**Software Engineering**

**What is software?**

* Computer programs and associated documentation such as requirements, design models and user manuals.
* Software products may be developed for a particular customer or may be developed for a general market.
* Software products may be
  + Generic - developed to be sold to a range of different customers

E.g. PC software such as Excel or Word.

* + Bespoke (custom) - developed for a single customer according to their specification.
* New software can be created by developing new programs, configuring generic software systems or reusing existing software.

**What is software engineering?**

* Software engineering is an engineering discipline that is concerned with all aspects of software production.
* Software engineers should adopt a systematic and organized approach to their work and use appropriate tools and techniques depending on the problem to be solved, the development constraints and the resources available.

**What is the difference between software engineering and computer science?**

* Computer science is concerned with theory and fundamentals; software engineering is concerned with the practicalities of developing and delivering useful software.
* Computer science theories are still insufficient to act as a complete underpinning for software engineering (unlike e.g. physics and electrical engineering).

**What is the difference between software engineering and system engineering?**

* System engineering is concerned with all aspects of computer-based systems development including hardware, software and process engineering. Software engineering is part of this process concerned with developing the software infrastructure, control, applications and databases in the system.
* System engineers are involved in system specification, architectural design, integration and deployment.

**What is a software process?**

* A set of activities whose goal is the development or evolution of software.
* Generic activities in all software processes are:

• Specification - what the system should do and its development constraints.

• Development - production of the software system.

• Validation - checking that the software is what the customer wants.

• Evolution - changing the software in response to changing demands.

**System modeling**

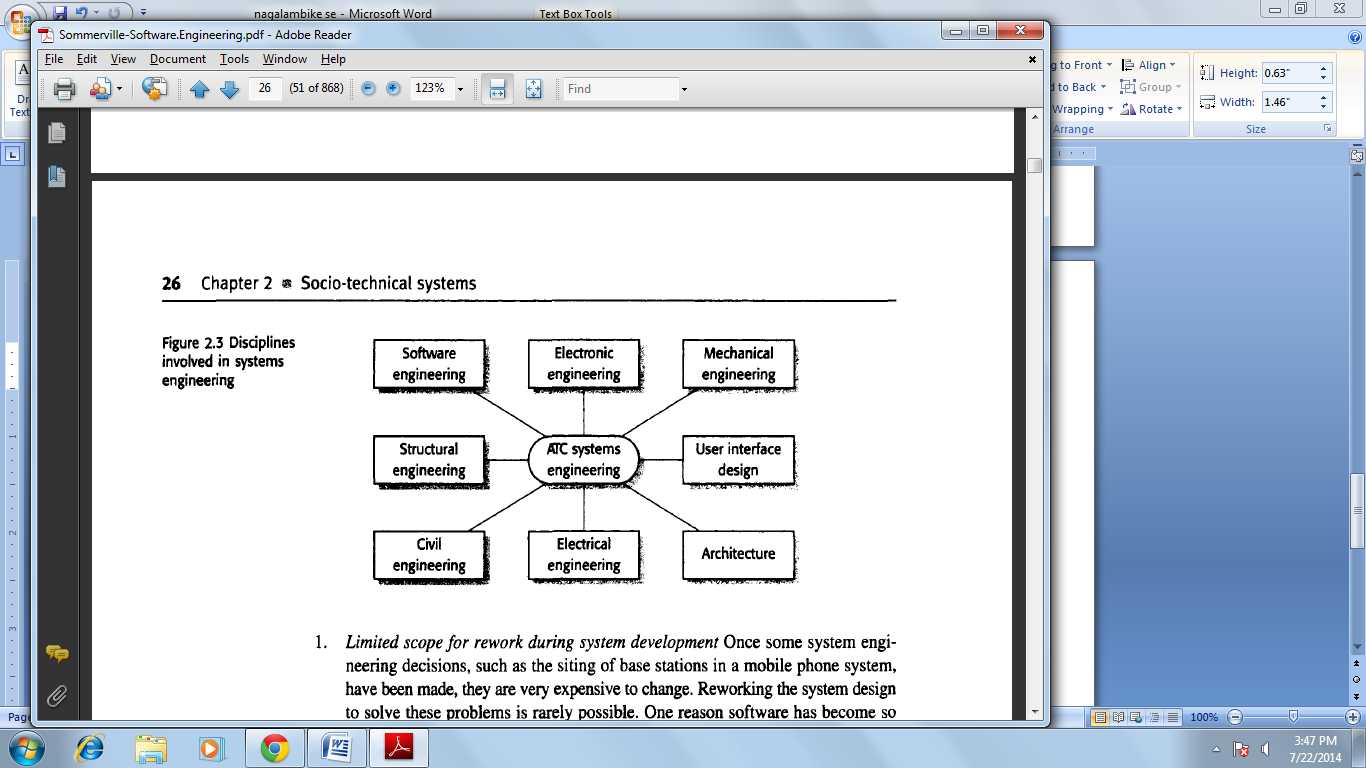
**Systems modeling** or **system modeling** is the [interdisciplinary study](http://en.wikipedia.org/wiki/Interdisciplinarity) of the use of [models](http://en.wikipedia.org/wiki/Scientific_modeling) to conceptualize and construct [systems](http://en.wikipedia.org/wiki/System) in [business](http://en.wikipedia.org/wiki/Business) and [IT development](http://en.wikipedia.org/wiki/Information_technology).

* System modeling is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system.
* System modeling has now come to mean representing a system using some kind of graphical notation, which is now almost always based on notations in the Unified Modeling Language (UML).
* System modeling helps the analyst to understand the functionality of the system and models are used to communicate with customers.

**Systems engineering**

Systems engineering is the activity of specifying, designing, implementing, validating, deploying and maintaining socio-technical systems. Systems engineers are not just concerned with software but also with hardware and the system's interactions with users and its environment. They must think about the services that the system provides the constraints under which the system must be built and operated and the ways in which the system is used to fulfil its purpose. As I have discussed, software engineers need an understanding of system engineering because problems of software engineering are often a result of system engineering decisions.

**Disciplines involved in systems engineering**



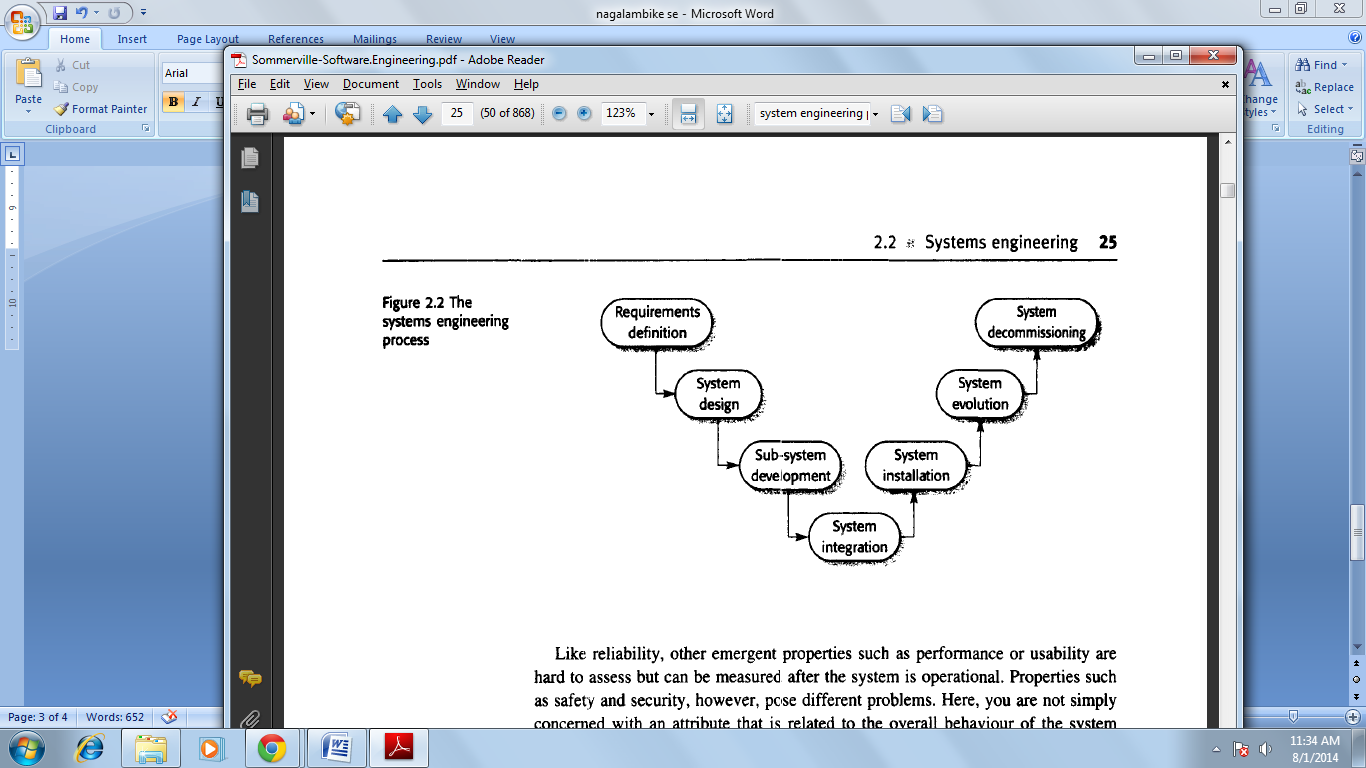
There are important distinctions between the system engineering process and the

Software development process:

1. ***Limited scope for rework during system development***Once some system engineering decisions, such as the siting of base stations in a mobile phone system, have been made, they are very expensive to change. Reworking the system design to solve these problems is rarely possible. One reason software has become so important in systems is that it allows changes to be made during system development, in response to new requirements.
2. ***Interdisciplinary involvement***Many engineering disciplines may be involved in system engineering. There is a lot of scope for misunderstanding because different engineers use different terminology and conventions.

**Systems engineering Process**

Systems engineering is the activity of specifying, designing, implementing, validating, deploying and maintaining socio-technical systems. Systems engineers are not just concerned with software but also with hardware and the system's interactions with users and its environment.



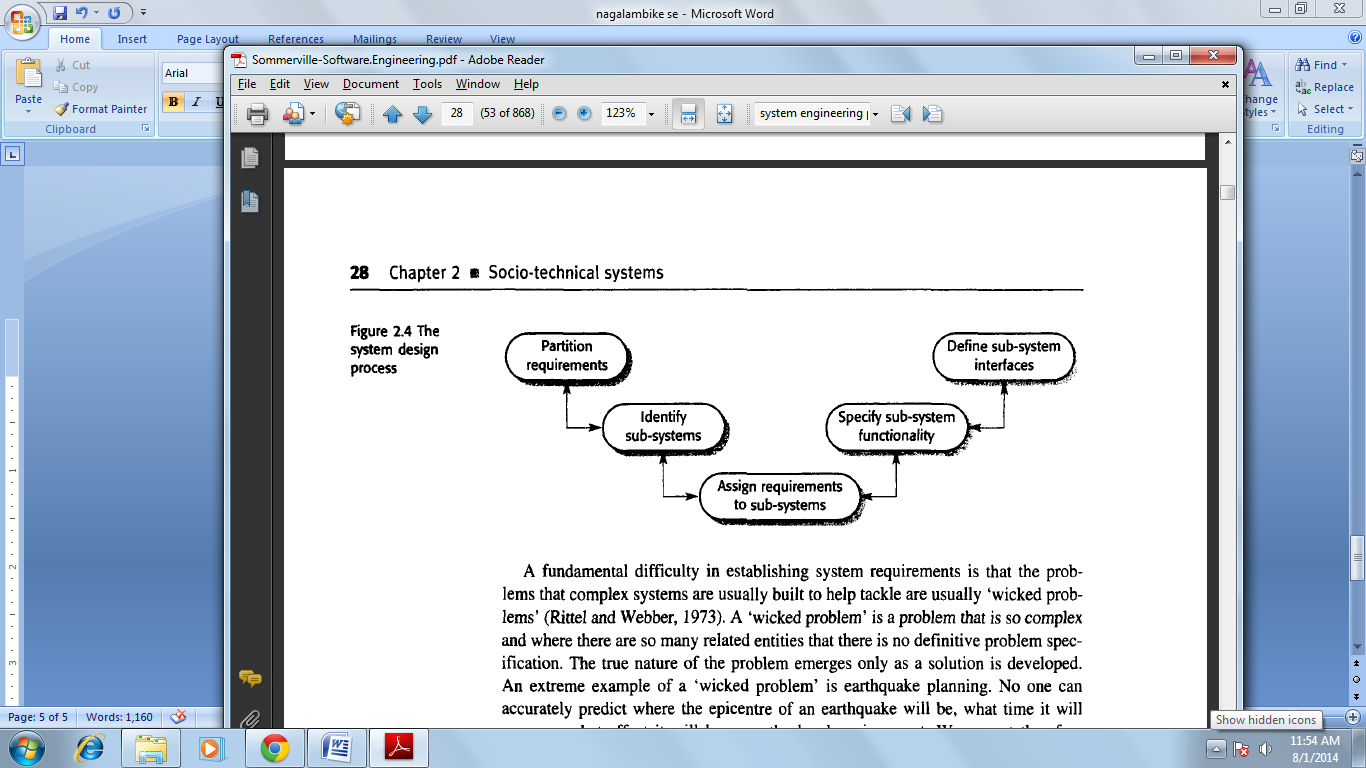
**System requirements definition**

System requirements definitions specify what the system should do (its functions) and its essential and desirable system properties. As with software requirements analysis creating system requirements definitions involves consultations with system customers and end-users.

This requirements definition phase usually concentrates on deriving three types of requirement:

1. ***Abstract functional requirements***The basic functions that the system must provide are defined at an abstract level. More detailed functional requirements specification takes place at the sub-system level. For example, in an air traffic control system, an abstract functional requirement would specify that a flight-plan database should be used to store the flight plans of all aircraft entering the controlled airspace. However, you would not normally specify the details of the database unless they affected the requirements of other sub-systems.
2. ***System properties***These are non-functional emergent system properties such as availability, performance and safety, as I have discussed above. These nonfunctional system properties affect the requirements for all sub-systems.
3. ***Characteristics that the system must not exhibit***Itis sometimes as important to specify what the system must notdo as it is to specify what the system should do. For example, if you are specifying an air traffic control system, you might specify that the system should not present the controller with too much information.

**System design**

****

System design is concerned with how the system functionality is to be provided by the components of the system. The activities involved in this process are:

1. *Partition requirements* You analyse the requirements and organise them into related groups. There are usually several possible partitioning options, and you may suggest a number of alternatives at this stage of the process.
2. *Identify sub-systems* You should identify sub-systems that can individually or collectively meet the requirements. Groups of requirements are usually related to sub-systems, so this activity and requirements partitioning may be amalgamated. However, sub-system identification may also be influenced by other organizational or environmental factors.
3. *Assign requirements to sub-systems* You assign the requirements to subsystems. In principle, this should be straightforward if the requirements partitioning is used to drive the sub-system identification. In practice, there is never a clean match between requirements partitions and identified sub-systems. Limitations of externally purchased sub-systems may mean that you have to change the requirements to accommodate these constraints.
4. *Specify sub-system functionality* You should specify the specific functions provided by each sub-system. This may be seen as part of the system design phase or, if the sub-system is a software system, part of the requirements specification activity for that system. You should also try to identify relationships between sub-systems at this stage.
5. *Define sub-system interfaces* You define the interfaces that are provided and required by each sub-system. Once these interfaces have been agreed upon, it becomes possible to develop these sub-systems in parallel.

**Sub-system development**

During sub-system development, the sub-systems identified during system design are implemented. This may involve starting another system engineering process for individual sub-systems or, if the sub-system is software, a software process involving requirements, design, implementation and testing.

Occasionally, all sub-systems are developed from scratch during the development process. Normally, however, some of the sub-systems are commercial, off-the-shelf (COTS) systems that are bought for integration into the system. It is usually much cheaper to buy existing products than to develop special-purpose components. At this stage, you may have to reenter the design activity to accommodate a bought-in component. COTS systems may not meet the requirements exactly but, if off-the-shelf products are available, it is usually worth the expense of rethinking the design.

Sub-systems are usually developed in parallel. When problems are encountered that cut across sub-system boundaries, a system modification request must be made. Where systems involve extensive hardware engineering, making modifications after manufacturing has started is usually very expensive. Often ‘work-arounds’ that compensate for the problem must be found. These ‘work-arounds’ usually involve software changes because of the software's inherent flexibility. This leads to changes in the software requirements so, as I have discussed in Chapter I, it is important to design software for change so that the new requirements can be implemented without excessive additional costs.

**Systems integration**

During the systems integration process, you take the independently developed sub-systems and put them together to make up a complete system. Integration can be done suing a ‘big bang’ approach, where all the sub-systems are integrated at the same time. However, for technical and managed purposes, an incremental integration process where sub-systems are integrated one at a time is the best approach, for two reasons:

1. It is usually impossible to schedule the development of all the sub-systems so that they are all finished at the same time.
2. Incremental integration reduces the cost of error location. If many sub-systems are simultaneously integrated, an error that arises during testing may be in any of these sub-systems. When a single sub-system is integrated with an already working system, errors that occur are probably in the newly integrated sub-system or in the interaction between the existing subsystems and the new sub-system.

**System evolution**

Large, complex systems have a very long lifetime. During their life, they are changed to correct errors in the original system requirements and to implement new requirements that have emerged. The system’s computers are likely to be replaced with new, faster machines. The organization that uses the system may reorganize itself and hence use the system in a different way. The external environment of the system may change, forcing changes to the system.

System evolution, like software evolution, is inherently costly for several reasons:

1. Proposed changes have to be analysed very carefully from a business and a technical perspective. Changes have to contribute to the goals of the system and should not simply be technically motivated.
2. Because sub-systems are never completely independent, changes to one subsystem may adversely affect the performance or behaviour of other subsystems. Consequent changes to these sub-systems may therefore be needed.
3. The reasons for original design decisions are often unrecorded. Those responsible for the system evolution have to work out why particular design decisions were made.
4. As systems age, their structure typically becomes corrupted by change so the costs of making further changes increases.

**System decommissioning**

System decommissioning means taking the system out of service after the end of its useful operational lifetime. For hardware systems this may involve disassembling and recycling materials or dealing with toxic substances. Software has no physical decommissioning problems, but some software may be incorporated in a system to assist with the decommissioning process. For example, software may be used to monitor the state of hardware components. When the system is decommissioned, components that are not worn can therefore be identified and reused in other systems.

If the data in the system that is being decommissioned is still valuable to your organisation, you may have to convert it for use by some other system. This can often involve significant costs as the data structures may be implicitly defined in the software itself. You have to analyse the software to discover how the data is structured and then write a program to reorganise the data into the required structures for the new system.

**System Procurement**

It is the act of buying goods and services. [**Acquisition**](http://www.sebokwiki.org/wiki/Acquisition_(glossary)) covers the conceptualization, initiation, design, development, testing, contracting, production, deployment, logistics support, modification, and disposal of weapons and other systems, as well as supplies or services (including construction) to satisfy organizational needs intended for use in, or in support of, defined missions

 It is the [acquisition](http://en.wikipedia.org/wiki/Acquisition) of goods, services or works from an outside external source. It is favourable that the goods, services or works are appropriate and that they are procured at the best possible [cost](http://en.wikipedia.org/wiki/Total_cost_of_ownership) to meet the needs of the purchaser in terms of quality and quantity, time, and location. Corporations and public bodies often define processes intended to promote fair and open competition for their business while minimizing exposure to fraud and collusion.

Almost all purchasing decisions include factors such as delivery and handling, marginal benefit, and price fluctuations. Procurement generally involves making buying decisions under conditions of [scarcity](http://en.wikipedia.org/wiki/Scarcity). If good data is available, it is good practice to make use of economic analysis methods such as [cost-benefit analysis](http://en.wikipedia.org/wiki/Cost-benefit_analysis) or [cost-utility analysis](http://en.wikipedia.org/wiki/Cost-utility_analysis).

**Software processes**

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.

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**

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. In this section, I introduce a number of very general process models (sometimes called *process paradigms)* and present these from an architectural perspective. That is, we see the framework of the process but not the details of specific activities.

