Software Engineering

What is software?
Many people equate the term software with computer programs. However, I prefer a broader definition where software is not just the programs but also all associated documentation and configuration data that is needed to make these programs operate correctly. A software system usually consists of a number of separate programs, configuration files, which are used to set up these programs, system documentation, which describes the structure of the system, and user documentation, which explains how to use the system and web sites for users to download recent product information. Software engineers are concerned with developing software products, i.e., software which can be sold to a customer. There are two fundamental types of software product:

1. **Generic products**: These are stand-alone systems that are produced by a development organization and sold on the open market to any customer who is able to buy them. Examples of this type of product include software for PCs such as databases, word processors, drawing packages and project management tools.

2. **Customized products**: These are systems which are commissioned by a particular customer. A software contractor develops the software especially for that customer. Examples of this type of software include control systems for electronic devices, systems written to support a particular business process and air traffic control systems.

What is software engineering?
Software engineering is an engineering discipline that is concerned with all aspects of software production from the early stages of system specification to maintaining the system after it has gone into use. In this definition, there are two key phrases:

1. **Engineering discipline** Engineers make things work. They apply theories, methods and tools where these are appropriate, but they use them selectively and always try to discover solutions to problems even when there are no applicable theories and methods. Engineers also recognize that they must work to organizational and financial constraints, so they look for solutions within these constraints.

2. **All aspects of software production** Software engineering is not just concerned with the technical processes of software development but also with activities such as software project management and with the development of tools, methods and theories to support software production.

In general, software engineers adopt a systematic and organized approach to their work, as this is often the most effective way to produce high-quality software. However, engineering is all about selecting the most appropriate method for a set of circumstances and a more creative, less formal approach to development may be effective in some circumstances. Less formal development is particularly appropriate for the development of web-based systems, which requires a blend of software and graphical design skills.

What’s the difference between software engineering and computer science?
Essentially, computer science is concerned with the theories and methods that underlie computers and software systems, whereas software engineering is concerned with the practical problems of producing software. Some knowledge of computer science is essential for software engineers in the same way that some knowledge of physics is essential for electrical engineers. Ideally, all of software engineering should be underpinned by theories of computer science, but in reality this is not the case. Software engineers must often use ad hoc approaches to developing the software. Elegant theories of computer science cannot always be applied to real, complex problems that require a software solution.
What is the difference between software engineering and system engineering?

System engineering is concerned with all aspects of the development and evolution of complex systems where software plays a major role. System engineering is therefore concerned with hardware development, policy and process design and system deployment as well as software engineering. System engineers are involved in specifying the system, defining its overall architecture and then integrating the different parts to create the finished system. They are less concerned with the engineering of the system components (hardware, software, etc.). System engineering is an older discipline than software engineering. People have been specifying and assembling complex industrial systems such as aircraft and chemical plants for more than a hundred years. However, as the percentage of software in systems has increased, software engineering techniques such as use-case modeling and configuration management are being used in the systems engineering process.

What is a software process?

A software process is the set of activities and associated results that produce a software product. There are four fundamental process activities that are common to all software processes. These are:

1. **Software specification** where customers and engineers define the software to be produced and the constraints on its operation.
2. **Software development** where the software is designed and programmed.
3. **Software validation** where the software is checked to ensure that it is what the customer requires.
4. **Software evolution** where the software is modified to adapt it to changing customer and market requirements.

Different types of systems need different development processes. For example, real-time software in an aircraft has to be completely specified before development begins whereas, in e-commerce systems, the specification and the program are usually developed together. Consequently, these generic activities may be organized in different ways and described at different levels of detail for different types of software. However, use of an inappropriate software process may reduce the quality or the usefulness of the software product to be developed and/or increase the development costs.

What is a software process model?

A software process model is a simplified description of a software process that presents one view of that process. Process models may include activities that are part of the software process, software products and the roles of people involved in software engineering. Some examples of the types of software process model that may be produced are:

1. **A workflow model**: This shows the sequence of activities in the process along with their inputs, outputs and dependencies. The activities in this model represent human actions.
2. **A dataflow or activity model**: This represents the process as a set of activities, each of which carries out some data transformation. It shows how the input to the process, such as a specification, is transformed to an output, such as a design. The activities here may represent transformations carried out by people or by computers.
3. **A role/action model**: This represents the roles of the people involved in the software process and the activities for which they are responsible.

