented. This model combines the elements of the waterfall model with the iterative philosophy of prototyping.

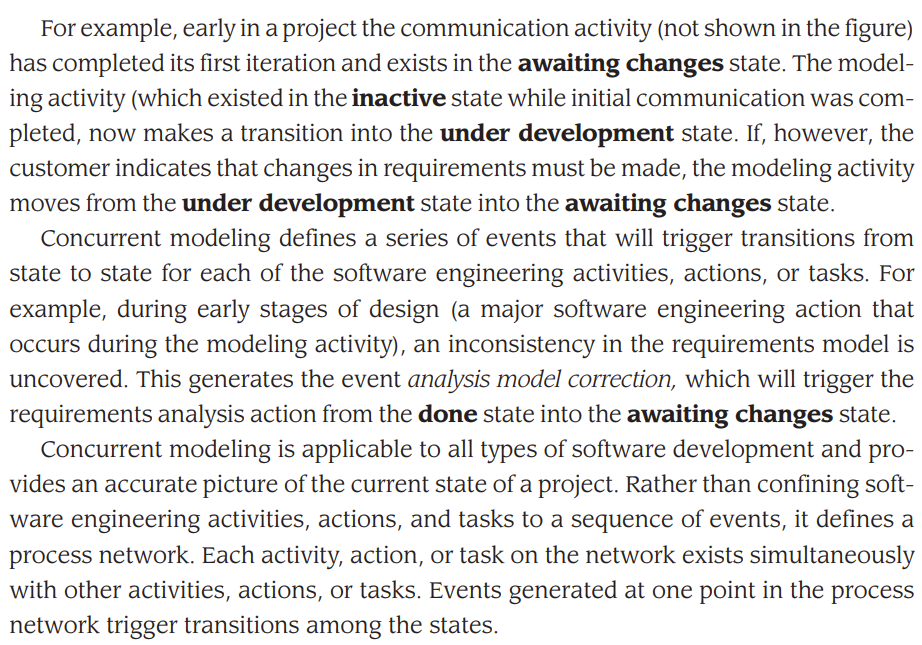


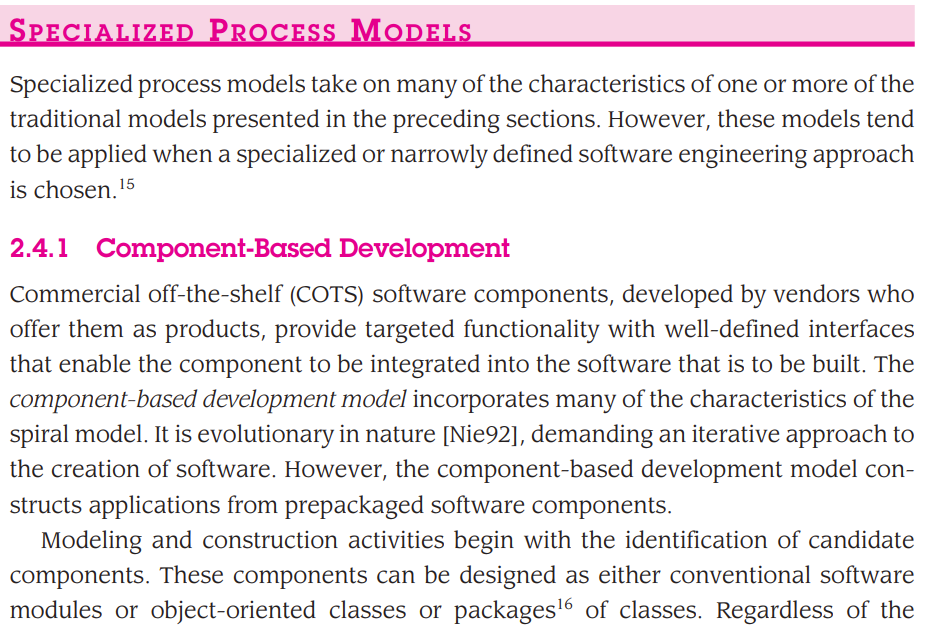
**** ****  

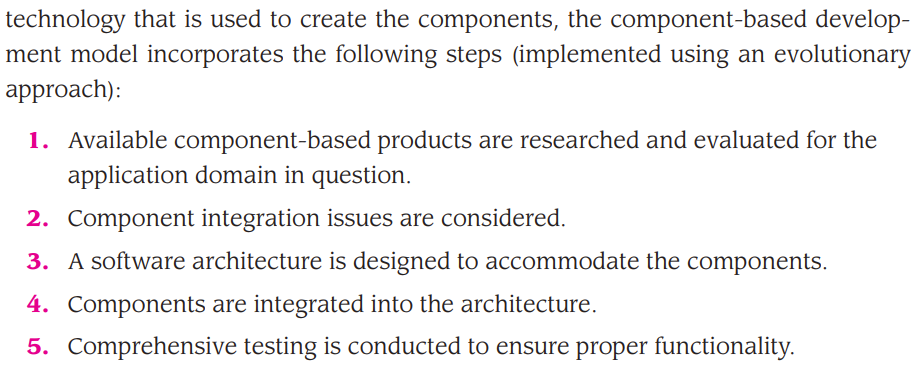


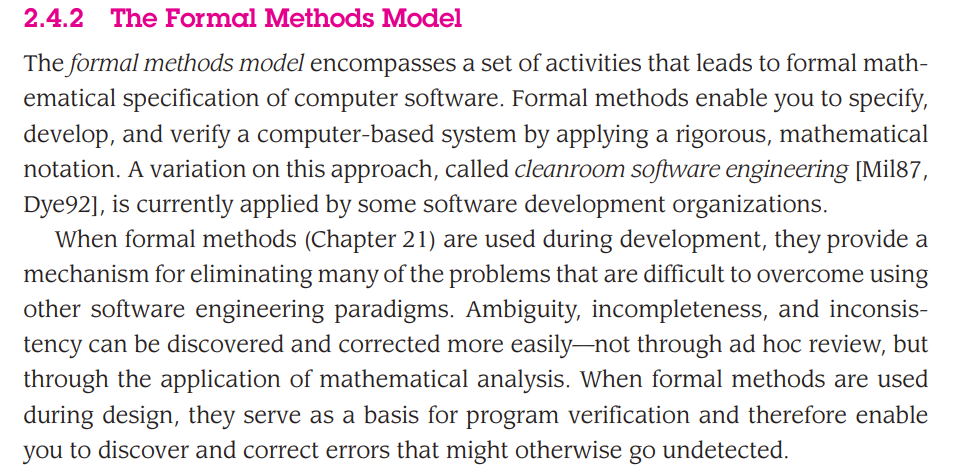




****

****

****

****

1. [↑](#footnote-ref-1)
2. [↑](#footnote-ref-2)