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 that I cover here are:

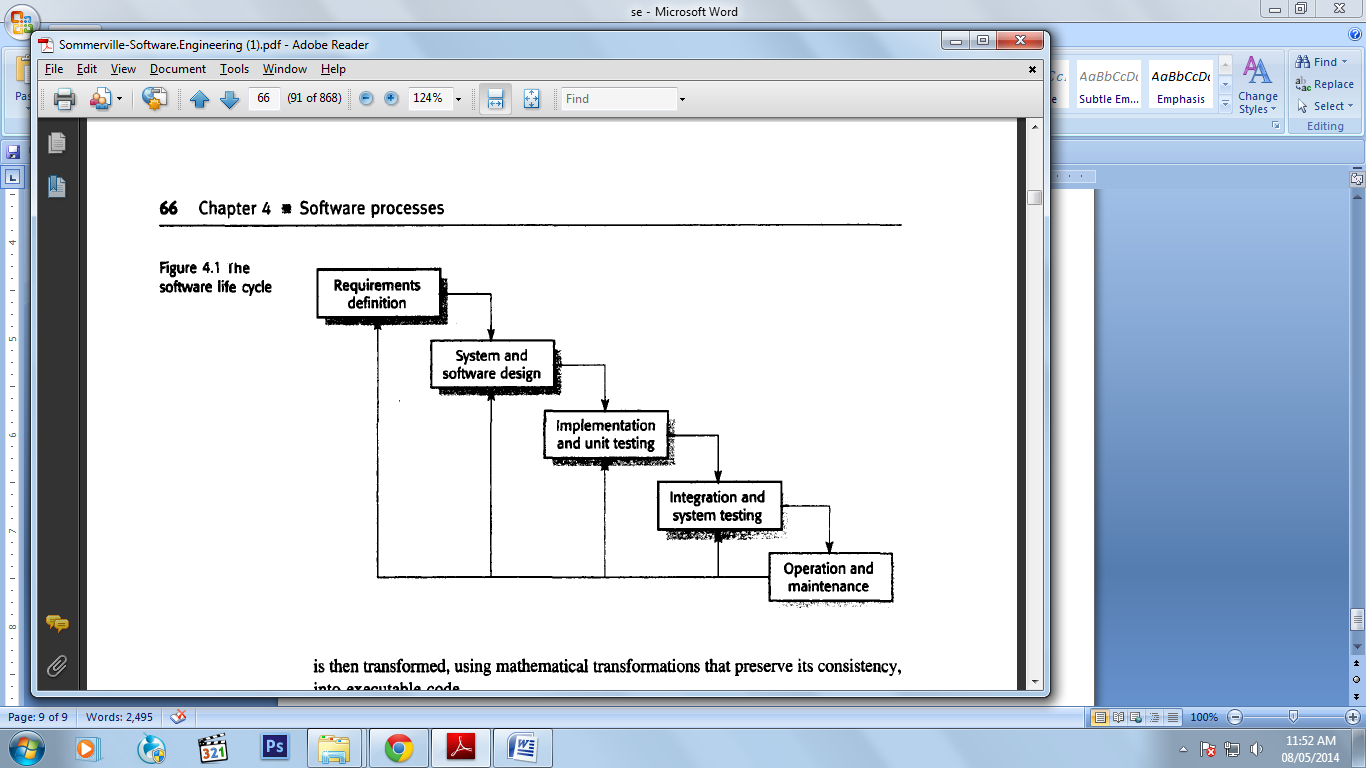
*1.* ***The waterfall model***This takes the fundamental process activities of specification, development, validation and evolution and represents them as separate process phases such as requirements specification, software design, implementation, testing and so on.

*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.

1. **The waterfall model**

The first published model of the software development process was derived from more general system engineering processes (Royce, 1970). 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 an overall system architecture. Software design involves identifying and describing the fundamental software system abstractions and their relationships.

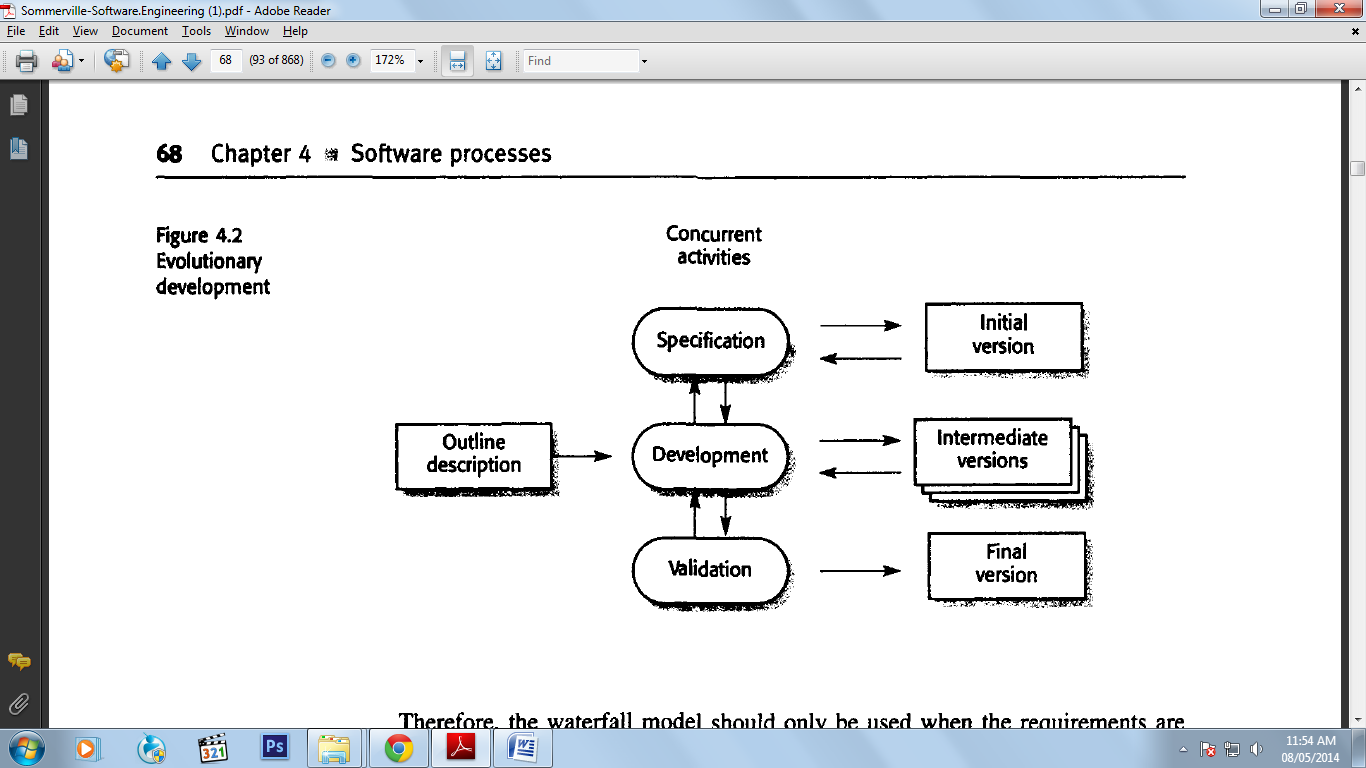
*3.* ***Implementation and unit testing***During this stage, the software design is realised as a set of programs or program units. Unit testing involves verifying that each unit meets its specification.

*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.

1. **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 (Figure 4.2). 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.

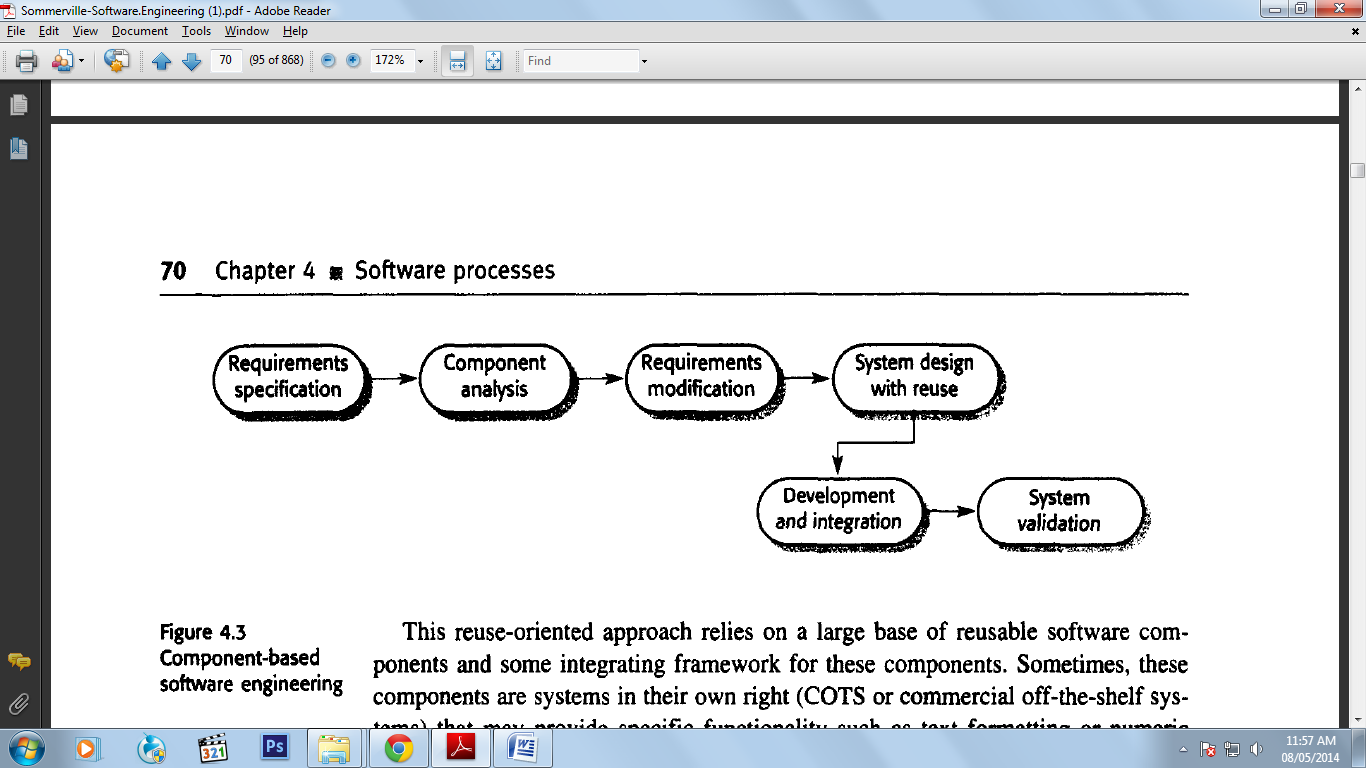
However, from an engineering and management perspective, the evolutionary approach has two problems:

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.

**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.



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 analysed 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 organise the framework to cater to this.

Some new software may have to be designed if reusable components are not available.

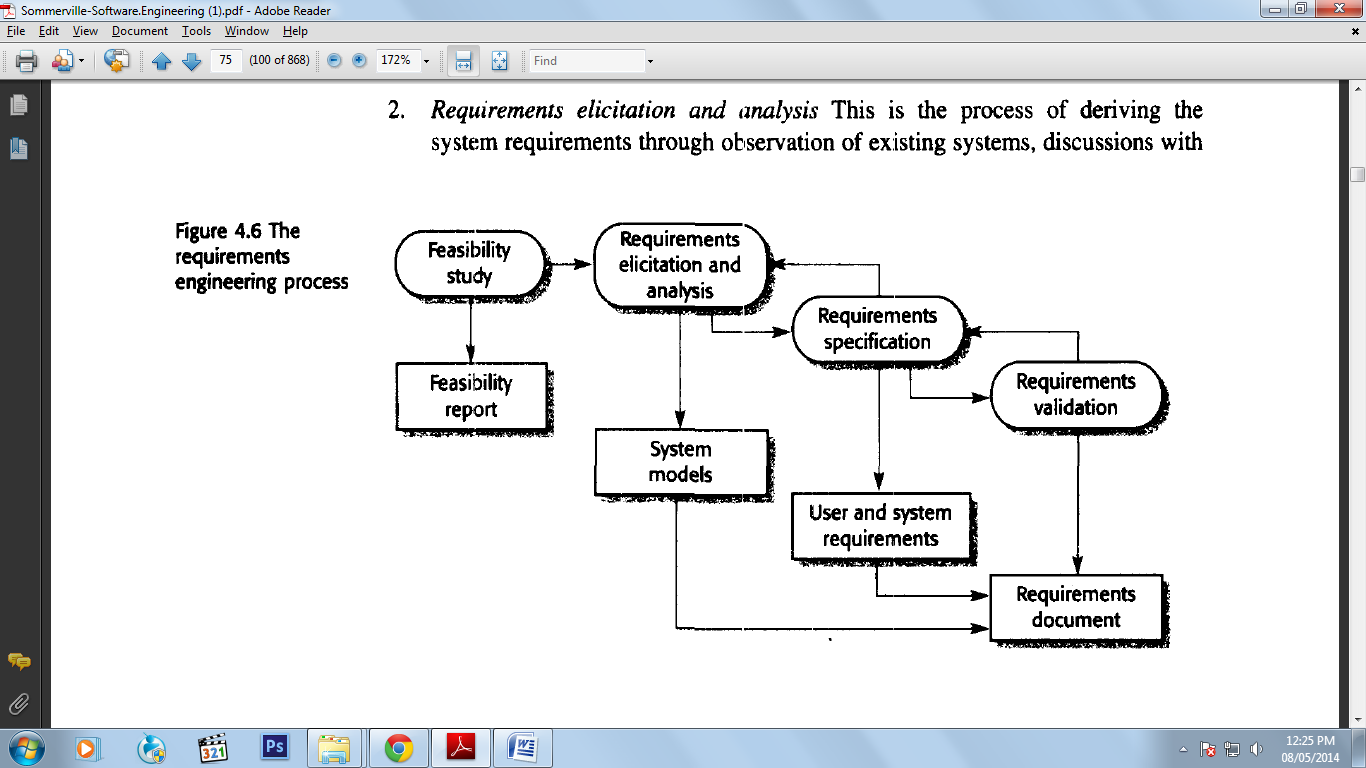
*4. Development and integration* Software that cannot be externally procured is developed, and the components and COTS systems are integrated to create the new system. System integration, in this model, may be part of the development process rather than a separate activity.

**Process activities**

The four basic process activities of specification, development. Validation and evolution are organised differently in different development processes. In the waterfall model, they are organised in sequence, whereas in evolutionary development they are interleaved. How these activities are carried out depends on the type of software, people and organisational structures involved. There is no right or wrong way to organise these activities and my goal in this section is simply to provide you with an introduction to how they can be organised.

**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.



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.

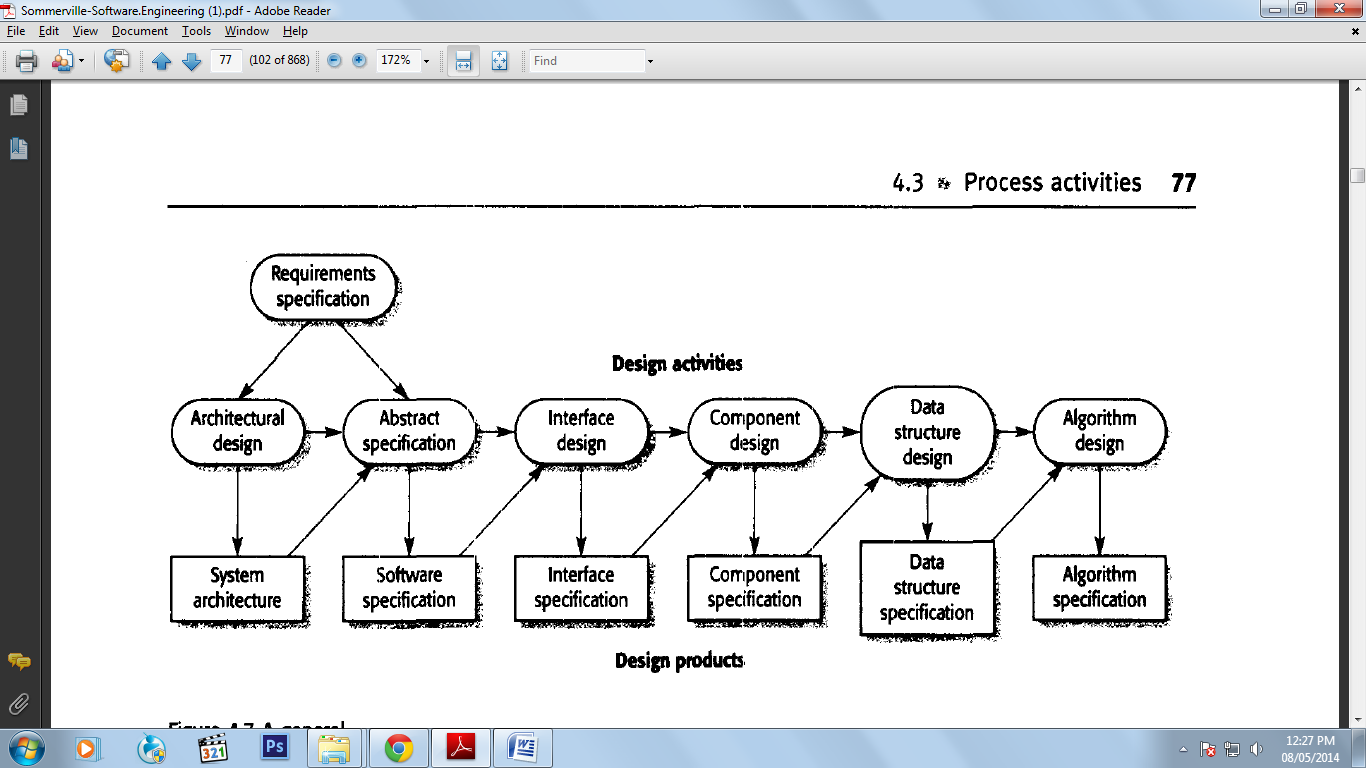
*3.* ***Requirements specification***The activity of translating the information gathered during the analysis activity into a document that defines a set of requirements. Two types of requirements may be included in this document. *User requirements* are abstract statements of the system requirements 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.

**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 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 specil1ed.

*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.

A structured method includes a design process model, notations to represent the design, report formats, rules and design guidelines. Structured methods may support some or all of the following models of a system:

1. An object model that shows the object classes used in the system and their dependencies.

2. A sequence model that shows how objects in the system interact when the system is executing.

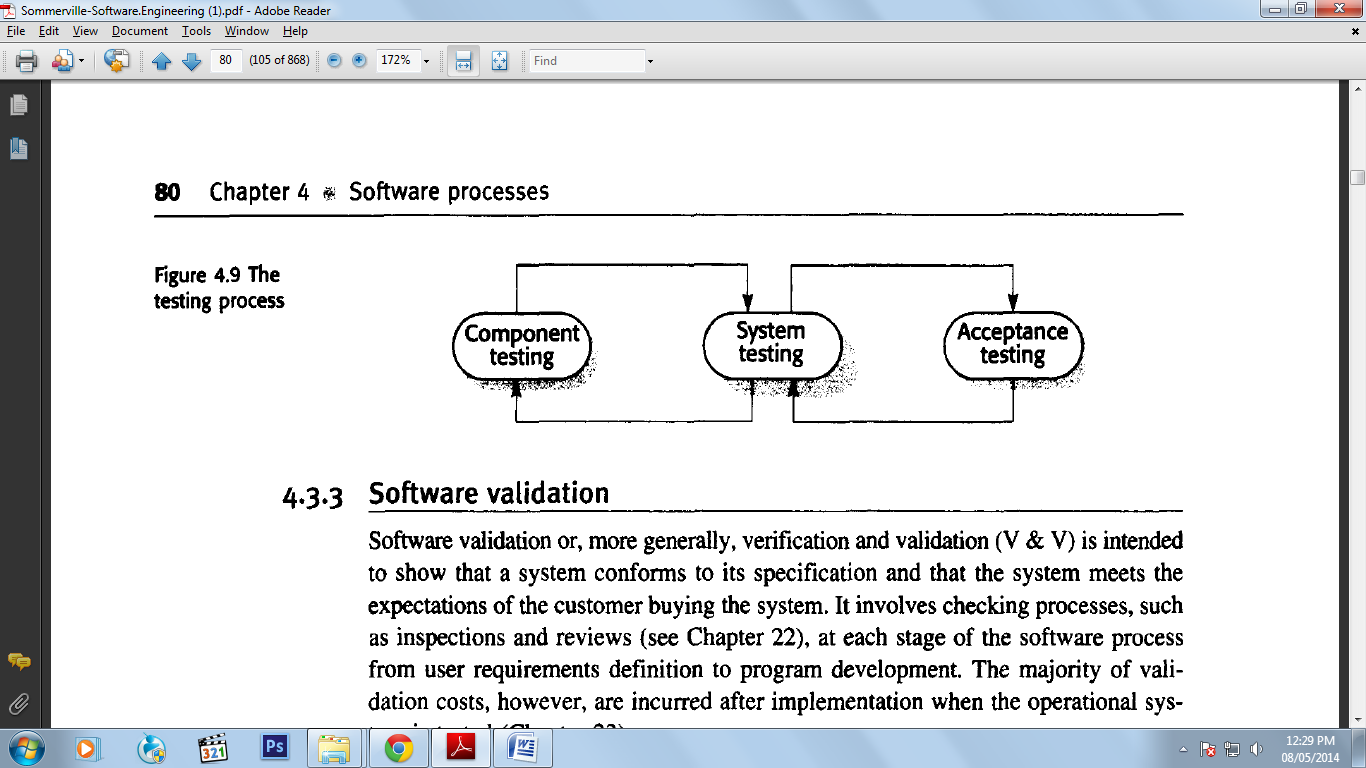
3. A state transition model that shows system states and the triggers for the transitions from one state to another.