Most software process models are based on one of three general models or paradigms of software development:

1. **The waterfall approach** This takes the above activities and represents them as separate process phases such as requirements specification, software design, implementation, testing and so on. After each stage is defined it is 'signed-off', and development goes on to the following stage.
2. **Iterative development** This approach interleaves the activities of specification, development and validation. An initial system is rapidly developed from very abstract specifications. This is then refined with customer input to produce a system that satisfies the customer's needs. The system may then be delivered. Alternatively, it may be re-implemented using a more structured approach to produce a more robust and maintainable system.

3. **Component-based software engineering (CBSE)** This technique assumes that parts of the system already exist. The system development process focuses on integrating these parts rather than developing them from scratch.

**What are the costs of software engineering?**

There is no simple answer to this question as the distribution of costs across the different activities in the software process depends on the process used and the type of software that is being developed. For example, real-time software usually requires more extensive validation and testing than web-based systems. However, each of the different generic approaches to software development has a different profile of cost distribution across the software process activities. If you assume that the total cost of developing a complex software system is 100 cost units then Figure below illustrates how these are spent on different process activities:

---

### Software engineering activity cost distribution

- **Waterfall Model**
- **Iterative development**
- **Component-based software engineering**

---

Ian Sommerville - Software Engineering- 8e
In the waterfall approach, the costs of specification, design, implementation and integration are measured separately. Notice that system integration and testing is the most expensive development activity. Normally, this is about 40% of the total development costs but for some critical systems it is likely to be at least 50% of the system development costs. If the software is developed using an iterative approach, there is no hard line between specification, design and development. Specification costs are reduced because only a high-level specification is produced before development in this approach. Specification, design, implementation, integration and testing are carried out in parallel within a development activity. However, you still need an independent system testing activity once the initial implementation is complete.
Component-based software engineering has only been widely used for a short time. We don't have accurate figures for the costs of different software development activities in this approach. However, we know that development costs are reduced. Integration and testing costs are increased because you have to ensure that the components that you use actually meet their specification and work as expected with other components.

What is CASE?
The acronym CASE stands for Computer-Aided Software Engineering. It covers a wide range of different types of programs that are used to support software process activities such as requirements analysis, system modeling, debugging and testing. All methods now come with associated CASE technology such as editors for the notations used in the method, analysis modules which check the system model according to the method rules and report generators to help create system documentation. The CASE tools may also include a code generator that automatically generates source code from the system model and some process guidance for software engineers.

What are the attributes of good software?
As well as the services that it provides, software products have a number of other associated attributes that reflect the quality of that software. These attributes are not directly concerned with what the software does. Rather, they reflect its behavior while it is executing and the structure and organization of the source program and associated documentation. The attributes of good software is:

1. **Maintainability**: Software should be written in such a way that it may evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable consequence of a changing business environment

2. **Dependability**: Software dependability has a range of characteristics, including reliability, security and safety. Dependable software should not cause physical or economic damage in the event of system failure.

3. **Efficiency**: Software should not make wasteful use of system resources such as memory and processor cycles. Efficiency therefore includes responsiveness, processing time, memory utilization, etc

4. **Usability**: Software must be usable, without undue effort, by the type of user for whom it is designed. This means that it should have an appropriate user interface and adequate documentation.

**Professional and ethical responsibility**
Like other engineering disciplines, software engineering is carried out within a legal and social framework that limits the freedom of engineers. Software engineers must accept that their job involves wider responsibilities than simply the application of technical skills. They must also behave
in an ethical and morally responsible way if they are to be respected as professionals. It goes without saying that you should always uphold normal standards of honesty and integrity. You should not use your skills and abilities to behave in a dishonest way or in a way that will bring disrepute to the software engineering profession. However, there are areas where standards of acceptable behavior are not bounded by laws but by the more tenuous notion of professional responsibility. Some of these are:

1. **Confidentiality** You should normally respect the confidentiality of your employers or clients irrespective of whether a formal confidentiality agreement has been signed.

2. **Competence** You should not misrepresent your level of competence. You should not knowingly accept work that is outside your competence.