4. A structural model where the system components and their aggregations are documented.

5. A data flow model where the system is modelled using the data transformations that take place as it is processed. This is not normally used in object-oriented methods but is still frequently used in real-time and business system design.

**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.



The stages in the testing process are:

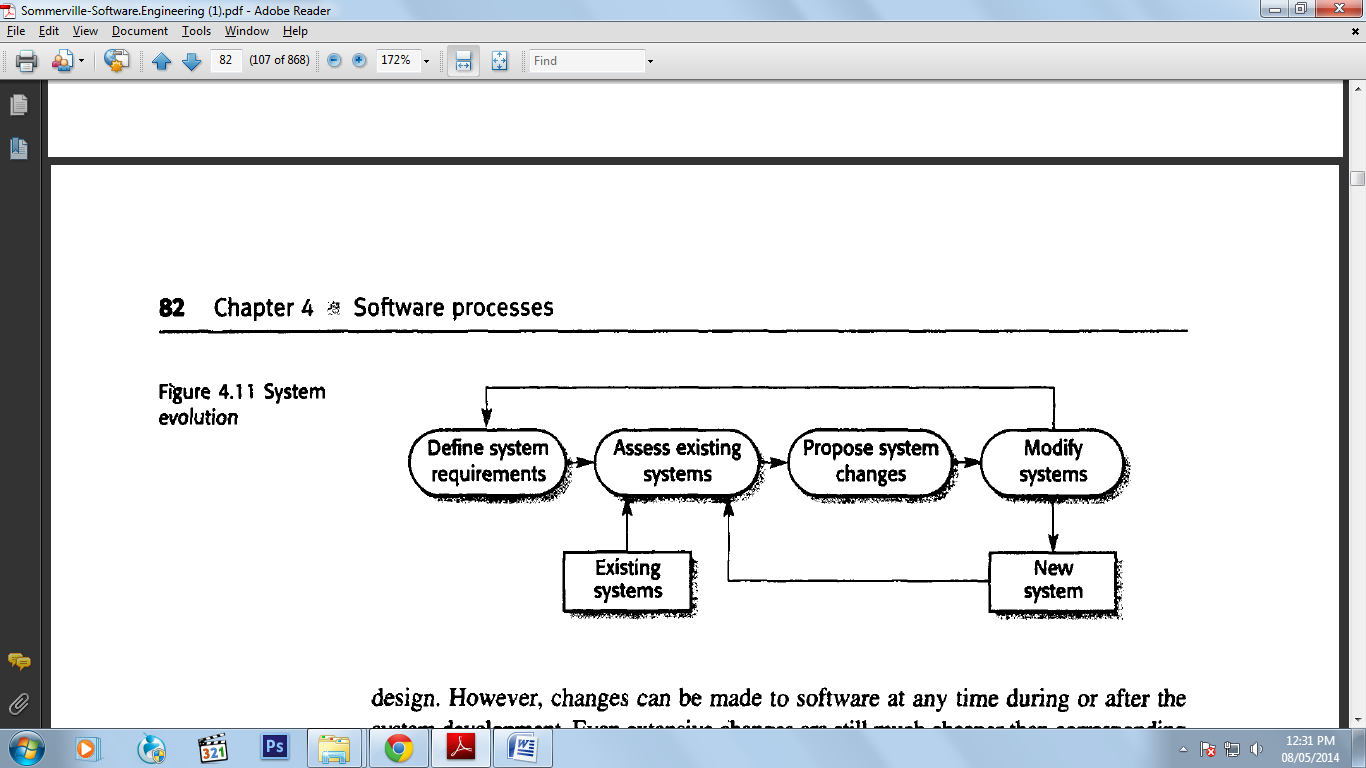
*1. Component (or unit) testing* Individual components are tested to ensure that they operate correctly. Each component is tested independently, without other system components. Components may be simple entities such as functions or object classes, or may be coherent groupings of these entities.

*2. System testing* The components is 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.

**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.

**Requirement Engineering**

Perhaps the major problem that we face in developing large and complex software systems is that of requirements engineering. Requirements engineering is concerned with establishing what the system should do, it’s desired and essential emergent properties, and the constraints on system operation and the software development processes. You can therefore think of requirements engineering as the communications process between the software customers and users and the software developers.

Requirements engineering is not simply technical process. The system requirements are influenced by users' likes, dislikes and prejudices, and by political and organisational issues.

Some of the problems that arise during the requirements engineering process are a result of failing to make a clear separation between these different levels of description. I distinguish between them by using the term *user requirements* to mean the high-level abstract requirements and *system requirements* to mean the detailed description of what the system should do. User requirements and system requirements may be defined as follows:

*1.* ***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.

*2****. System requirements***set out the system's functions, services and operational constraints in detail. The system requirements document (sometimes called a functional specification) should be precise. It should define exactly what is to be implemented. It may be part of the contract between the system buyer and the software developers.

**Functional and non-functional requirements**

Software system requirements are often classified as functional requirements, non-functional requirements or domain requirements:

*I. Functional requirements* These are statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations. In some cases, the functional requirements may also explicitly state what the system should not do.

*2. Non-functional requirements* These are constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process and standards. Non-functional requirements often apply to the system as a whole. They do not usually just apply to individual system features or services.

*3. Domain requirements* These are requirements that come from the application domain of the system and that reflect characteristics and constraints of that domain. They may be functional or non-functional requirements

**Functional requirements**

The functional requirements for a system describe what the system should do. These requirements depend on the type of software being developed, the expected users of the software and the general approach taken by the organisation when writing requirements. When expressed as user requirements, the requirements are usually described in a fairly abstract way. However, functional system requirements describe the system function in detail, its inputs and outputs, exceptions, and so on.

Functional requirements for a software system may be expressed in a number of ways. For example, here are examples of functional requirements for a university library system called L:"BSYS, used by students and faculty to order books and documents from other libraries.

1. The user shall be able to search either all of the initial set of databases or select a subset from it.

2. The system shall provide appropriate viewers for the user to read documents in the document store.

3. Every order shall be allocated a unique identifier (ORDER\_ID), which the user shall be able to copy to the account's permanent storage area.

**Non-functional requirements**

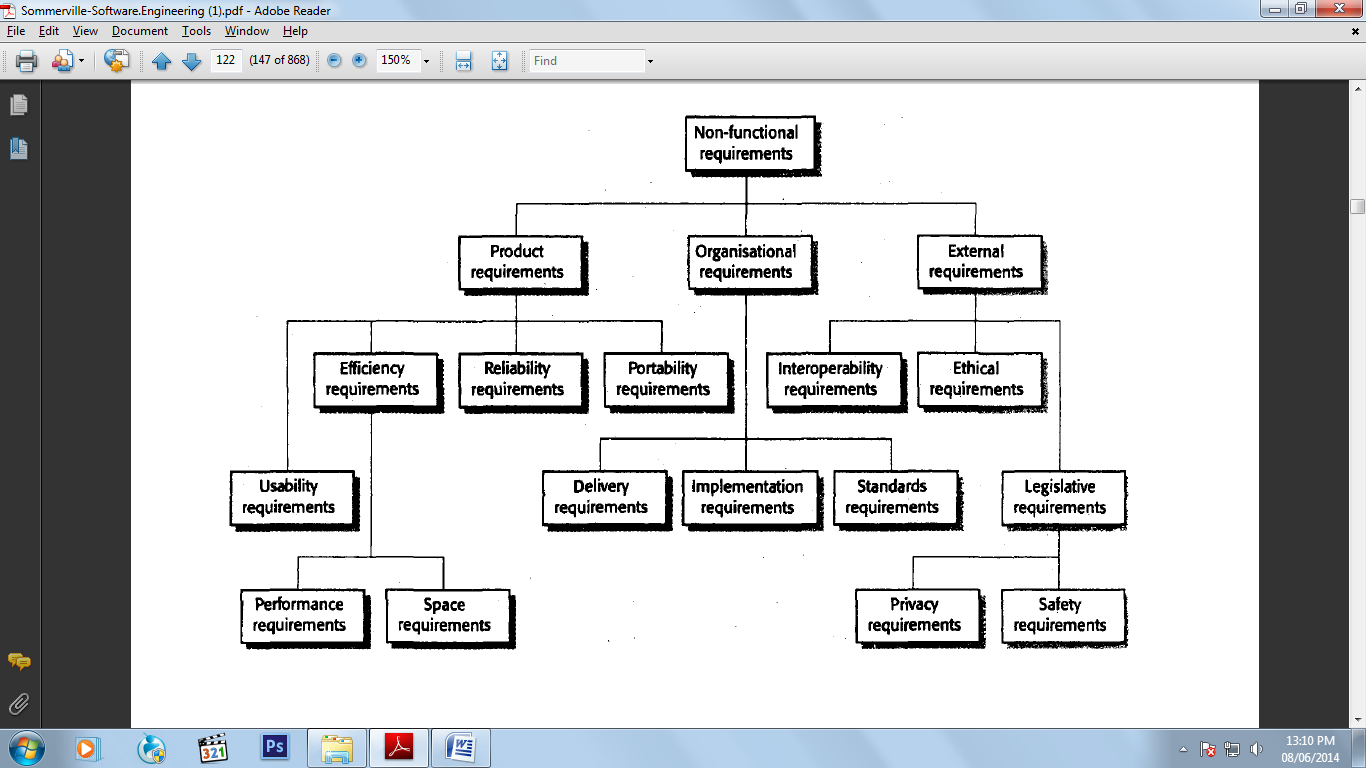
Non-functional requirements, as the name suggests, are requirements that are not directly concerned with the specific functions delivered by the system. They may relate to emergent system properties such as reliability, response time and store occupancy. Alternatively, they may defll1e constraints on the system such as the capabilities of I/Odevices and the data representations used in system interfaces.

The types of non-functional requirements are:

1. ***Product requirements***These~ requirements specify product behaviour. Examples include performance requirements on how fast the system must execute and how much memory it requires; reliability requirements that set out the acceptable failure rate; portability requirements; and usability requirements.

*2.* ***Organisational requirements***These requirements are derived from policies and procedures in the customer s and developer s organisation. Examples include process standards that must be used; implementation requirements such as the programming language or design method used; and delivery requirements that specify when the product and its documentation are to be delivered.

*3.* ***External requirements***This broad heading covers all requirements that are derived from factors external to the system and its development process. These may include interoperability requirements that define how the system interacts with systems in other organisations: legislative requirements that must be followed to ensure that the system operates within the law; and ethical requirements. Ethical requirements are requirements placed on a system to ensure that it will be acceptable to its users and the general public.



**User requirements**

The user requirements for a system should describe the functional and nonfunctional requirements so that they are understandable by system users without detailed technical knowledge. They should only specify the external behaviour of the system and should avoid, as far as possible, system design characteristics. Consequently, if you are writing user requirements, you should not use software jargon, structured notations or formal notations, or describe the requirement by describing the system implementation. You should write user requirements in simple language, with simple tables and forms and intuitive diagrams.

However, various problems can arise when requirements are written in natural language sentences in a text document:

1. ***Lack of clarity***It is sometimes difficult to use language in a precise and unambiguous way without making the document wordy and difficult to read.

*2.* ***Requirements confusion***Functional requirements, non-functional requirements, system goals and design information may not be clearly distinguished.

*3.* ***Requirements amalgamation***Several different requirements may be expressed together as a single requirement.

The first sentence mixes up three kinds of requirements.

1. A conceptual, functional requirement states that the editing system should provide a grid. It presents a rationale for this.

2. A non-functional requirement giving detailed information about the grid units (centimeters or inches).

3. A non-functional user interface requirement that defines how the grid is switched on and off by the user.

**System requirements**

System requirements are expanded versions of the user requirements that are used by software engineers as the starting point for the system design. They add detail and explain how the user requirements should be provided by the system. They may be used as part of the contract for the implementation of the system and should therefore be a complete and consistent specification of the whole system.

Ideally, the system requirements should simply describe the external behavior of the system and its operational constraints. They should not be concerned with how the system should be designed or implemented. However, at the level of detail required to completely specify a complex software system, it is impossible, in practice, to exclude all design information.

There are several reasons for this:

1. You may have to design an initial architecture of the system to help structure the requirements specification. The system requirements are organised according to the different sub-systems that make up the system.

2. In most cases, systems must interoperate with other existing systems. These constrain the design, and these constraints impose requirements on the new system.

3. The use of a specific architecture to satisfy non-functional requirements may be necessary. An external regulator who needs to certify that the system is safe may specify that an architectural design that has already been certified be used.

**Structured language specifications**

Structured natural language is a way of writing system requirements where the freedom of the requirements writer is limited and all requirements are written in a standard way. The advantage of this approach is that it maintains most of the expressiveness and understandability of natural language but ensures that some degree of uniformity is imposed on the specification. Structured language notations limit the terminology that can be used and use templates to specify system requirements.

They may incorporate control constructs derived from programming languages and graphical highlighting to partition the specification.

When a standard form is used for specifying functional requirements, the following information should be included:

1. Description of the function or entity being specified

2. Description of its inputs and where these come from

3. Description of its outputs and where these go to

4. Indication of what other entities are used (the *requires* part)

5. Description of the action to be taken

6. If a functional approach is used, a pre-condition setting out what must be true before the function is called and a post-condition specifying what is true after the function is called

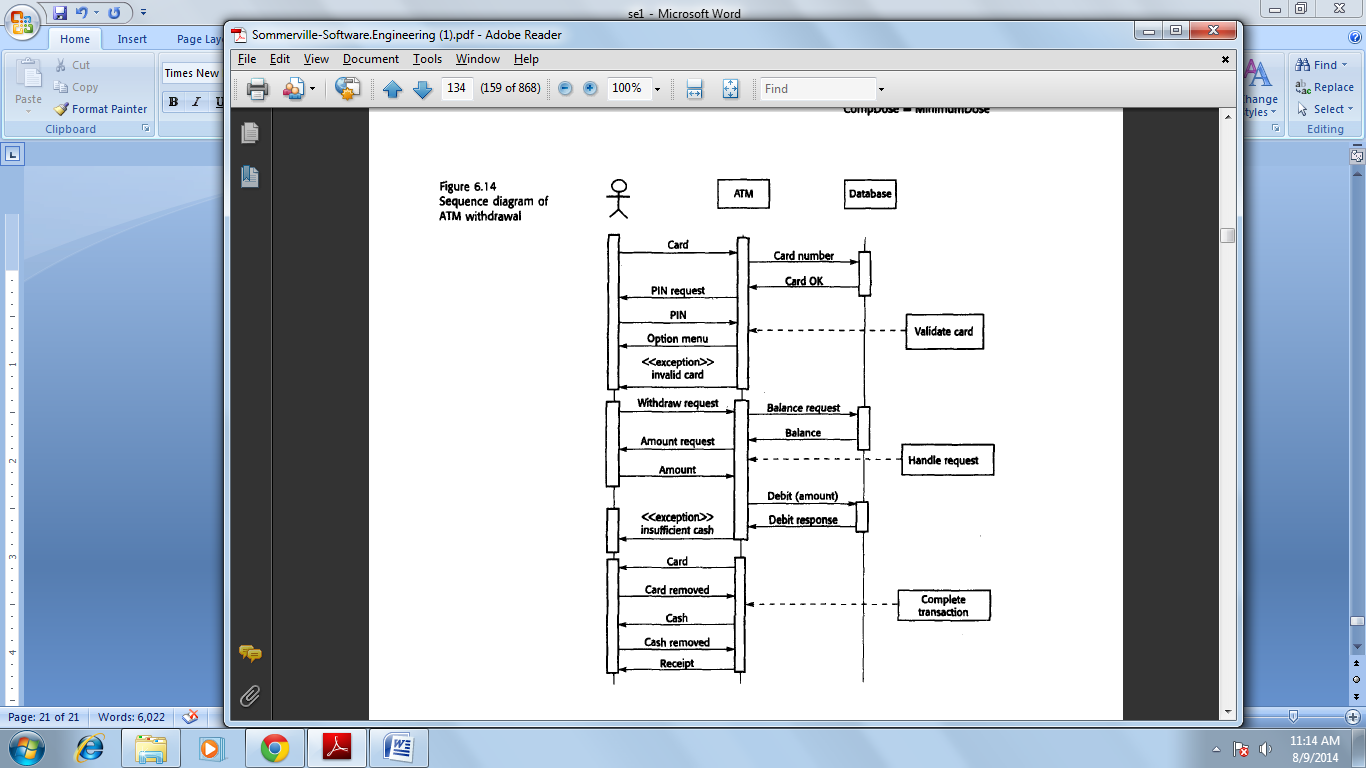
7. Description of the side effects (if any) of the operation.

You should read a sequence diagram from top to bottom to see the order of the actions that take place.

1. ***Validate card***The user s card is validated by checking the card number and user s PIN.

*2.* ***Handle request***The user s request is handled by the system. For a withdrawal, the database must be queried to check the user's balance and to debit the amount withdrawn. Notice the exception here if the requestor does not have enough money in their account.

3. ***Complete transaction***The user s card is returned and, when it is removed, the cash and receipt are delivered.

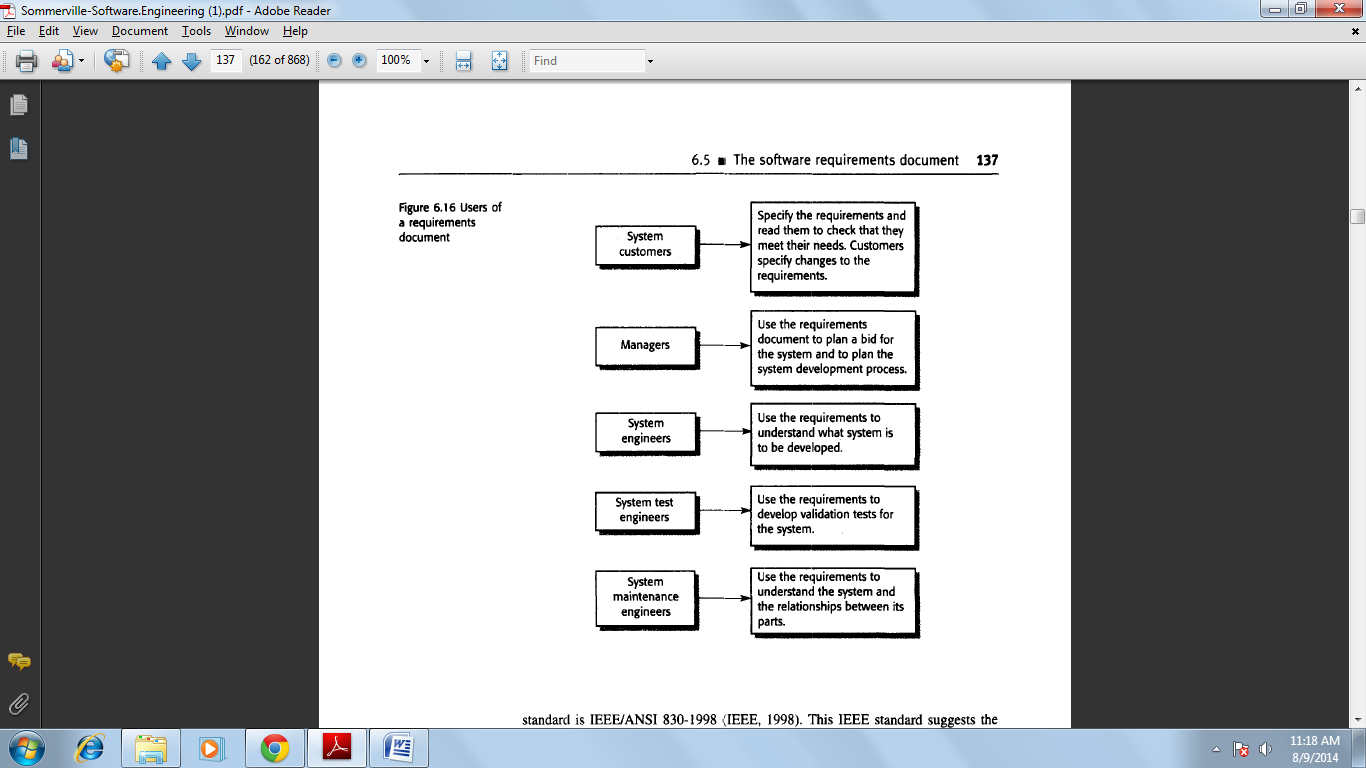


**The software requirements document**

The software requirements document (sometimes called the software requirements specification or SRS) is the official statement of what the system developers should implement. It should include both the user requirements for a system and a detailed specification of the system requirements. In some cases, the user and system requirements may be integrated into a single description. In other cases, the user requirements are defined in an introduction to the system requirements specification.

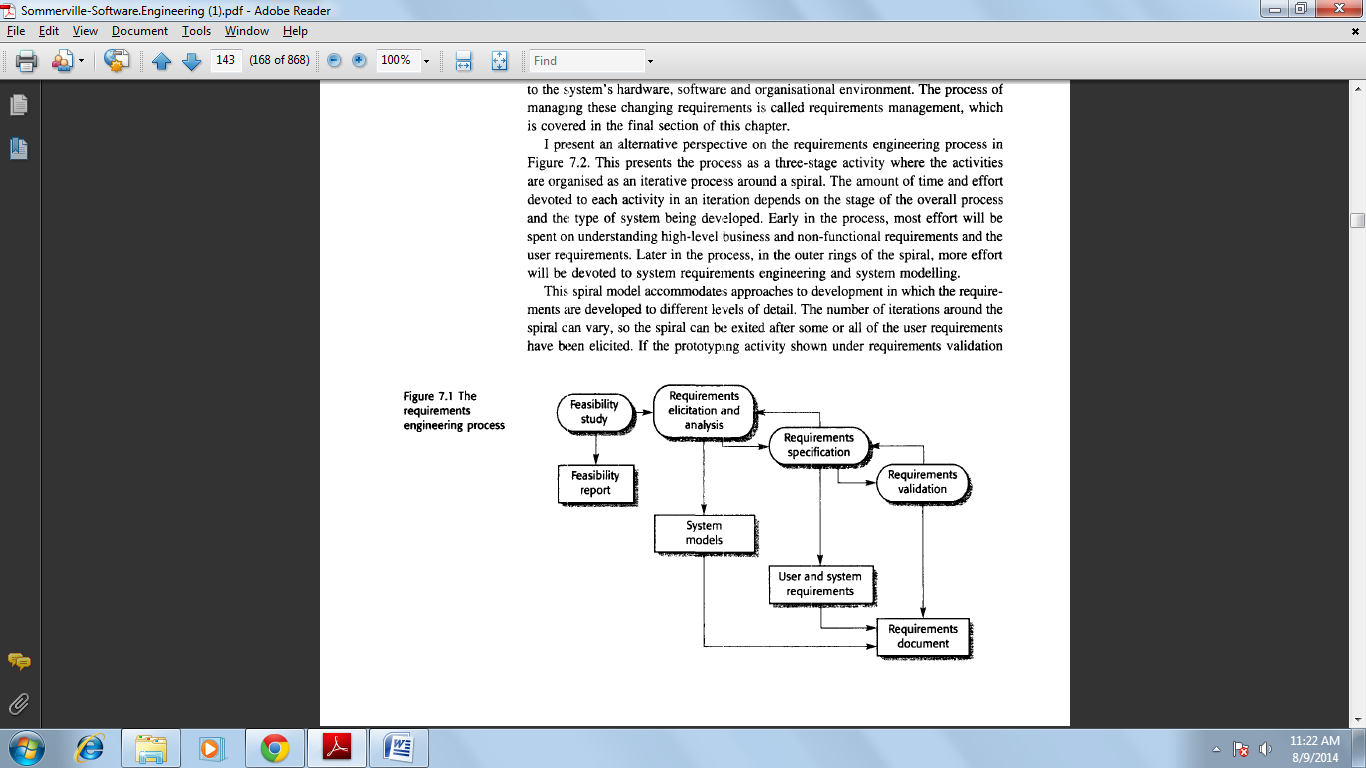
If there are a large number of requirements, the detailed system requirements may be presented in a separate document.

The diversity of possible users means that the requirements document has to be a compromise between communicating the requirements to customers, defining the requirements in precise detail for developers and testers, and including information about possible system evolution. Information on anticipated changes can help system designers avoid restrictive design decisions and help system maintenance engineers who have to adapt the system to new requirements.



**Requirements engineering processes**

The goal of the requirements engineering process is to create and maintain a system requirements document. The overall process includes four high-level requirements engineering sub-processes.

****

**Feasibility studies**

For all new systems, the requirements engineering process should start with a feasibility study. The input to the feasibility study is a set of preliminary business requirements, an outline description of the system and how the system is intended to support business processes. The results of the feasibility study should be a report that recommends whether or not it is worth carrying on with the requirements engineering and system development process.

A feasibility study is a short, focused study that aims to answer a number of questions:

1. Does the system contribute to the overall objectives of the organisation?

2. Can the system be implemented using current technology and within given cost and schedule constraints?

1. Can the system be integrated with other systems which are already in place?

**Requirements elicitation and analysis**

The next stage of the requirements engineering process is requirements elicitation and analysis. In this activity, software engineers work with customers and system end-users to find out about the application domain, what services the system should provide, the required performance of the system, hardware constraints, and so on.

Requirements elicitation and analysis may involve a variety of people in an organisation. The term *stakeholder* is used to refer to any person or group who will be affected by the system, directly or indirectly. Stakeholders include end-users who interact with the system and everyone else in an organisation that may be affected by its installation. Other system stakeholders may be engineers who are developing or maintaining related systems, business managers, domain experts and trade union representatives.

Eliciting and understanding stakeholder requirements is difficult for several reasons:

1. Stakeholders often don't know what they want from the computer system except in the most general terms. They may find it difficult to articulate what they want the system to do or make unrealistic demands because they are unaware of the cost of their requests.

2. Stakeholders naturally express requirements in their own terms and with implicit knowledge of their own work. Requirements engineers, without experience in the customer's domain, must understand these requirements.

3. Different stakeholders have different requirements, which they may express in different ways. Requirements engineers have to consider all potential sources of requirements and discover commonalities and conflict.

4. Political factors may influence the requirements of the system. For example, managers may demand specific system requirements that will increase their influence in the organisation.

5. The economic and business environment in which the analysis takes place is dynamic. It inevitably changes during the analysis process. Hence the importance of particular requirements may change. New requirements may emerge from new stakeholders who were not originally consulted.

The process activities are:

*1.* ***Requirements discovery***This is the process of interacting with stakeholders in the system to collect their requirements. Domain requirements from stakeholders and documentation are also discovered during this activity.

*2.* ***Requirements classification and organisation***This activity takes the unstructured collection of requirements, groups related requirements and organises them into coherent clusters.

*3.* ***Requirements prioritisation and negotiation***Inevitably, where multiple stakeholders are involved, requirements will conflict. This activity is concerned with prioritisatrion requirements, and finding and resolving requirements conflicts through negotiation.

*4.* ***Requirements documentation***The requirements are documented and input into the next round of the spiral. Formal or informal requirements documents may be produced.

In the requirements documentation stage, the requirements that have been elicited are documented in such a way that they can be used to help with further requirements discovery. At this stage, an early version of the system requirements document may be produced, but it will have missing sections and incomplete requirements. Alternatively, the requirements may be documented as tables in a document or on cards. Writing requirements on cards (the approach used in extreme programming) can be very effective, as these are easy for stakeholders to handle, change and organise.

**Requirements discovery**

Requirements discovery is the process of gathering information about the proposed and existing systems and distilling the user and system requirements from this information. Sources of information during the requirements discovery phase include documentation, system stakeholders and specifications of similar systems. You interact with stakeholders through interviews and observation, and may use scenarios and prototypes to help with the requirements discovery.

Stakeholders range from system end-users through managers and external stakeholders such as regulators who certify the acceptability of the system. For example, system stakeholders for a bank ATM include:

1. *Current bank customers* who receive services from the system

*2. Representatives from other banks* who have reciprocal agreements that allow each other's ATMs to be used

3. *Managers of bank branches* who obtain management information from the system

*4. Counter staff at bank branches* who are involved in the day-to-day running of the system

*5. Database administrators* who are responsible for integrating the system with the bank's customer database

*6. Bank security managers* who must ensure that the system will not pose a security hazard

*7. The bank's marketing department* who are likely be interested in using the system as a means of marketing the bank

*8. Hardware and software maintenance engineers* who are responsible for maintaining and upgrading the hardware and software

*9. National banking regulators* who are responsible for ensuring that the system conforms to banking regulations

**Viewpoints**

Viewpoints can be used as a way of classifying stakeholders and other sources of requirements. There are three generic types of viewpoint:

1. *Interactor viewpoints* represent people or other systems that interact directly with the system. In the bank ATM system, examples of interactor viewpoints are the bank's customers and the bank's account database.

*2. Indirect viewpoints* represent stakeholders who do not use the system themselves but who influence the requirements in some way. In the bank ATM system, examples of indirect viewpoints are the management of the bank and the bank security staff.

*3. Domain viewpoints* represent domain characteristics and constraints that influence the system requirements. In the bank ATM system, an example of a domain viewpoint would be the standards that have been developed for interbank communications.

The initial identification of viewpoints that are relevant to a system can sometimes be difficult. To help with this process, you should try to identify more specific viewpoint types:

l. Providers of services to the system and receivers of system services

2. Systems that should interface directly with the system being specified

3. Regulations and standards that apply to the system

4. The sources of system business and non-functional requirements

5. Engineering viewpoints reflecting the requirements of people, who have to develop, manage and maintain the system

6. Marketing and other viewpoints that generate requirements on the product features expected by customers and how the system should reflect the external image of the organization.

**Interviewing**

Formal or informal interviews with system stakeholders are part of most requirements engineering processes. In these interviews, the requirements engineering team puts questions to stakeholders about the system that they use and the system to be develped.

Requirements are derived from the answers to these questions. Interviews may be of two types:

1. Closed interviews where the stakeholder answers a predefined set of questions.

2. Open interviews where there is no predefined agenda. The requirements engineering team explores a range of issues with system stakeholders and hence develops a better understanding of their needs.

It is hard to elicit domain knowledge during interviews for two reasons:

1. All application specialists use terminology and jargon that is specific to a domain. It is impossible for them to discuss domain requirements without using this terminology. They normally use terminology in a precise and subtle way that is easy for requirements engineers to misunderstand.

2. Some domain knowledge is so familiar to stakeholders that they either find it difficult to explain or they think it is so fundamental that it isn't worth mentioning. For example, for a librarian, it goes without saying that all acquisitions are catalogued before they are added to the library. However, this may not be obvious to the interviewer so it isn't taken into account in the requirements.

Effective interviewers have two characteristics:

1. They are open-minded, avoid preconceived ideas about the requirements and are willing to listen to stakeholders. If the stakeholder comes up with surprising requirements, they are willing to change their mind about the system.

2. They prompt the interviewee to start discussions with a question, a requirements proposal or by suggesting working together on a prototype system. Saying to people 'tell me what you want' is unlikely to result in useful information. Most people find it much easier to talk in a defined context rather than in general terms.

**Scenarios**

People usually find it easier to relate to real-life examples than to abstract descriptions.

They can understand and critique a scenario of how they might interact with a software system, Requirements engineers can use the information gained from this discussion to formulate the actual system requirements.

The scenario starts with an outline of the interaction, and, during elicitation, details are added to create a complete description of that interaction. At its most general, a scenano may include:

1. A description of what the system and users expect when the scenario starts

2. A description of the normal flow of events in the scenario

3. A description of what can go wrong and how this is handled

4. Information about other activities that might be going on at the same time

5. A description of the system state when the scenario finishes.

**Use-cases**

Use-cases are a scenario-based technique for requirements elicitation which were first introduced in the Objectory method. They have now become a fundamental feature of the UML notation for describing object-oriented system models. In their simplest form, a use-case identifies the type of interaction and the actors involved.



**Ethnography**

Software systems do not exist in isolation--they are used in a social and organizational context, and software system requirements may be derived or constrained by that context. Satisfying these social and organisational requirements is often critical for the success of the system. One reason why many software systems are delivered but never used is that they do not take proper account of the importance of these requirements.

*Ethnography* is an observational technique that can be used to understand social and organisational requirements. An analyst immerses him or herself in the working environment where the system will be used. He: or she observes the day-to-day work and notes made of the actual tasks in which participants are involved. The value of ethnography is that it helps analysts discover implicit system requirements that reflect the actual rather than the formal processes ill which people are involved.

Ethnography is particularly effective at discovering two types of requirements:

1. ***Requirements that are derived from the way in which people actually work***rather than the way in which process definitions say they ought to work. For example, air traffic controllers may switch off an aircraft conflict alert system that detects aircraft with intersecting flight paths even though normal control procedures specify that it should be used. Their control strategy is designed to ensure that these aircraft are moved apart before problems occur and they find that the conflict alert alarm distracts them from their work.

*2.* ***Requirements that are derived from cooperation and awareness of other people's activities.***For example, air traffic controllers may use an awareness of other controllers' work to predict the number of aircraft that will be entering their control sector. They then modify their control strategies depending on that predicted workload. Therefore, an automated ATC system should allow controllers in a sector to have some visibility of the work in adjacent sectors.



**Requirements validation**

Requirements validation is concerned with showing that the requirements actually define the system that the customer wants. Requirements validation overlaps analysis in that it is concerned with finding problems with the requirements.

Requirements validation is important because errors in a requirements document can lead to extensive rework costs when they are discovered during development or after the system is in service. The cost of fixing a requirements problem by making a system change is much greater than repairing design or coding errors.

The reason for this is that a change to the requirements usually means that the system design and implementation must also be changed and then the system must be tested again.

During the requirements validation process, checks should be carried out on the requirements in the requirements document. These checks include:

1. ***Validity checks***A user may think that a system is needed to perform certain functions. However, further thought and analysis may identify additional or different functions that are required. Systems have diverse stakeholders with distinct needs, and any set of requirements is inevitably a compromise across the stakeholder community.

*2.* ***Consistency checks***Requirements in the document should not conflict. That is, there should be no contradictory constraints or descriptions of the same system function.

*3.* ***Completeness checks***The requirements document should include requirements, which define all functions, and constraints intended by the system user.

*4.* ***Realism checks***Using knowledge of existing technology, the requirements should be checked to ensure that they could actually be implemented. These checks should also take account of the budget and schedule for the system development.

*5.* ***Verifiability***To reduce the potential for dispute between customer and contractor, system requirements should always be written so that they are verifiable. This means that you should be able to write a set of tests that can demonstrate that the delivered system meets each specified requirement.

A number of requirements validation techniques can be used in conjunction or individually:

*1.* ***Requirements reviews***The requirements are analysed systematically by a team of reviewers. This process is discussed in the following section.

*2.* ***Prototyping*** In this approach to validation, an executable model of the system is demonstrated to end-users and customers. They can experiment with this model to see if it meets their real needs.

*3.* ***Test-case generation***Requirements should be testable. If the tests for the requirements are devised as part of the validation process, this often reveals requirements problems. If a test is difficult or impossible to design, this usually means that the requirements will be difficult to implement and should be reconsidered. Developing tests from the user requirements before any code is written is an integral part of extreme programming.

**Requirements reviews**

A requirements review is a manual process that involves people from both client and contractor organisations. They check the requirements document for anomalies and omissions. The review process may be managed in the same way as program inspections. Alternatively, it may be organised as a broader activity with different people checking different parts of the document.

In a formal requirements review, the development team should 'walk' the client through the system requirements, explaining the implications of each requirement.

The review team should check each requirement for consistency as well as check the requirements as a whole for completeness. Reviewers may also check for:

*1.* ***Verifiability***Is the requirement as stated realistically testable?

*2.* ***Comprehensibility***Do the procurers or end-users of the system properly understand the requirement?

*3.* ***Traceability***Is the origin of the requirement clearly stated? You may have to go back to the source of the requirement to assess the impact of a change

Traceability is important as it allows the impact of change on the rest of the system to be assessed. I discuss it in more detail in the following section.

*4.* ***Adaptability***Is the requirement adaptable? That is, can the requirement be changed without large-scale effects on other system requirements?

**Requirements management**

Furthermore, once a system has been installed, new requirements inevitably emerge.

It is hard for users and system customers to anticipate what effects the new system will have on the organisation. Once end-users have experience of a system, they discover new needs and priorities:

1. Large systems usually have a diverse user community where users have different requirements and priorities. These may be conflicting or contradictory. The final system requirements are inevitably a compromise between them and, with experience, it is often discovered. That the balance of support given to different users has to be changed.

2. The people who pay for a system and the users of a system are rarely the same people. System customers impose requirements because of organisational and budgetary constraints. These may conflict with end-user requirements and, after delivery; new features may have to be added for user support if the system is to meet its goals.

3. The business and technical environment of the system changes after installation, and these changes must be reflected in the system. New hardware may be introduced, it may be necessary to interface the system with other systems, business priorities may change with consequent changes in the system support, and new legislation and regulations may be introduced which must be implemented by the system.

The process of requirements management should start as soon as a draft version of the requirements document is available, but you should start planning how to manage changing requirements during the requirements elicitation process.

**Enduring and volatile requirements**

Requirements evolution during the RE process and after a system has gone into service is inevitable. Developing software requirements focuses attention on software capabilities, business objectives and other business systems. As the requirements definition is developed, you normally develop a better understanding of users needs.

*1.* ***Enduring requirements***These are relatively stable requirements that derive from the core activity of the organisation and which relate directly to the domain of the system. For example, in a hospital, there will always be requirements concerned with patients, doctors, nurses and treatments.

*2.* ***Volatile requirements***These are requirements that are likely to change during the system development process or after the system has been become operational. An example would be requirements resulting from government healthcare policies.

**Requirements management planning**

Planning is an essential first stage in the requirements management process. Requirements management is very expensive. For each project, the planning stage establishes the level of requirements management detail that is required. During the requirements management stage, you have to decide on:

1. ***Requirements identification***Each requirement must be uniquely identified so that it can be cross-referenced by other requirements and so that it may be used in traceability assessments
2. **A *change management process***This is the set of activities that assess the impact and cost of changes. I discuss this process in more detail in the following section.

*3.* ***Traceability policies***These policies define the relationships between requirements, and between the requirements and the system design that should be recorded and how these records should be maintained.

*4.* ***CASE tool support***Requirements management involves the processing of large amounts of information about the requirements. Tools that may be used range from specialist requirements management systems to spreadsheets and simple database systems.