3. **Intellectual property rights** You should be aware of local laws governing the use of intellectual property such as patents and copyright. You should be careful to ensure that the intellectual property of employers and clients is protected.

4. **Computer misuse** You should not use your technical skills to misuse other people's computers. Computer misuse ranges from relatively trivial.

**Software process**

A software process is a set of activities that leads to the production of a software product. These activities may involve the development of software from scratch in a standard programming language like Java or C. Increasingly, however, new software is developed by extending and modifying existing systems and by configuring and integrating off-the-shelf software or system components. Software processes are complex and, like all intellectual and creative processes, rely on people making decisions and judgments. Because of the need for judgment and creativity, attempts to automate software processes have met with limited success. Computer-aided software engineering (CASE) tools can support some process activities. However, there is no possibility, of more extensive automation where software takes over creative design from the engineers involved in the software process. One reason the effectiveness of CASE tools is limited is because of the immense diversity of software processes. There is no ideal process, and many organizations have developed their own approach to software development. Processes have evolved to exploit the capabilities of the people in an organization and the specific characteristics of the systems that are being developed. For some systems, such as critical systems, a very structured development process is required. For business systems, with rapidly changing requirements, a flexible, agile process is likely to be more effective. Although there are many software processes, some fundamental activities are common to all software processes:

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

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

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

4. **Software evolution** The software must evolve to meet changing customer needs.
Software process models
A software process model is an abstract representation of a software process. Each process model represents a process from a particular perspective, and thus provides only partial information about that process. There are number of very general process models (sometime called process paradigms). These generic models are not definitive descriptions of software processes. Rather, they are abstractions of the process that can be used to explain different approaches to software development. You can think of them as process frameworks that may be extended and adapted to create more specific software engineering processes. The process models are:

1. **The waterfall model**: This takes the fundamental process activities of specification, development, validation and evolution and represents them as separate process phases such as requirements specification, software design, implementation, testing and so on.

2. **Evolutionary development**: This approach interleaves the activities of specification, development and validation. An initial system is rapidly developed from abstract specifications. This is then refined with customer input to produce a system that satisfies the customer’s needs.

3. **Component-based software engineering**: This approach is based on the existence of a significant number of reusable components. The system development process focuses on integrating these components into a system rather than developing them from scratch.

These three generic process models are widely used in current software engineering practice. They are not mutually exclusive and are often used together, especially for large systems development.

The waterfall model
The first published model of the software development process was derived from more general system engineering processes. This is illustrated in Figure below. Because of the cascade from one phase to another, this model is known as the waterfall model or software life cycle. The principal stages of the model map onto fundamental development activities:

1. **Requirements analysis and definition** The system's services, constraints and goals are, established by consultation with system users. They are then defined in detail and serve as a system specification.

2. **System and software design** The systems design process partitions the requirements to either hardware or software systems. It establishes overall system architecture. Software design involves identifying and describing the fundamental software system abstractions and their relationships.

3. **Implementation and unit testing** During this stage, the software design is realized as a set of programs or program units. Unit testing involves verifying that each unit meets its specification.

4. **Integration and system testing** The individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met. After testing, the software system is delivered to the customer.
5. **Operation and maintenance** Normally (although not necessarily) this is the longest life-cycle phase. The system is installed and put into practical use. Maintenance involves correcting errors which were not discovered in earlier stages of the life cycle, improving the implementation of system units and enhancing the system's services as new requirements are discovered.

In principle, the result of each phase is one or more documents that are approved (signed off). The following phase should not start until the previous phase has finished. In practice, these stages overlap and feed information to each other. During design, problems with requirements are identified; during coding design problems are found and so on. The software process is not a simple linear model but involves a sequence of iterations of the development activities. Because of the costs of producing and approving documents, iterations are costly and involve significant rework. Therefore, after a small number of iterations, it is normal to freeze parts of the development, such as the specification, and to continue with the later development stages. Problems are left for later resolution, ignored or programmed around. This premature freezing of requirements may mean that the system won't do what the user wants. It may also lead to badly structured systems as design problems are circumvented by implementation tricks. During the final life-cycle phase (operation and maintenance), the software is put into use. Errors and omissions in the original software requirements are discovered. Program and design error, emerge and the need for new functionality is identified. The system must therefore evolve to remain useful. Making these changes (software maintenance) may involve repeating previous process stages. The advantages of the waterfall model are that documentation is produced at each phase and that it fits with other engineering process models. Its major problem is its inflexible partitioning of the project into distinct stages. Commitments must be made at, an early stage in the process, which makes it difficult to respond to changing customer requirements.
Evolutionary development
Evolutionary development is based on the idea of developing an initial implementation, exposing this to user comment and refining it through many versions until an adequate system has been developed. Specification, development and validation activities are interleaved rather than separate, with rapid feedback across activities. There are two fundamental types of evolutionary development:

1. **Exploratory development** where the objective of the process is to work with the customer to explore their requirements and deliver a final system. The development starts with the parts of the system that are understood. The system evolves by adding new features proposed by the customer.

2. **Throwaway prototyping** where the objective of the evolutionary development process is to understand the customer’s requirements and hence develop a better requirements definition for the system. The prototype concentrates on experimenting with the customer requirements that are poorly understood.

An evolutionary approach to software development is often more effective than waterfall approaches in producing systems that meet the immediate needs of customers. The advantage of a software process that is based on an evolutionary approach is that the specification can be developed incrementally. As users develop a better understanding of their problem, this can be reflected in the software system. However, from an engineering and management perspective, the evolutionary approach has two problems:

1. **The process is not visible** Managers need regular deliverables to measure progress. If systems are developed quickly, it is not cost-effective to produce documents that reflect every version of the system.

2. **Systems are often poorly structured** Continual change tends to corrupt the software structure. Incorporating software changes becomes increasingly difficult and costly.

For small and medium-sized systems the evolutionary approach is the best approach to development. The problems of evolutionary development become particularly acute for large, complex, long-lifetime systems, where different teams develop different parts of the system. It is difficult to establish a stable system architecture using this approach, which makes it hard to integrate contributions from the teams.

For large systems, I recommend a mixed process that incorporates the best features of the waterfall and the evolutionary development models. This may involve developing a throwaway prototype using an evolutionary approach to resolve uncertainties in the system specification. You can then re-implement the system using a more structured approach. Parts of the system that are well understood can be specified and developed using a waterfall-based process. Other parts of the system, such as the user interface, which are difficult to specify in advance, should always be developed using an exploratory programming approach.
Component based software engineering

In the majority of software projects, there is some software reuse. This usually happens informally when people working on the project know of designs or code which is similar to that required. They look for these, modify them as needed and incorporate them into their system. In the evolutionary approach, reuse is often essential for rapid system development. This informal reuse takes place irrespective of the development process that is used. However, in the last few years, an approach to software development called component-based software engineering (CBSE), which relies on reuse, has emerged and is becoming increasingly used.

This reuse-oriented approach relies on a large base of reusable software components and some integrating framework for these components. Sometimes, these components are systems in their own right (COTS or commercial off-the-shelf systems) that may provide specific functionality such as text formatting or numeric calculation.

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

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

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

3. **System design with reuse** During this phase, the framework of the system is designed or an existing framework is reused. The designers take into account the components that are reused and organize the framework to cater to this. Some new software may have to be designed if reusable components are not available.

4. **Development and integration** Software that cannot be externally procured is developed, and the components and COTS systems are integrated to create the new system. System integration, in this model, may be part of the development process rather than a separate activity.
Component-based software engineering has the obvious advantage of reducing the amount of software to be developed and so reducing cost and risks. It usually also leads to faster delivery of the software. However, requirements compromises are inevitable and this may lead to a system that does not meet the real needs of users.

**Process iteration**

Change is inevitable in all large software projects. The system requirements change as the business procuring the system responds to external pressures. Management priorities change. As new technologies become available, designs and implementation change. This means that the software process is not a one-off process; rather, the process activities are regularly repeated as the system is reworked in response to change requests. The two process models that have been explicitly designed to support process iteration:

1. **Incremental delivery** The software specification, design and implementation are broken down into a series of increments that are each developed in turn.
2. **Spiral development** The development of the system spirals outwards from an initial outline through to the final developed system.

The essence of iterative processes is, that the specification is developed in conjunction with the software. However, this conflicts with the procurement model of many organizations where the complete system specification is part of the system development contract. In the incremental approach, there is no complete system specification until the final increment is specified. This requires a new form of contract, which large customers such as government agencies may find difficult to accommodate.