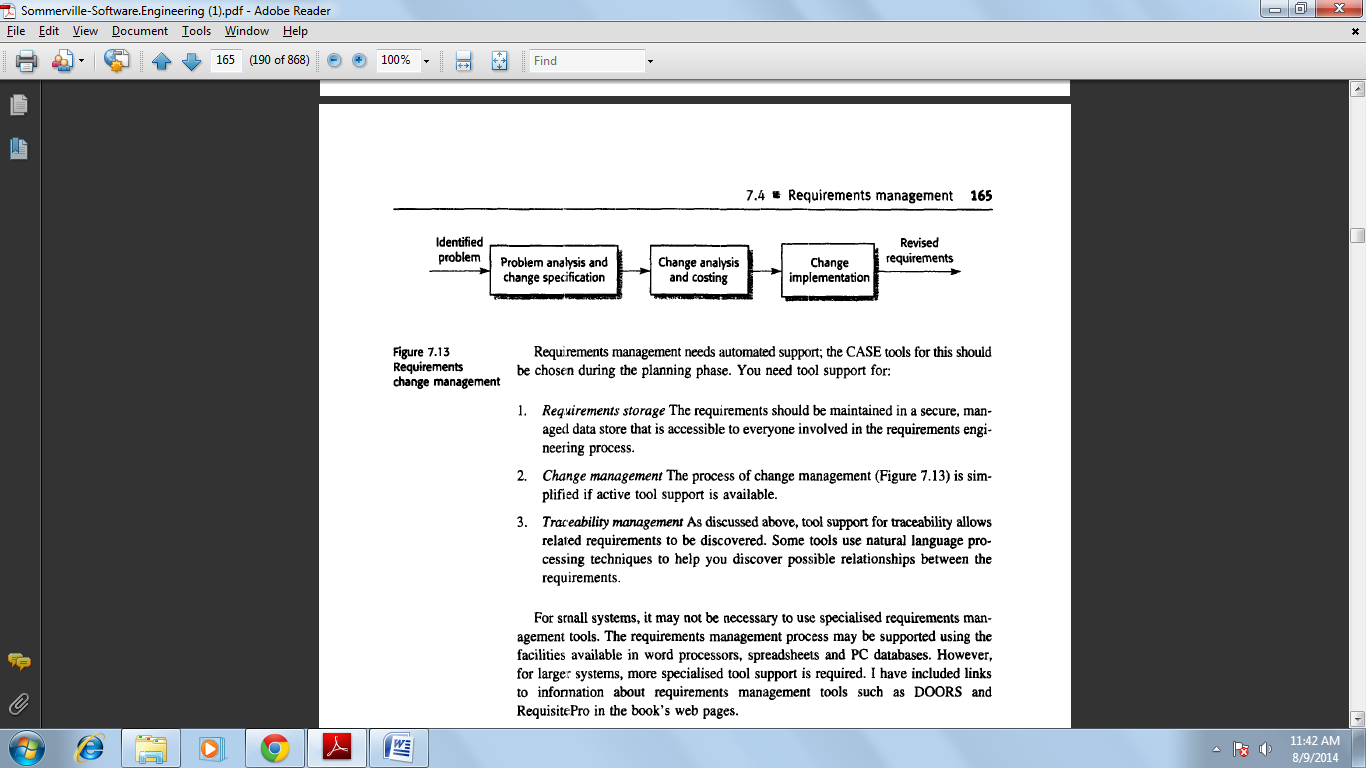
There are three types of traceability information that may be maintained:

1. ***Source traceability***information links the requirements to the stakeholders who proposed the requirements and to the rationale for these requirements. When a change is proposed, you use this information to find and consult the stakeholders about the change.

*2.* ***Requirements traceability***information links dependent requirements within the requirements document. You use this information to assess how many requirements are likely to be affected by a proposed change and the extent of consequential requirements changes that may be necessary.

*3.* ***Design traceability***information links the requirements to the design modules where these requirements are implemented. You use this information to assess the impact of proposed requirements changes on the system design and implementation.

Traceability matrices may be used when a small number of requirements have to be managed, but they become unwieldy and expensive to maintain for large systems with many requirements. For these systems, you should capture traceability information in a requirements database where each requirement is explicitly linked to related requirements. You can then assess the impact of changes by using the database browsing facilities. Traceability matrices can be generated automatically from the database.



Requirements management needs automated support; the CASE tools for this should be chosen during the planning phase. You need tool support for:

l. ***Requirements storage***The requirements should be maintained in a secure, managed data store that is accessible to everyone involved in the requirements engineering process.

*2.* ***Change management***The process of change management is simplified if active tool support is available.

*3.* ***Traceability management***Some tools use natural language processing techniques to help you discover possible relationships between the requirements.

Unit – 2

**System models**

The objective of this chapter is to introduce a number of system models that may be developed during the requirements engineering process.

User requirements should be written in natural language because they have to be understood by people who are not technical experts. However, more detailed system requirements may be expressed in a more technical way. One widely used technique is to document the system specification as a set of system models.

You can use models in the analysis process to develop an understanding of the existing system that is to be replaced or improved or to specify the new system that is required. You may develop different models to represent the system from different perspectives.

1. An external perspective, where the context or environment of the system is modeled.

2. A behavioural perspective, where the behaviour of the system is modeled.

3. A structural perspective, where the architecture of the system or the structure of the data processed by the system is modeled.

Different types of system models are based on different approaches to abstraction.

A data-flow model (for example) concentrates on the flow of data and the functional transformations on that data. It leaves out details of the data structures. By contrast, a model of data entities and their relationships documents the system data structures rather than its functionality.

Examples of the types of system models that you might create during the analysis process are:

1. ***A data- flow model*** Data-flow models show how data is processed at different stages in the system.

*2.* ***A composition model***A composition or *aggregation* model shows how entities in the system are composed of other entities.

*3.* ***An architectural model***Architectural models show the principal sub-systems that make up a system.

*4****. A classification model***Object class/inheritance diagrams show how entities have common characteristics.

*5.* ***A stimulus-response model***A stimulus-response model, or *state transition diagram,* shows how the system reacts to internal and external events.

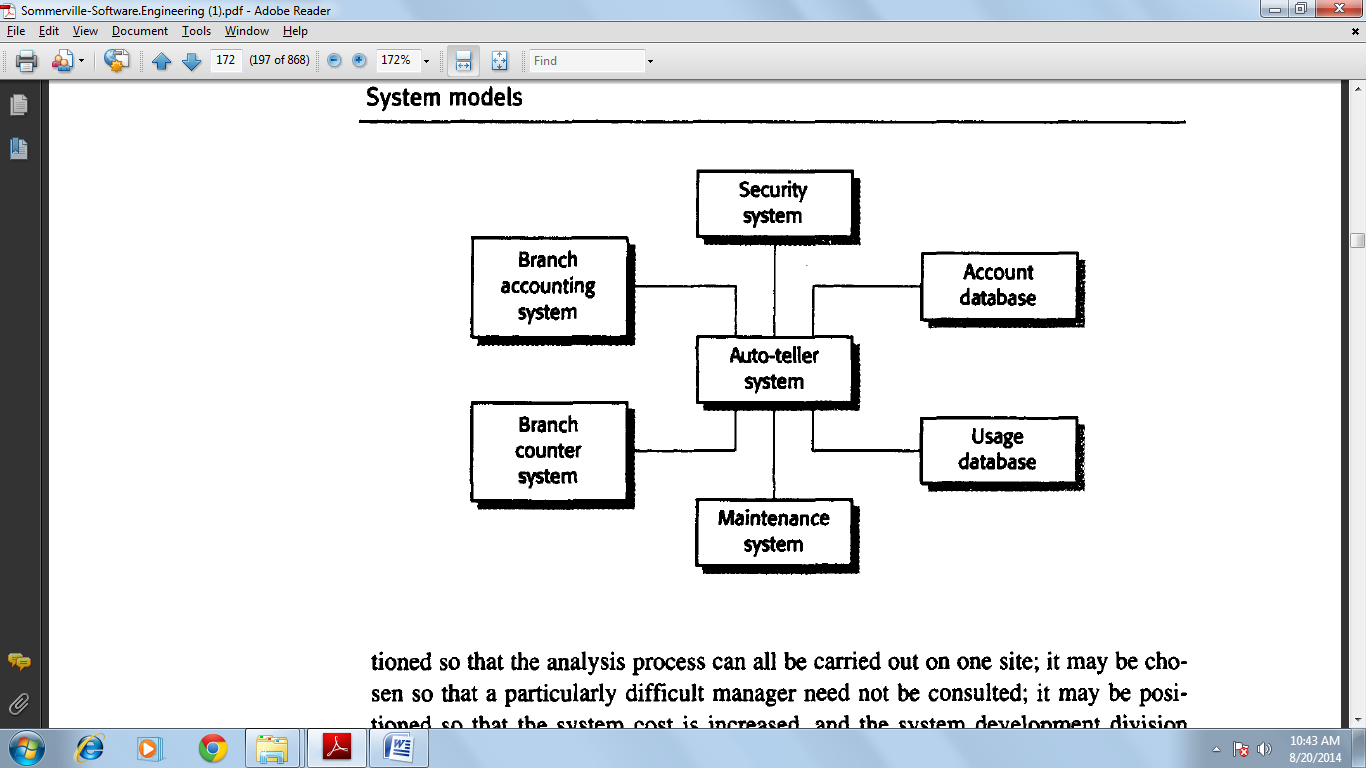
**Context models**

At an early stage in the requirements elicitation and analysis process you should decide the boundaries of the system. This involves working with system stakeholders to distinguish what is the system and what is the system's environment.

You should make these decisions early in the process to limit the system costs and the time needed for analysis.

In some cases, the boundary between a system and its environment is relatively clear. For example, where an automated system is replacing an existing manual or

computerised system, the environment of the new system is usually the same as the existing system's environment. In other cases, there is more flexibility, and you decide what constitutes the boundary between the system and its environment during the requirements engineering process.



We see that each ATM is connected to an account database, a local branch accounting system, a security system and a system to support machine maintenance. The system is also connected to a usage database that monitors how the network of ATMs is used and to a local branch counter system. This counter system provides services such as backup and printing. These, therefore, need not be included in the ATM system itself.

Process model of equipment procurement



Illustrates a process model for the process of procuring equipment in an organisation. This involves specifying the equipment required, finding and choosing suppliers, ordering the equipment, taking delivery of the equipment and testing it after delivery. When specifying computer support for this process, you have to decide which of these activities will actually be supported. The other activities are outside the boundary of the system.

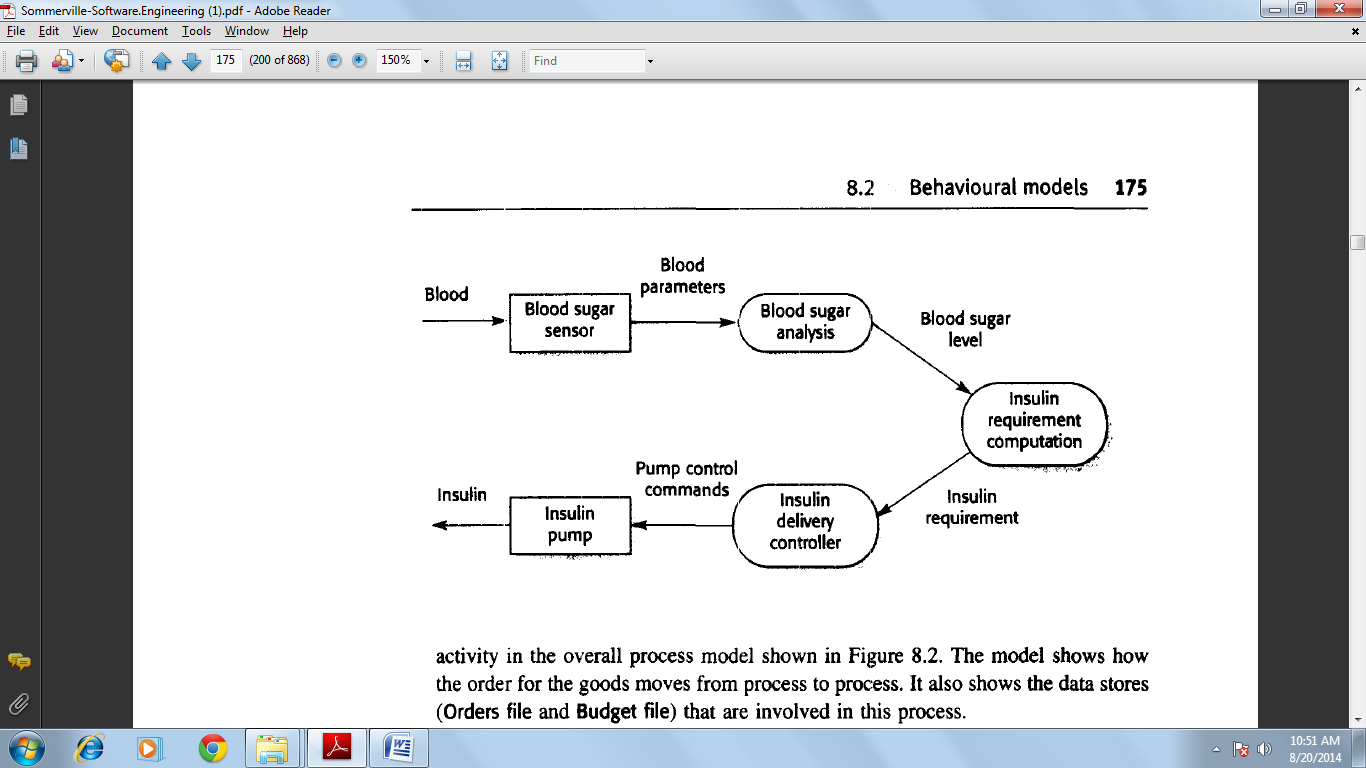
**Behavioural models**

Behavioural models are used to describe the overall behaviour of the system. I discuss two types of behavioural model here: data-flow models, which model the data processing in the system, and state machine models, which model how the system reacts to events. These models may be used separately or together, depending on the type of system that is being developed.

**Data flow models**

Data-flow models are an intuitive way of showing how data is processed by a system. At the analysis level, they should be used to model the way in which data is processed in the existing system. Data-flow models are used to show how data flows through a sequence of processing steps. For example, a processing step could be to filter duplicate records in a customer database. The data is transformed at each step before moving on to the next stage. These processing steps or transformations represent software processes or functions when data-flow diagrams are used to document a software design.

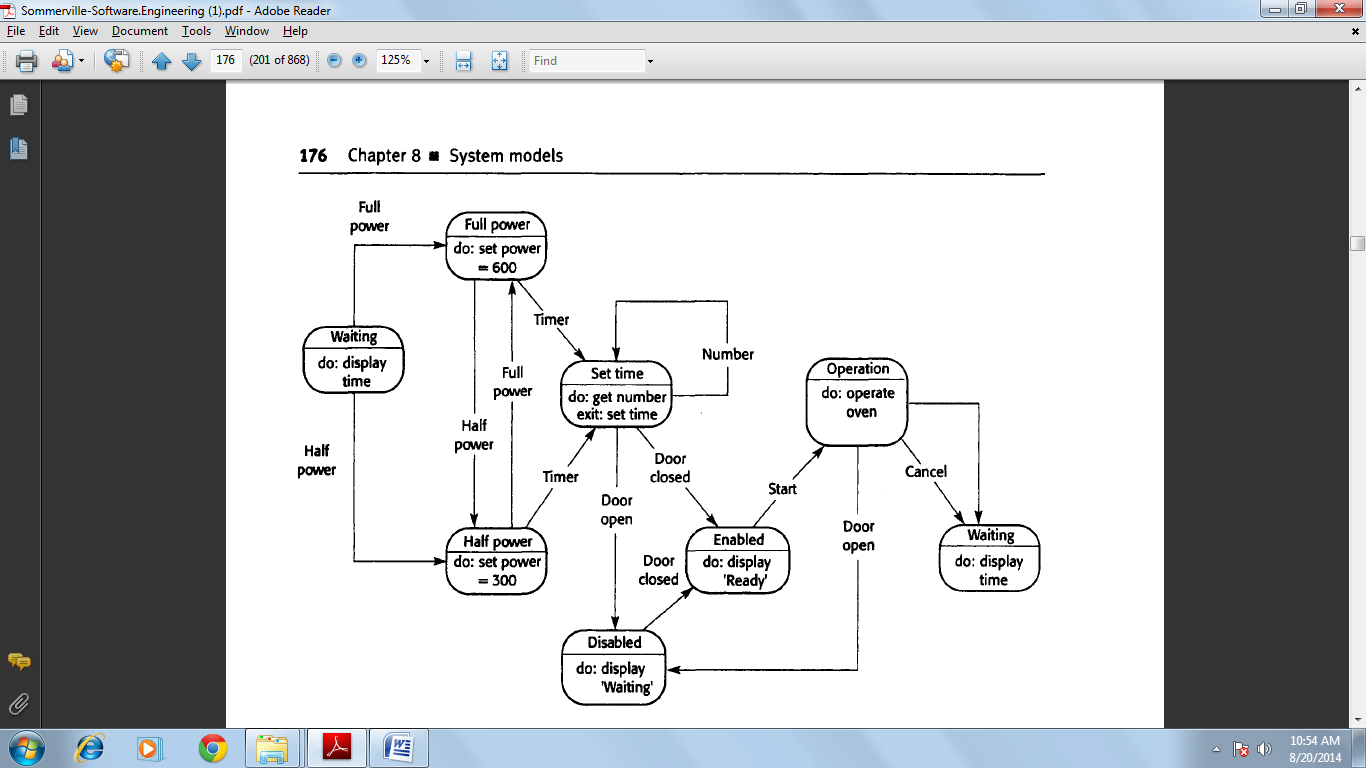
However, in an analysis model, people or computers may carry out the processing.

****

**State machine models**

A state machine model describes how a system responds to internal or external events.

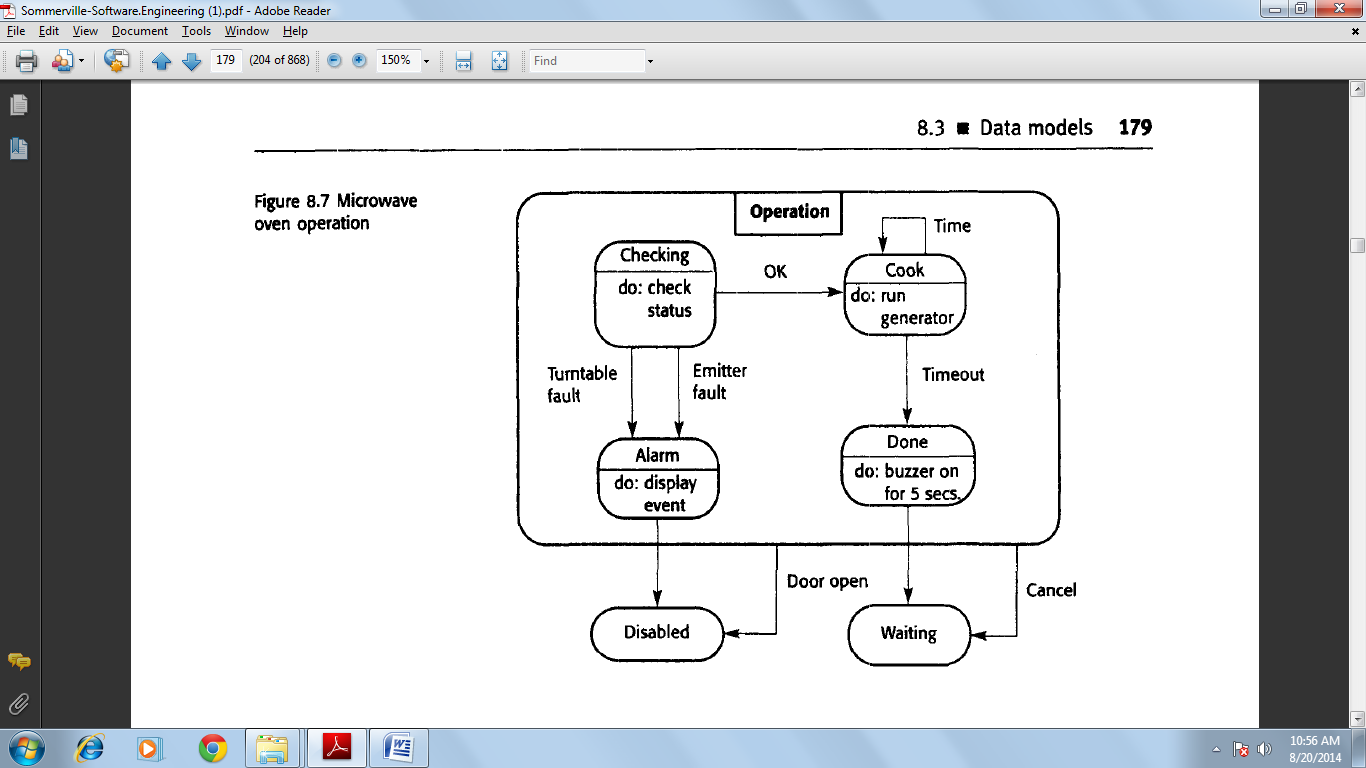
The state machine model shows system states and events that cause transitions from one state to another. It does not show the flow of data within the system. This type of model is often used for modeling real-time systems because these systems are often driven by stimuli from the system's environment. For example, the real-time alarm system.



**Data models**

Most large software systems make use of a large database of information. In some cases, this database is independent of the software system. In others, it is created for the system being developed. An important pmt of systems modeling is defining the logical form of the data processed by the system. These are sometimes called

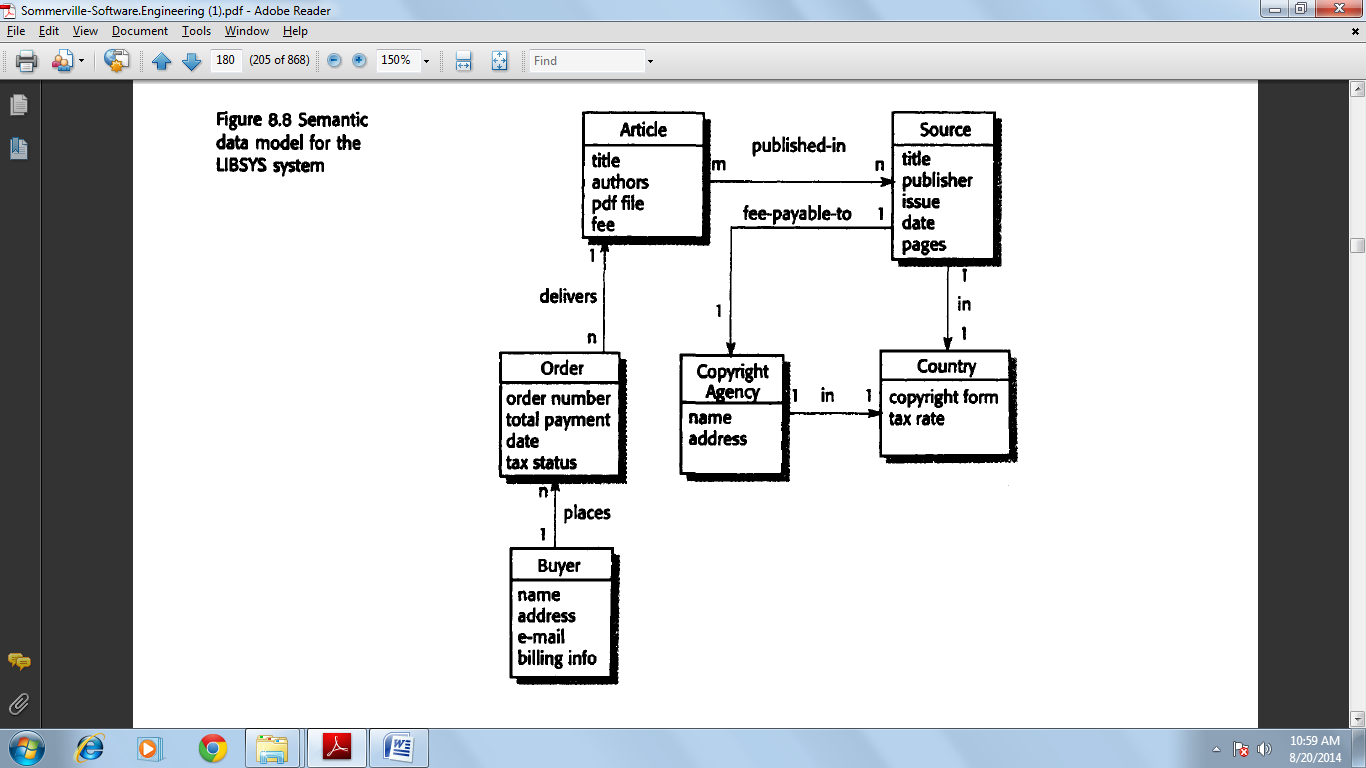
*Semantic data models.*

****

Entity-relationship models have been widely used in database design. The relational database schemas derived from these models are naturally in third normal form, which is a desirable characteristic. Because of the explicit typing and the recognition of sub- and super-types, it is also straightforward to implement these models using object-oriented databases.

**Semantic data model for the L1BSYS system**

A data dictionary is. Simplistically, an alphabetic list of the names included in the system models. As well as the name, the dictionary should include an associated description of the named entity and, if the name represents a composite object, a description of the composition. Other information such as the date of creation. The creator and the representation of the entity may also be included depending on the type of model being developed.

****

The advantages of using a data dictionary are:

1. *It is a mechanism for name management.* Many people may have to invent names for entities and relationships when developing a large system model. These names should be used consistently and should not clash. The data dictionary software can check for name uniqueness where necessary and warn requirements analysts of name duplications.

*2. It serves as a store of organisational information.* As the system is developed, information that can link analysis, design, implementation and evolution is added to the data dictionary, so that all information about an entity is in one place.

**Object models**

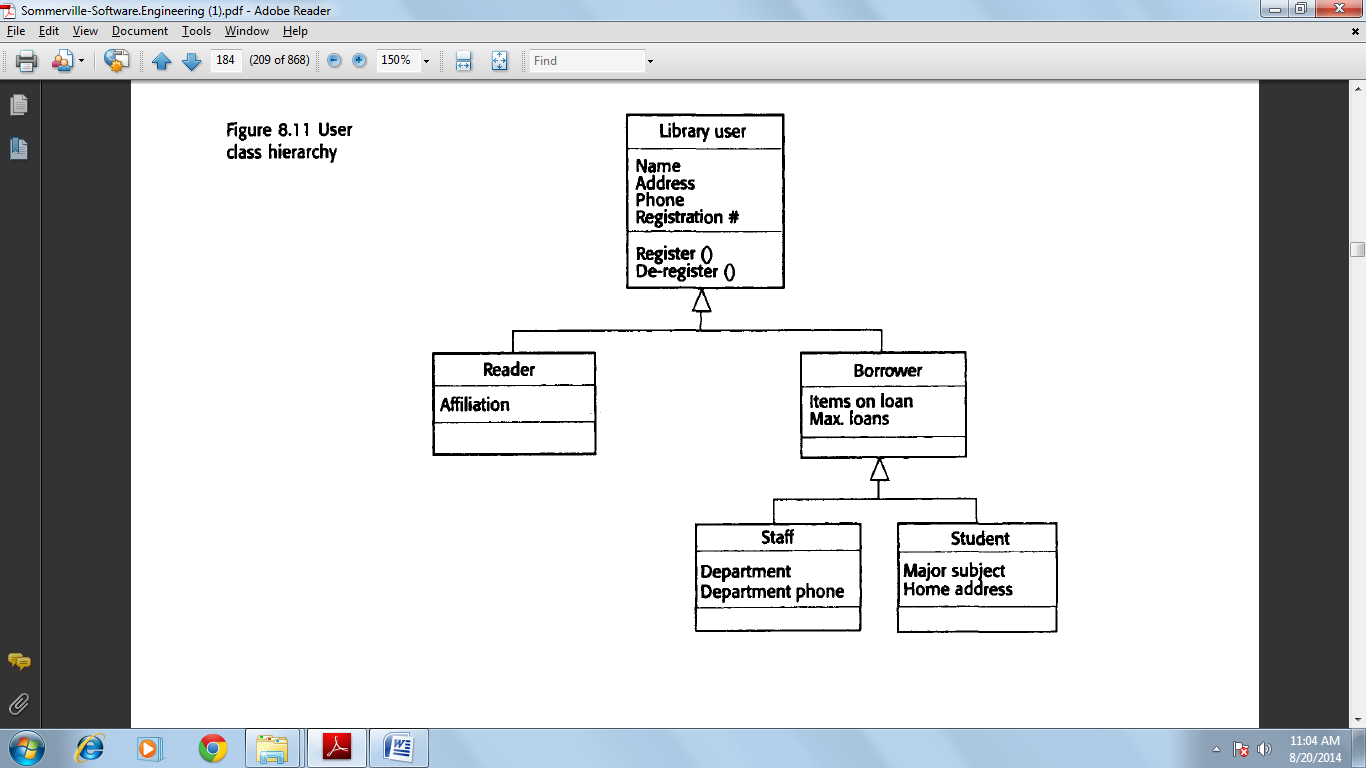
An object-oriented approach to the whole software development process is now commonly used, particularly for interactive systems development. This means expressing the systems requirements using an object model, designing using objects and developing the system in an object-oriented programming language such as Java or C++.

I focus here on object models that show how objects can be classified and can inherit attributes and operations from other objects, aggregation models that show how objects are composed, and simple behavioural models, which show object interactions.

**Inheritance models**

Object-oriented modelling involves identifying the classes of object that are important in the domain being studied. These are then organised into taxonomy. Taxonomy is a classification scheme that shows how atl object class is related to other classes through common attributes and services.

To display this taxonomy, the classes are organised into an inheritance hierarchy with the most general object classes at the top of the hierarchy. More specialized objects inherit their attributes and services. These specialised objects may have their own attributes and services.

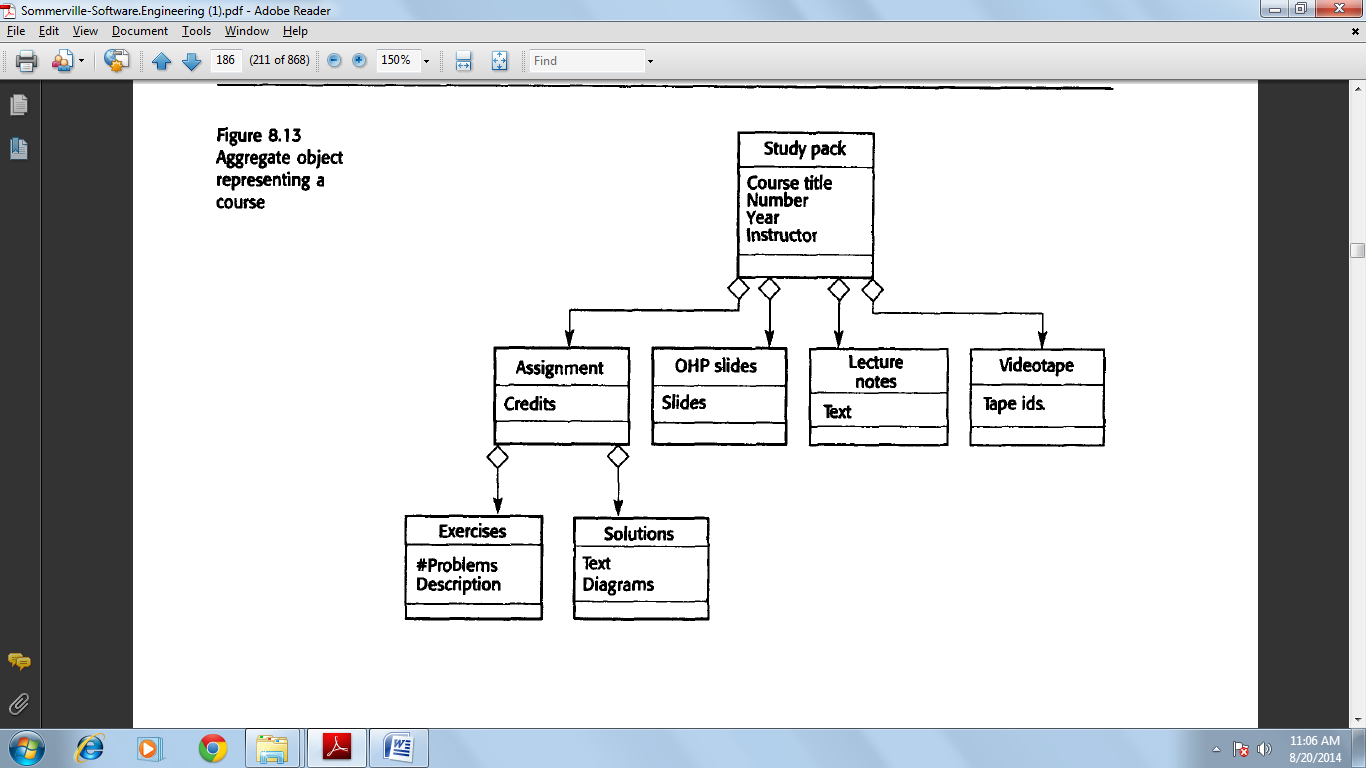


The design of class hierarchies is not easy, so the analyst needs to understand, in detail, the domain in which the system is to be installed. As an example of the subtlety of the problems that arise in practice, consider the library item hierarchy.

It would seem that the attribute Title could be held in the most general item, then inherited by lower-level items.

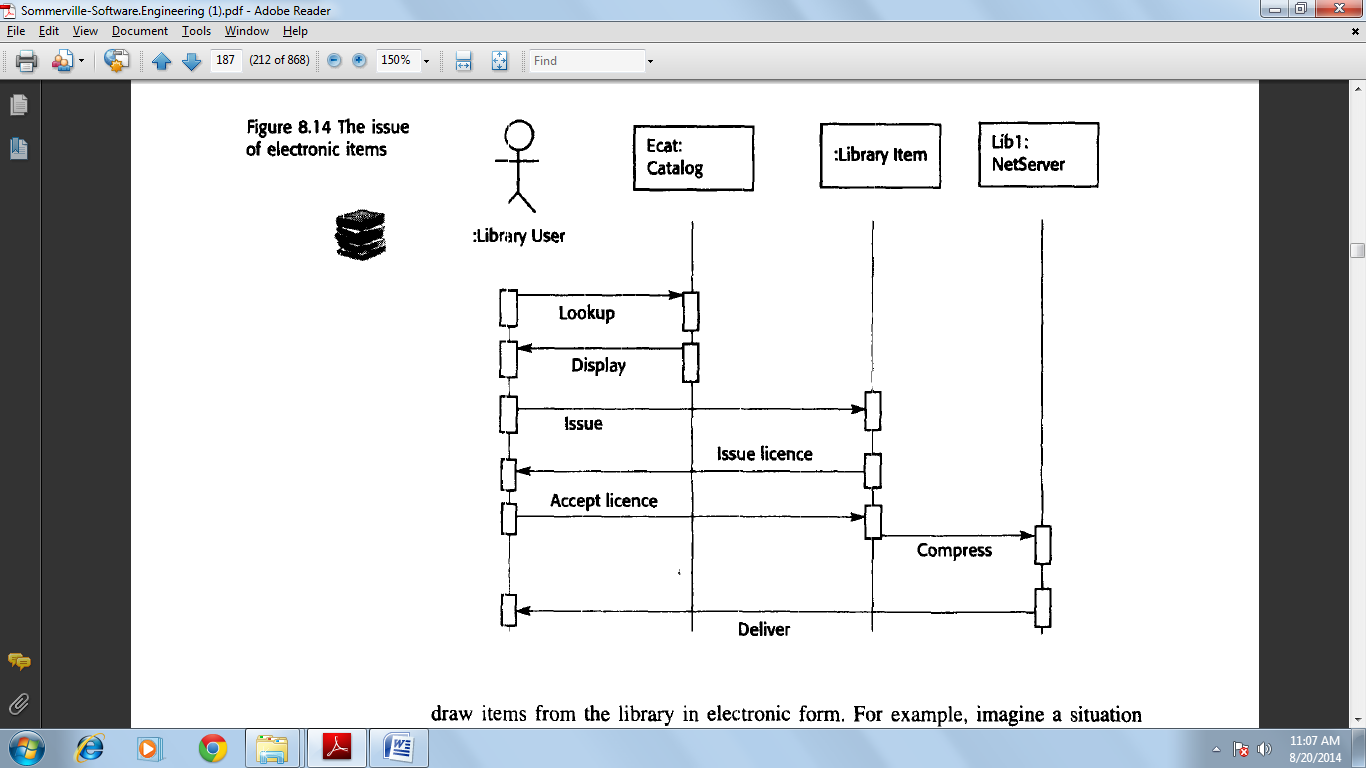
**Object aggregation**

As well as acquiring attributes and services through an inheritance relationship with other objects, some objects are groupings of other objects. That is, an object is an *aggregate* of a set of other objects. The classes representing these objects may be modelled using an object aggregation model,



**Object behaviour modeling**

To model the behaviour of objects, you have to show how the operations provided by the objects are used. In the UML, you model behaviours using scenarios that are represented as UML use-cases. One way to model behaviour is to use UML sequence diagrams that show the sequence of actions involved in a use-case. As well as sequence diagrams, the UML also includes collaboration diagrams that show the sequence of messages exchanged by objects. These are similar to sequence diagrams so I do not cover them here.



In a sequence diagram, objects and actors are aligned along the top of the diagram.

Labeled arrows indicate operations; the sequence of operations is from top to bottom. In this scenario, the library user accesses the catalogue to see whether the item required is available electronically; if it is, the user requests the electronic issue of that item. For copyright reasons, this must be licensed so there is a transaction between the item and the user where the license is agreed. The item to be issued is then sent to a network server object for compression before being sent to the library user.

**Prototyping in software process**

**Prototype**

A prototype is an initial version of a software system which is used to demonstrate concepts, try out design options and generally to find out more about the problem and its possible solutions. It supports the requirement engineering activities such as Requirement Elicitation and Requirements validation.

The benefits of prototyping are

1. Improved system usability
2. A closer match of the system to the user needs.
3. Improved design quality
4. Improved maintainability
5. Reduced development effort

**The Process of Prototype development**

Establish prototype objectives

Define prototype functionality

Develop Prototype

Evaluate Prototype

Prototyping Plan

Outline Description

Executable prototype

Evaluation Prototype

The objectives of Prototyping should be made Explicit from the start of the process. These may be to develop a system to prototype the user interface to develop a system to validate functional system requirements or to develop a system to demonstrate to feasibility of the application to management .The same prototype cannot meet all objectives .If objectives are left implicit, management or end –users may mcounderstand the function of the prototype .The next stage in the process to is decide what to put into and, Perhaps more importantly. What to leave prototyping costs, and accurate the delivery schedule, you may leave some functionality out o the prototype. The final stage of the process is prototype evaluation provision must be made during this stage for user training and the prototype objectives should be used to derive a plan for evaluation.

**PROTOTYPING IN THE SOFTWARE PROCESS**

Outline Requirement

Delivered System

Executable Prototype System Specification

Evolutionary Prototyping

Throw-away Prototyping

Objectives of evolutionary prototyping is to deliver a working system to end-user .This means that you should normally start with the user requirements which are best understood & which have the highest priority. Lower priority & vaguer requirements are implemented when and if they are demanded by the users.

Objectives of throw-away prototyping is to validate or derive the system requirements .You should start with those requirements that are not well understood , because you need to find out more about them requirements that are straight forward may never need to be prototyping.

**Evolutionary Prototyping**

Evolutionary prototyping is based on the idea of developing an initial implementation, exposing this to user comment and refining this through many stages until an adequate system has been developed .This approach to development was used initially for those systems which are difficult or impossible to specify, however, it has now become a mainstream technique of software development.

Develop abstract Specification

Deliver System

Build prototype System

Use prototype System

System Adequate

No

Yes

Advantages:

1. Accelerated delivery of the system.
2. User engagement with the system.

There are differences in detail between the particular methods of rapid. Software development but they all share some fundamental characteristics.

1. The processes of specification, design & implementation are interleaved .There is no detailed system specification & the design documentation produced. Usually depends on the tools used to implement the system. The user requirements document only defines the most important characteristics of the system.
2. The system is developed in a series of increments. End-users and other system stakeholders are involved in designing and evaluating each increment. They may propose changes to the software and new requirements which should be implemented in a later version of the system.
3. Techniques for rapid system development are used .These may include CASE tools & fourth-generation language .
4. System user interfaces are usually developed using an interactive development system which allows the interface design to be created quickly by drawing & placing icons on the surface.

Evolutionary prototyping & specification-based approaches to software different in their views of verification & validation. verification to the process of checking that a program conforms to its, specification. As there is no detailed specification for the prototype verification is therefore impossible.

Validation should demonstrate that the program is suitable for its intended purpose rather than its specification as there is no explicit statements of purpose.

The problem with evolutionary prototype is

1. *Management problems*: Software management structures for large systems are set up to dual with a software process model that generates regular deliverables to access programs.
2. *Maintenance problems*: Conditional change tends to corrupt the structure of the prototype system. This means that anyone apart from the original developers is likely to find it difficult to understand.
3. *Contractual problems*: The normal contractual model between a customer & a software developer is based around a system specification. When there is no such specification it may be difficult to design a contract for the system development.

**THROW-AWAY PROTOTYPING**

A software process model based on an initial throw away prototyping stage is illustrated in figure. This approach extends the requirements analysis process with the intention of reducing overall life-cycle costs. The principal function of the prototype is to clarify requirements and provide additional information for managers to assess process risks. After evaluation the prototype is thrown away. It is not used as a basis for further system development.

Outline Requirements

Develop Prototype

Evaluate Prototype

Specify System

Develop Software

Validate System

Delivered Software System

Reusable components

The system must be developed as quickly as possible. So that users can feed back their prototype experience to the development of the system specification functionality may be stripped from the throw-away prototype where these functions are well understood, quality standards may be relaxed and performance criteria ignored. The prototype development language will often be different from the final system implementation language.

**USER INTERFACE PROTOTYPING**

Graphical user interfaces have now become the norm for interactive systems. The effort involved in specifying, designing and implementing. A user interface represents a significant part of application development costs. The user must take part in the interface design process. This realization led to an approach to design called user centered design. That depends on interface prototyping and user involvement throughout the interface design process.