**Incremental delivery**

The waterfall model of development requires customers for a system to commit to a set of requirements before design begins and the designer to commit to particular design strategies before implementation. Changes to the requirements require rework of the requirements, design and implementation. However, the separation of design and implementation should lead to well-documented systems that are amenable to change. By contrast, an evolutionary approach to development allows requirements and design decisions to be delayed but also leads to software that may be poorly structured and difficult to understand and maintain. Incremental delivery is an in-between approach that combines the advantages of these models. In an incremental development process, customers identify, in outline, the services to be provided by the system. They identify which of the services are most important and which are least important to them. A number of
delivery increments are then defined, with each increment providing a sub-set of the system functionality. The allocation of services to increments depends on the service priority with the highest priority services delivered first. Once the system increments have been identified, the requirements for the services to be delivered in the first increment are defined in detail, and that increment is developed. During development, further requirements analysis for later increments can take place, but requirements changes for the current increment are not accepted. Once an increment is completed and delivered, customers can put it into service. This means that they take early delivery of part of the system functionality. They can experiment with the system that helps them clarify their requirements for later increments and for later versions of the current increment. As new increments are completed, they are integrated with existing increments so that the system functionality improves with each delivered increment. The common services may be implemented early in the process or may be implemented incrementally as functionality is required by an increment. This incremental development process has a number of advantages:

1. Customers do not have to wait until the entire system is delivered before they can gain value from it. The first increment satisfies their most critical requirements so they can use the software immediately.
2. Customers can use the early increments as prototypes and gain experience that informs their requirements for later system increments.
3. There is a lower risk of overall project failure. Although problems may be encountered in some increments, it is likely that some will be successfully delivered to the customer.
4. As the highest priority services are delivered first, and later increments are integrated with them, it is inevitable that the most important system services receive the most testing. This means that customers are less likely to encounter software failures in the most important parts of the system.

However, there are problems with incremental delivery. Increments should be relatively small, and each increment should deliver some system functionality. It can be difficult to map the customer’s requirements onto increments of the right size. Furthermore, most systems require a set of basic facilities that are used by different parts of the system. As requirements are not defined in detail until an increment is to be implemented, it can be hard to identify common facilities that are needed by all increments. A variant of this incremental approach called extreme programming. This is based around the development and delivery of very small increments of functionality, customer involvement in the process, constant code improvement and pair programming.
Spiral development
The spiral model of the software process was originally proposed by Boehm (Boehm, 1988). Rather than represent the software process as a sequence of activities with some backtracking from one activity to another, the process is represented as a spiral. Each loop in the spiral represents a phase of the software process. Thus, the innermost loop might be concerned with system feasibility, the next loop with requirements definition, the next loop with system designed so on. Each loop in the spiral is split into four sectors:

1. **Objective setting** Specific objectives for that phase of the project are defined. Constraints on the process and the product are identified and a detailed management plan is drawn up. Project risks are identified. Alternative strategies, depending on these risks, may be planned.
2. **Risk assessment and reduction** For each of the identified project risks, a detailed analysis is carried out. Steps are taken to reduce the risk. For example, if there is a risk that the requirements are inappropriate, a prototype system may be developed.
3. **Development and validation** After risk evaluation, a development model for the system is chosen. For example, if user interface risks are dominant, an appropriate development model might be evolutionary prototyping. If safety risks are the main consideration, development based on formal transformations may be the most appropriate and so on. The waterfall model may be the most appropriate development model if the main identified risk is sub-system integration.
4. **Planning** The project is reviewed and a decision made whether to continue with a further loop of the spiral. If it is decided to continue, plans are drawn up for the next phase of the project.

The main difference between the spiral model and other software process models is the explicit recognition of risk in the spiral model. Informally, risk simply means something that can go wrong. For example, if the intention is to use a new programming language, a risk is that the available compilers are unreliable or do not produce sufficiently efficient object code. Risks result in project problems such as schedule and cost overrun so risk minimisation is a very important project management activity.