Prototyping is an essential part of the user interface design process. Because of the dynamic nature of user interfaces, textual descriptions and diagrams are not good enough for expressing the user interface requirements .Therefore, evolutionary prototyping with end-user involvement is the only sensible way to develop graphical user interfaces for software systems.

Interface generators are graphical screen design systems where interface components such as menus, fields, icons and buttons are selected from a menu and positioned on an interface. As I have already discussed, Systems of this type are an essential part of a database programming system and visual basic has based its standard development technique around such a system. Interface generators create a well structured program generated from an interface specification.

Web-based user interfaces may be prototyped by using a standard web-site editor which is, essentially, a user interface builder.

**Software design**

The essence of software design is making decisions about the logical organization of the software. Sometimes, you represent this logical organisation as a model in a defined modelling language such as the UML and sometimes you simply use informal notations and sketches to represent the design. Of course, you rarely start from scratch when making decisions about the software organisation but base your design on previous design experience.

**Architectural design**

Large systems are always decomposed into sub-systems that provide some related set of services. The initial design process of identifying these sub-systems and establishing a framework for sub-system control and communication is called architectural design. The output of this design process is a description of the software architecture.

Three advantages of explicitly designing and documenting software architecture:

1. ***Stakeholder communication***The architecture is a high-level presentation of the system that may be used as a focus for discussion by a range of different stakeholders.

*2.* ***System analysis***Making the system architecture explicit at an early stage in the system development requires some analysis. Architectural design decisions have a profound effect on whether the system can meet critical requirements such as performance, reliability and maintainability.

*3.****Large-scale reuse***A system architecture model is a compact, manageable description of how a system is organised and how the components interoperate.

The system architecture is often the same for systems with similar requirements and so can support large-scale software reuse. It may be possible to develop product-line architectures where the same architecture is used across a range of related systems.

The particular style and structure chosen for an application may therefore depend on the non-functional system requirements:

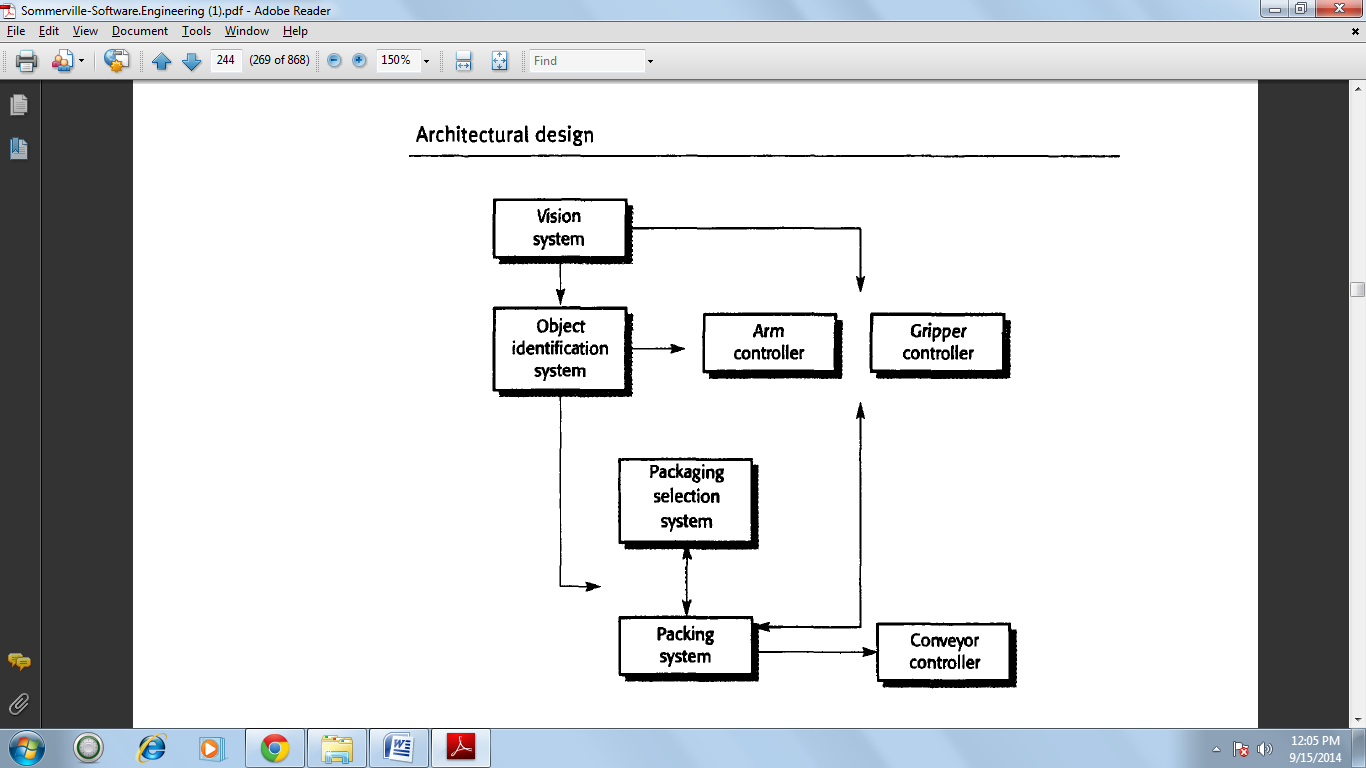
1. ***Performance*** If performance is a critical requirement, the architecture should be designed to localize critical operations within a small number of subsystems, with as little communication as possible between these sub-systems. This may mean using relatively large-grain rather than fine-grain components to reduce component communications.

*2.* ***Security***If security is a critical requirement, a layered structure for the architecture should be used, with the most critical assets protected in the innermost layers and with a high level of security validation applied to these layers.

*3.* ***Safety*** If safety is a critical requirement, the architecture should be designed so that safety-related operations are all located in either a single sub-system or in a small number of sub-systems. This reduces the costs and problems of safety validation and makes it possible to provide related protection systems.

*4.* ***Availability*** If availability is a critical requirement, the architecture should be designed to include redundant components and so that it is possible to replace and update components without stopping the system.

*5.* ***Maintainability*** If maintainability is a critical requirement, the system architecture should be designed using fine-grain, self-contained components that may readily be changed. Producers of data should be separated from consumers and shared data structures should be avoided.



Architectural models that may be developed may include:

1. **A *static structural model***that shows the sub-systems or components that are to be developed as separate units.

2. **A *dynamic process model***that shows how the system is organised into processes at run-time. This may be different from the static model.

3. **An *interface model***that defines the services offered by each sub-system through its public interface.

*4.* ***Relationship models***that show relationships, such as data flow, between the sub-systems.

5. **A *distribution model***that shows how sub-systems may be distributed across computers.

**System organisation**

The organisation of a: system reflects the basic strategy that is used to structure a system. You have to make decisions on the overall organisational model of a system early in the architectural design process. The system organisation may be directly reflected in the sub-system structure. However, it is often the case that the sub-system model includes more detail than the organisational model, and there is not always a simple mapping from sub-systems to organisational structure.

**The repository model**

Sub-systems making up a system must exchange information so that they can work together effectively. There are two fundamental ways in which this can be done.

1. All shared data is held in a central database that can be accessed by all subsystems. A system model based on a shared database is sometimes called a *repository model.*

2. Each sub-system maintains its own database. Data is interchanged with other sub-systems by passing messages to them. The majority of systems that use large amounts of data are organised around a shared database or repository. This model is therefore suited to applications where data is generated by one sub-system and used by another.

The advantages and disadvantages of a shared repository are as follows:

1. It is an efficient way to share large amounts of data. There is no need to transmit data explicitly from one sub-system to another.

2. However, sub-systems must agree on the repository data model. Inevitably, this is a compromise between the specific needs of each tool. Performance may be adversely affected by this compromise. It may be difficult or impossible to integrate new sub-systems if their data models do not fit the agreed schema.

3. Sub-systems that produce data need not be concerned with how that data is used by other sub-systems.

4. However, evolution may be difficult as a large volume of information is generated according to an agreed data model. Translating this to a new model will certainly be expensive; it may be difficult or even impossible.

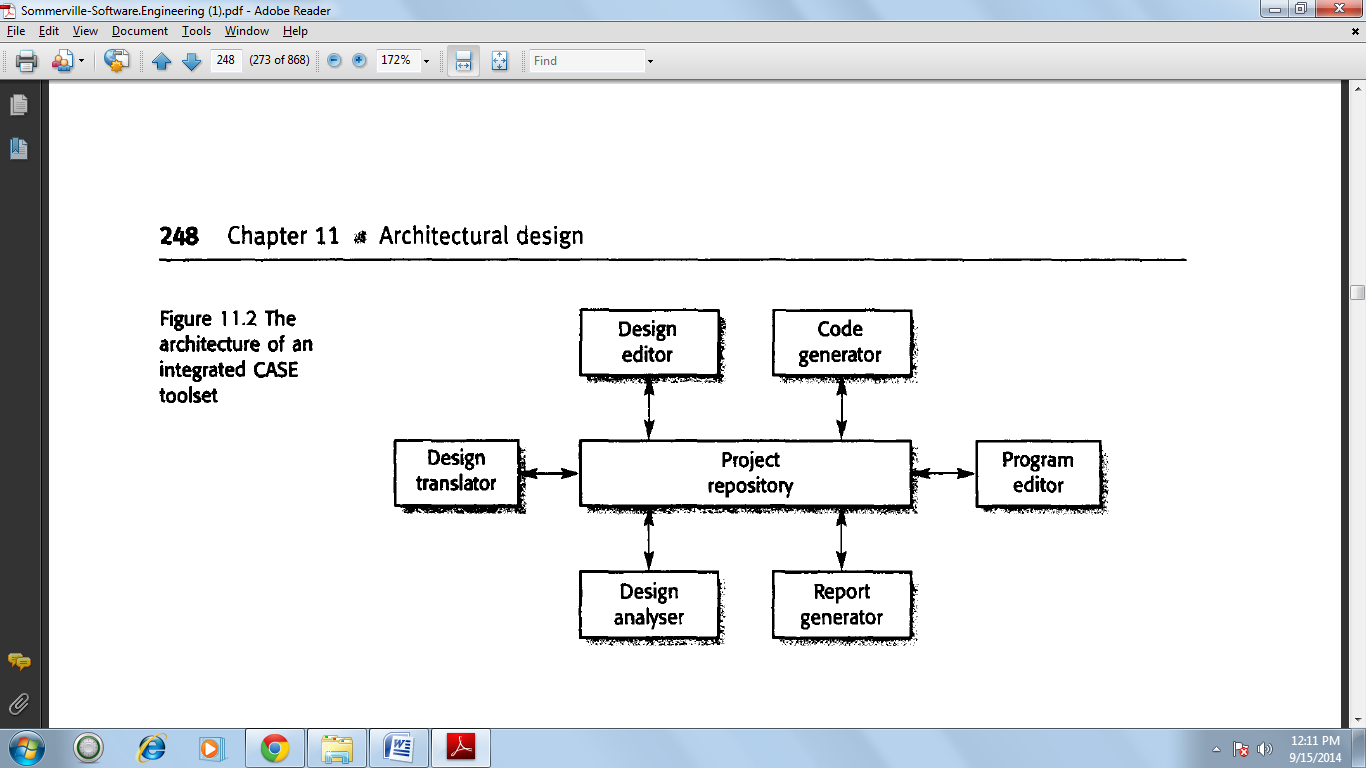
5. Activities such as backup, security, access control and recovery from error are centralised. They are the responsibility of the repository manager. Tools can focus on their principal function rather than be concerned with these issues.

6. However, different sub-systems may have different requirements for security, recovery and backup policies. The repository model forces the same policy on all sub-systems.

7.The model of sharing is visible through the repository schema. It is straightforward to integrate new tools given that they are compatible with the agreed data model.

8. However, it may be difficult to distribute the repository over a number of machines. Although it is possible to distribute a logically centralised repository, there may be problems with data redundancy and inconsistency.

**The architecture of an integrated CASE toolset**



**The client-server model**

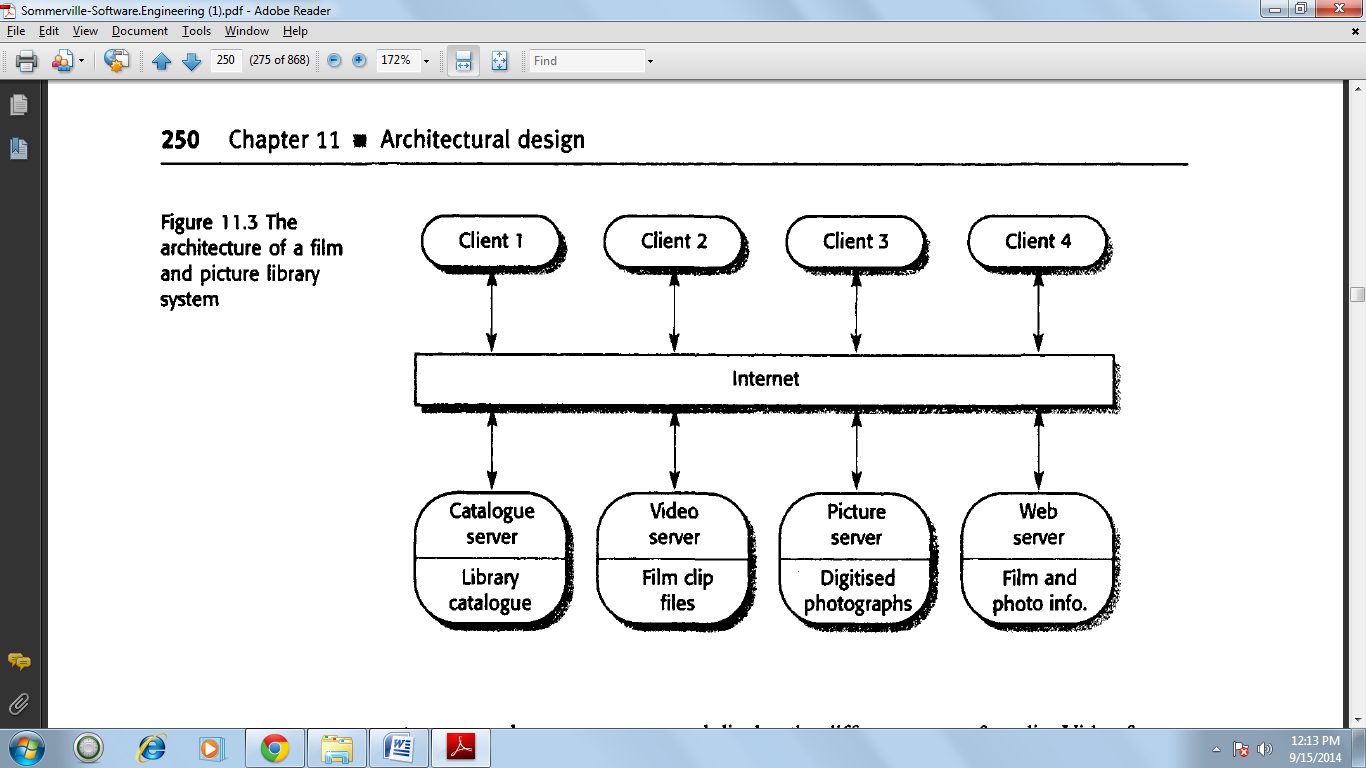
The client-server architectural model is a system model where the system is organized as 2 set of services and associated servers and clients that access and use the services. The major components of this model are:

1. A set of servers that offer services to other sub-systems. Examples of servers are print servers that offer printing services, file servers that offer file management services and a compile server, which offers programming language compilation services.

2. A set of clients that call on the services offered by servers. These are normally sub- systems. In their own right. There may be several instances of a client program executing concurrently.

3. A network that allows the clients to access these services. This is not strictly necessary as both the clients and the servers could run on a single machine. In practice, however, most client-server systems are implemented as distributed systems.

**The architecture of a film and picture library system**



Abstract Machine Model

The abstract machine model of an architecture models the interfacing of sub systems. It organizes a system into a series of layers each of which provides a set of services. Each layer defines an abstract machine whose machine language is used to implement the next level of abstract machine.

**The layered model**

The layered model of an architecture (sometimes called an abstract machine model) organises a system into layers, each of which provide a set of services. Each layer can be thought of as an abstract machine whose machine language is defined by the services provided by the layer. This 'language' is used to implement the next level of abstract machine. For example, a common way to implement a language is to define an ideal 'language machine' and compile the language into code for this machine. A further translation step then converts this abstract machine code to real machine code.

**Control models**

The models for structuring a system are concerned with how a system is decomposed into sub-systems. To work as a system, sub-systems must be controlled so that their services are delivered to the right place at the right time. Structural models do not (and should not) include control information. Rather, the architect should organise the sub-systems according to some control model that supplements the structure model that is used. Control models at the architectural level are concerned with the control flow between sub-systems.

**There are two generic control styles that are used in software systems:**

*1.* ***Centralised control***One sub-system has overall responsibility for control and starts and stops other sub-systems. It may also devolve control to another subsystem but will expect to have this control responsibility returned to it.

*2****. Event-based control***Rather than control information being embedded in a subsystem, each sub-system can respond to externally generated events. These events might come from other sub-systems or from the environment of the system.

Control styles are used in conjunction with structural styles. All the structural styles that I have discussed may be realised using centralised or event-based control.

**Centralised control**

In a centralised control model, one sub-system is designated as the system controller and has responsibility for managing the execution of other sub-systems. Centralised control models fall into two classes, depending on whether the controlled sub-systems execute sequentially or in parallel.

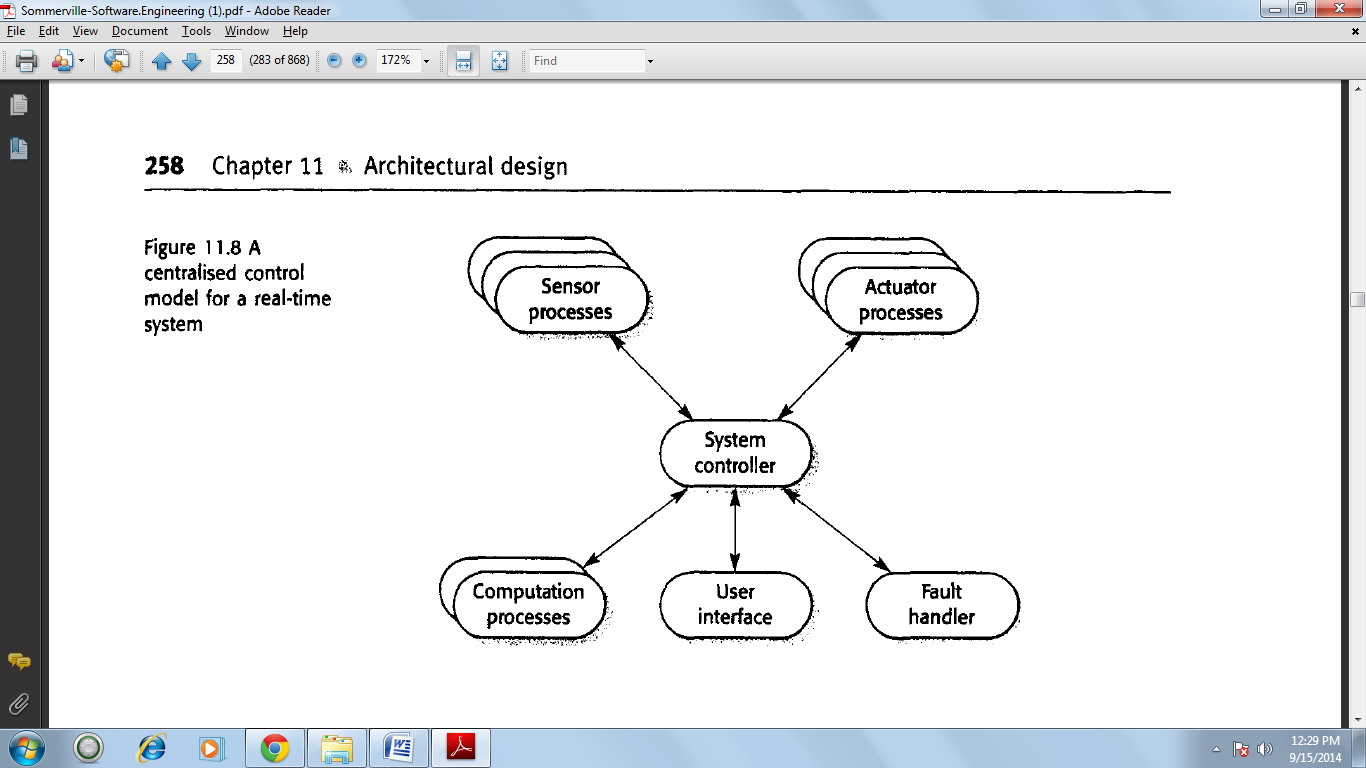
*1****. The call-return model***This is the familiar top-down subroutine model where control starts at the top of a subroutine hierarchy and, through subroutine calls, passes to lower levels in the tree. The subroutine model is only applicable to sequential systems.