A cycle of the spiral begins by elaborating objectives such as performance and functionality. Alternative ways of achieving these objectives and the constraints imposed on each of them are then enumerated. Each alternative is assessed against each objective and sources of project risk are identified. The next step is to resolve these risks by information-gathering activities such as more detailed analysis, prototyping and simulation. Once risks have been assessed, some development is carried out, followed by a planning activity for the next phase of the process.
Process activities
The four basic process activities of specification, development, validation and evolution are organized differently in different development processes. In the waterfall model, they are organized in sequence, whereas in evolutionary development they are interleaved. How these activities are carried out depends on the type of software, people and organizational structures involved. There is no right or wrong way to organize these activities and my goal in this section is simply to provide you with an introduction to how they can be organized.

Software specification
Software specification or requirements engineering is the process of understanding and defining what services are required from the system and identifying the constraints on the system's operation and development. Requirements engineering is a particularly critical stage of the software process as errors at this stage inevitably lead to later problems in the system design and implementation. This process leads to the production of a requirements document that is the specification for the system. Requirements are usually presented at two levels of detail in this document. End-users and customers need a high-level statement of the requirements; system developers need a more detailed system specification. There are four main phases in the requirements engineering process:

1. **Feasibility study**: An estimate is made of whether the identified user needs may be satisfied using current software and hardware technologies. The study considers whether the proposed system will be cost-effective from a business point of view and whether it can be developed within existing budgetary constraints. A feasibility study should be relatively cheap and quick. The result should inform the decision of whether to go ahead with a more detailed analysis.
2. **Requirements elicitation and analysis** This is the process of deriving the system requirements through observation of existing systems, discussions with potential users and procurers, task analysis and so on. This may involve the development of one or more system models and prototypes. These help the analyst understand the system to be specified.

3. **Requirements specification** The activity of translating the information gathered during the analysis activity into a document that defines a set of requirements. Two types of requirements may be included in this document. *User requirements* are abstract statements of the system requirements for the customer and end-user of the system; *system requirements* are a more detailed description of the functionality to be provided.

4. **Requirements validation** This activity checks the requirements for realism, consistency and completeness. During this process, errors in the requirements document are inevitably discovered. It must then be modified to correct these problems.

Of course, the activities in the requirements process are not simply carried out in a strict sequence. Requirements analysis continues during definition and specification, and new requirements come to light throughout the process. Therefore, the activities of analysis, definition and specification are interleaved. In agile methods such as extreme programming, requirements are developed incrementally according to user priorities, and the elicitation of requirements comes from users who are part of the development team.

**Software design and implementation**

The implementation stage of software development is the process of converting a system specification into an executable system. It always involves processes of software design and programming but, if an evolutionary approach to development is used, may also involve refinement of the software specification. A software design is a description of the structure of the software to be implemented, the data which is part of the system, the interfaces between system components and, sometimes, the algorithms used. Designers do not arrive at a finished design immediately but develop the design iteratively through a number of versions. The design process involves adding formality and detail as the design is developed with constant backtracking to correct earlier designs. The design process may involve developing several models of the system at different levels of abstraction. As a design is decomposed, errors and omissions in earlier stages are discovered. These feed back to allow earlier design models to be improved. The diagram below suggests that the stages of the design
process are sequential. In fact, design process activities are interleaved. Feedback from one stage to another and consequent design rework is inevitable in all design processes.

A specification for the next stage is the output of each design activity. This specification may be an abstract, formal specification that is produced to clarify the requirements, or it may be a specification of how part of the system is to be realized. As the design process continues, these specifications become more detailed. The final results of the process are precise specifications of the algorithms and data structures to be implemented. The specific design process activities are:

1. **Architectural design** The sub-systems making up the system and their relationships are identified and documented.
2. **Abstract specification** For each sub-system, an abstract specification of its services and the constraints under which it must operate is produced.
3. **Interface design** For each sub-system, its interface with other sub-systems is designed and documented. This interface specification must be unambiguous as it allows the sub-system to be used without knowledge of the sub-system operation.
4. **Component design** Services are allocated to components and the interfaces of these components are designed.
5. **Data structure design** The data structures used in the system implementation are designed in detail and specified.
6. **Algorithm design** The algorithms used to provide services are designed in detail and specified.

This is a general model of the design process and real, practical processes may adapt it in different ways. Possible adaptations are:

1. The last two stages of design-data structure and algorithm design may be delayed until the implementation process.
2. If an exploratory approach to design is used, the system interfaces may be designed after the data structures have been specified.
3. The abstract specification stage may be skipped, although it is usually an essential part of critical systems design.

Increasingly, where agile methods of development are used, the outputs of the design process will not be separate specification documents but will be represented in the code of the program. After the system architecture has been designed, later stages of the design are incremental. Each increment is
represented as program code rather than as a design model. A contrasting approach is taken by structured methods for design that rely on producing graphical models of the system and, in many cases, automatically generating code from these models.

**Software validation**

Software validation or, more generally, verification and validation (V & V) is intended to show that a system conforms to its specification and that the system meets the expectations of the customer buying the system. It involves checking processes, such as inspections and reviews at each stage of the software process from user requirements definition to program development. The majority of validation costs, however, are incurred after implementation when the operational system is tested. Except for small programs, systems should not be tested as a single, monolithic unit. Figure below shows a three-stage testing process where system components are tested, the integrated system is tested and, finally, the system is tested with the customer's data. Ideally, component defects are discovered early in the process and interface problems when the system is integrated. However, as defects are discovered the program must be debugged and this may require other stages in the testing process to be repeated. Errors in program components, say, may come to light during system testing. The process is therefore an iterative one with information being fed back from later stages to earlier parts of the process. The stages in the testing process are:

1. **Component (or unit) testing**: Individual components are tested to ensure that they operate correctly. Each component is tested independently, without other system components. Components may be simple entities such as functions or object classes, or may be coherent groupings of these entities.

2. **System testing**: The components are integrated to make up the system. This process is concerned with finding errors that result from unanticipated interactions between components and component interface problems. It is also concerned with validating that the system meets its functional and non-functional requirements and testing the emergent system properties. For large systems, this may be a multistage process where components are integrated to form sub-systems that are individually tested before they are themselves integrated to form the final system.

3. **Acceptance testing** This is the final stage in the testing process before the system is accepted for operational use. The system is tested with data supplied by the system customer rather than with simulated test data. Acceptance testing may reveal errors and omissions in the system requirements definition because the real data exercise the system in different ways from the test data. Acceptance testing may also reveal requirements problems where the system's facilities do not really meet the user's needs or the system performance is unacceptable.
Normally, component development and testing are interleaved. Programmers make up their own test data and incrementally test the code as it is developed. This is an economically sensible approach, as the programmer knows the component best and is therefore the best person to generate test cases. If an incremental approach to development is used, each increment should be tested as it is developed, with these tests based on the requirements for that increment. In extreme programming, tests are developed along with the requirements before development starts. This helps the testers and developers to understand the requirements and ensures that there are no delays as test cases are created. Later stages of testing involve integrating work from a number of programmers and must be planned in advance. An independent team of testers should work from preformulated test plans that are developed from the system specification and design. Acceptance testing is sometimes called alpha testing. Custom systems are developed for a single client. The alpha testing process continues until the system developer and the client agree that the delivered system is an acceptable implementation of the system requirements. When a system is to be marketed as a software product, a testing process called beta testing is often used. Beta testing involves delivering a system to a number of potential customers who agree to use that system. They report problems to the system developers. This exposes the product to real use and detects errors that may not have been anticipated by the system builders. After this feedback, the system is modified and released either for further beta testing or for general sale.
Software evolution

The flexibility of software systems is one of the main reasons why more and more software is being incorporated in large, complex systems. Once a decision has been made to procure hardware, it is very expensive to make changes to the hardware design. However, changes can be made to software at any time during or after the system development. Even extensive changes are still much cheaper than corresponding changes to system hardware.

Historically, there has always been a split between the process of software development and the process of software evolution (software maintenance). People think of software development as a creative activity where a software system was developed from an initial concept through to a working system. However, they sometimes think of software maintenance as dull and uninteresting. Although the costs of maintenance are often several times the initial development costs, maintenance processes are sometimes considered to be less challenging than original software development. This distinction between development and maintenance is becoming increasingly irrelevant. Few software systems are now completely new systems, and it makes much more sense to see development and maintenance as a continuum. Rather than two separate processes, it is more realistic to think of software engineering as an evolutionary process where software is continually changed over its lifetime in response to changing requirements and customer needs.