The main program can call Routines 1,2 and 3; Routine 1 can call Routines 1.2 or 1.2; Routine 3 can call Routines 3.1 or 3.2; and so on. This is a model of the program dynamics. It is *not* a structural model; there IS no need for Routine 1.1, for example, to be part of Routine 1. This familiar model is embedded in programming languages such as C, Ada and Pascal. Control passes from a higher-level routine in the hierarchy to a lower-level routine, It then retums to the point where the routine was called. The currently executingmbroutine has responsibility for control and can either call other routines or return I;ontrol to its parent. It is poor programming style to return to some other point in the program.

*2.* ***The manager model***This is applicable to concurrent systems. One system component is designated as a system manager and controls the starting, stopping and coordination of other system processes. A process is a sub-system or module that can execute in parallel with other processes. A form of this model may also be applied in sequential systems where a management routine calls particular sub-systems depending on the values of some state variables. This is usually implemented as a case statement.

**A centralised control model for a real-time system**



**Event driven systems**

In centralised control models, control decisions are usually determined by the values of some system state variables. By contrast, event-driven control models are driven by externally generated events. The term *event* in this context does not just mean a binary signal. It may be a signal that can take a range of values or a command input from a menu. The distinction between an event and a simple input is that the timing of the event is outside the control of the process that handles that event. There are many types of event-driven systems. These include editors where user interface events signify editing commands, rule-based production systems as used in AI where a condition becoming true causes an action to be triggered, and active objects where changing a value of an object's attribute triggers some actions.

**I discuss two event-driven control models:**

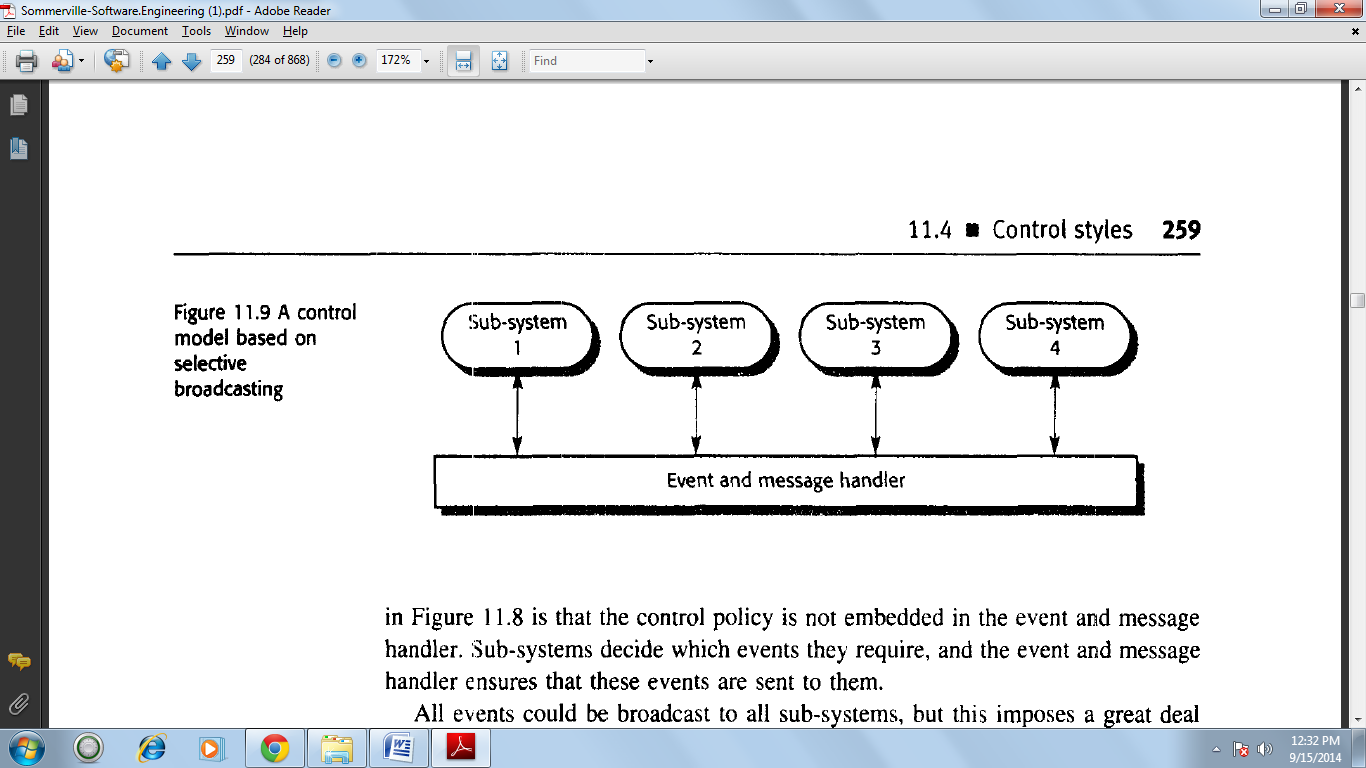
1. ***Broadcast models***In these models, an event is broadcast to all sub-systems. Any sub-system that has been programmed to handle that event can respond to it.

*2.* ***Interrupt-driven models***These are exclusively used in real-time systems where external interrupts are detected by an interrupt handler. They are then passed to some other component for processing.

**Broadcast models**

Broadcast models are effective in integrating sub-systems distributed across different computers on a network. Interrupt-driven models are used in real-time systems with stringent timing requirements.

**Control model based on selective broadcasting**



In a broadcast model, sub-systems register an interest in specific events. When these events occur, control is transferred to the sub-system that can handle the event.

**Interrupt-driven models**

There are a known number of interrupt types with a handler defined for each type. Each type of interrupt is associated with the memory location where its handler's address is stored.

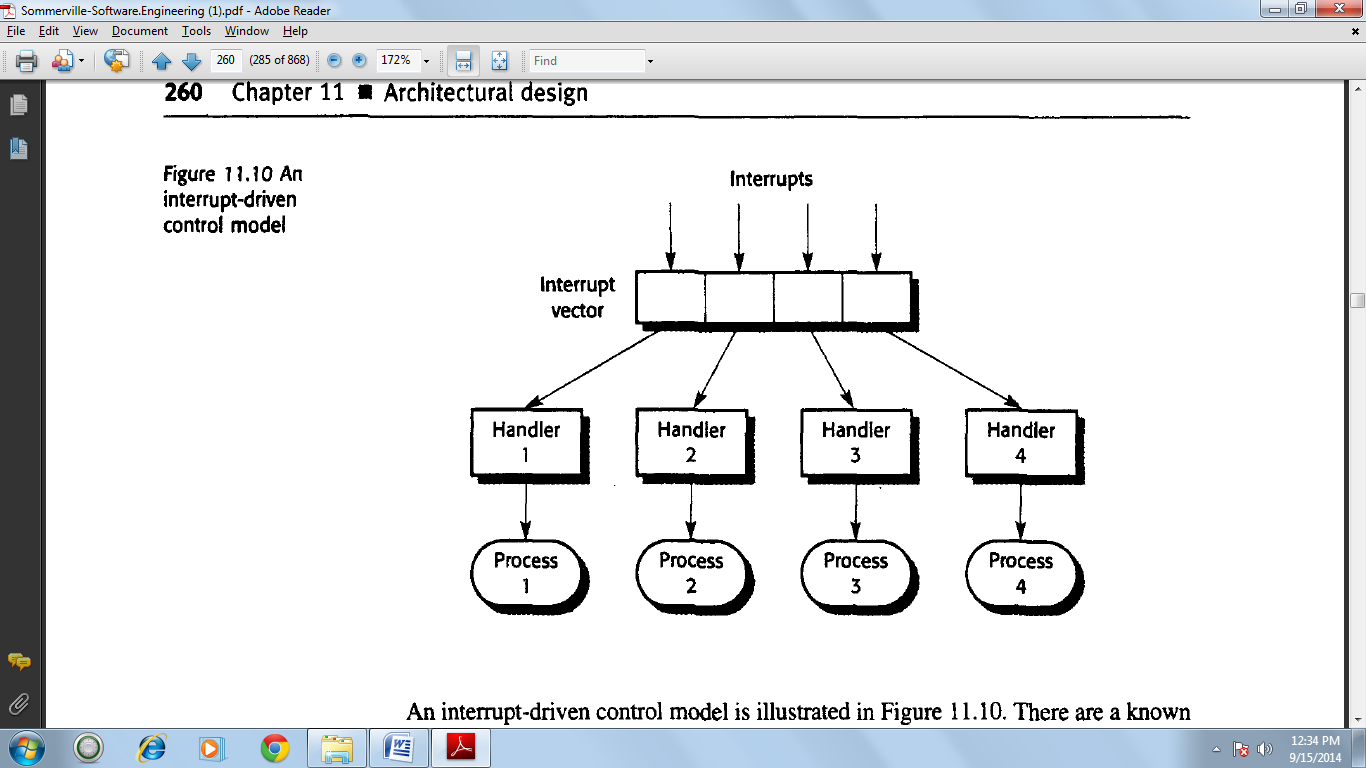
When an interrupt of a particular type is received, a hardware switch causes control to be transferred immediately to its handler. This interrupt handler may then start or stop other processes in response to the event signaled by the interrupt.

This model is mostly used in real-time systems where an immediate response to some event is necessary. It may be combined with the centralised management model.

The central manager handles the normal running of the system with interrupt-based control for emergencies.

The advantage of this approach is that it allows very fast responses to events to be implemented. Its disadvantages are that it is complex to program and difficult to validate. It may be impossible to replicate patterns of interrupt timing during system testing. It can be difficult to change systems developed using this model if the number of interrupts is limited by the hardware. Once this limit is reached, no other types of events can be handled. You can sometimes get around this limitation by mapping several types of events onto a single interrupt. The handler then works out which event has occurred. However, interrupt mapping may be impractical if a very fast response to individual interrupts is required.

**An interrupt-driven control model**



**Modular decomposition**

After a structural architecture has been designed the next stage of the architectural design process is the decomposition of sub systems into modules the components in modules is usually smaller than sub systems.

The two models which may be used when decomposing a sub system into modules.

1. An object Oriented model

The system is decomposed into a set of communicating objects.

1. A data flow model

The system is decomposed into functional modules which accept input data and transform it, in some way to output data. This is also called a pipeline approach.

**Object-oriented design**

An object-oriented system is made up of interacting objects that maintain their own local state and provide operations on that state. The representation of the state is private and cannot be accessed directly from outside the object. Object oriented design processes involve designing object classes and the relationships between these classes. These classes define the objects in the system and their interactions.

When the design is realised as an executing program, the objects are created dynamically from these class definitions.

Object-oriented design is part of object-oriented development where an object oriented strategy is used throughout the development process:

***Object-oriented analysis***is concerned with developing an object-oriented model of the application domain. The objects in that model reflect the entities and operations associated with the problem to be solved.

***Object-oriented design***is concerned with developing an object-oriented model of a software system to implement the identified requirements. The objects in an object-oriented design are related to the solution to the problem. There may be close relationships between some problem objects and some solution objects, but the designer inevitably has to add new objects and to transform problem objects to implement the solution.

***Object-oriented programming***is concerned with realising a software design using an object-oriented programming language, such as Java. An object-oriented programming language provides constructs to define object classes and a run-time system to create objects from these classes.

**Objects and object classes**

The terms *object* and *object-oriented* are applied to different types of entity, design methods, systems and programming languages. There is a general acceptance that an object is an encapsulation of information, and this is reflected in my definition of an object and an object class:

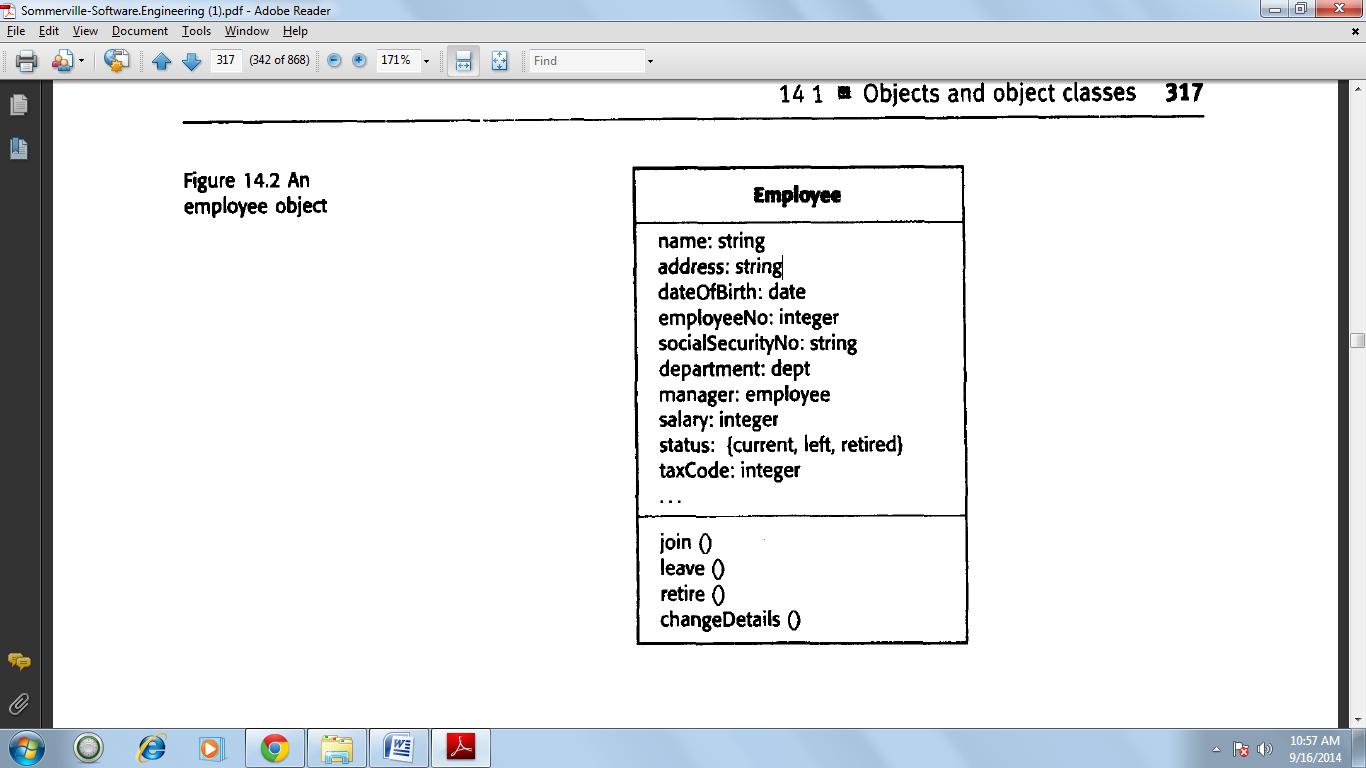
*An object is an entity that has a state and a defined set of operations that operate on that state. The state is represented as a set of object attributes.*

*The operations associated with the object provide services to other objects (clients) that request these services when some computation is required.*

*Objects are created according to an object class definition. An object class definition is both a type specification and a template for creating objects. It includes declarations of all the attributes and operations that should be associated with an object of that class.*

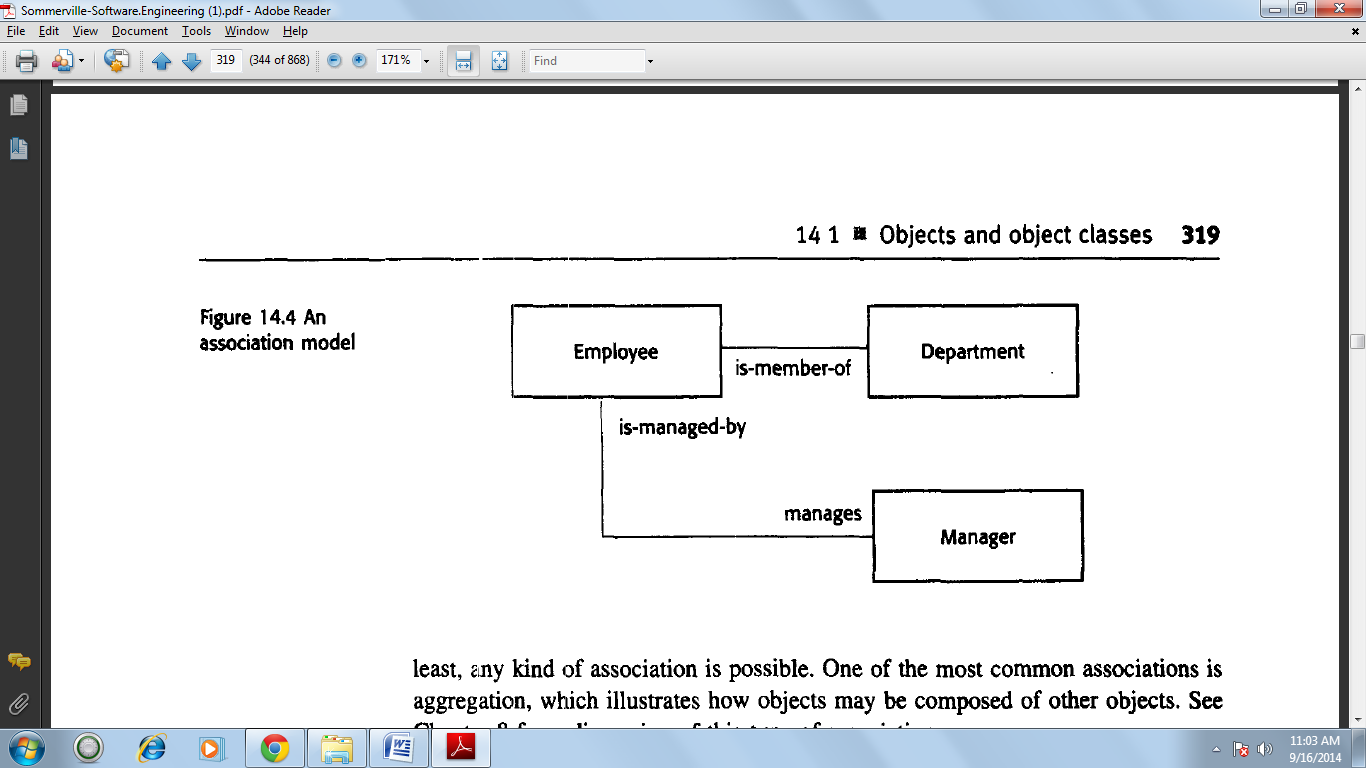
Objects communicate by requesting services (calling methods) from other objects and, if necessary, by exchanging the information required for service provision. The copies of information needed to execute the service and the results of service execution are passed as parameters. Some examples of this style of communication are:

**An employee object**



**An association model**

Association is a very general relationship and is often used in the UML to indicate that either an attribute of an object is an associated object or the implementation of an object method relies on the associated object. However, in principle at least, any kind of association is possible. One of the most common associations is aggregation, which illustrates how objects may be composed of other objects.



**Concurrent objects**

Conceptually, all object requests a service from another object by sending a service request' message to that object. There is no requirement for serial execution where one object waits for completion of a requested service. Consequently, the general model of object interaction allows objects to execute concurrently as parallel processes. These objects may execute on the same computer or as distributed objects on different machines.

In practice, most object-oriented programming languages have as their default a serial execution model where requests for object services are implemented in the same way as function calls. Therefore, when an object called the List is created from a normal object class, you write in Java:

There are two kinds of concurrent object implementation:

*1****. Servers***where the object is realised as a parallel process with methods corresponding to the defined object operations. Methods start up in response to an external message and may execute in parallel with methods associated with other objects. When they have completed their operation, the object suspends itself and waits for further requests for service.

*2.* ***Active objects***where the state of the object may be changed by internal operations executing within the object itself. The process representing the object continually executes these operations so never suspends itself.

**An object-oriented design process**

In this section, I illustrate the process of object-oriented design by developing an example design for the control software that is embedded in an automated weather station. As I discussed in the introduction, there are several methods of object-oriented design with no definitive 'best' method or design process. The process that I cover here is a general one that incorporates activities common to most OOD processes.

The general process that I use here for object-oriented design has a number of stages:

1. Understand and define the context and the modes of use of the system.

2. Design the system architecture.

3. Identify the principal objects in the system.

4. Develop design models.

5. Specify object interfaces.

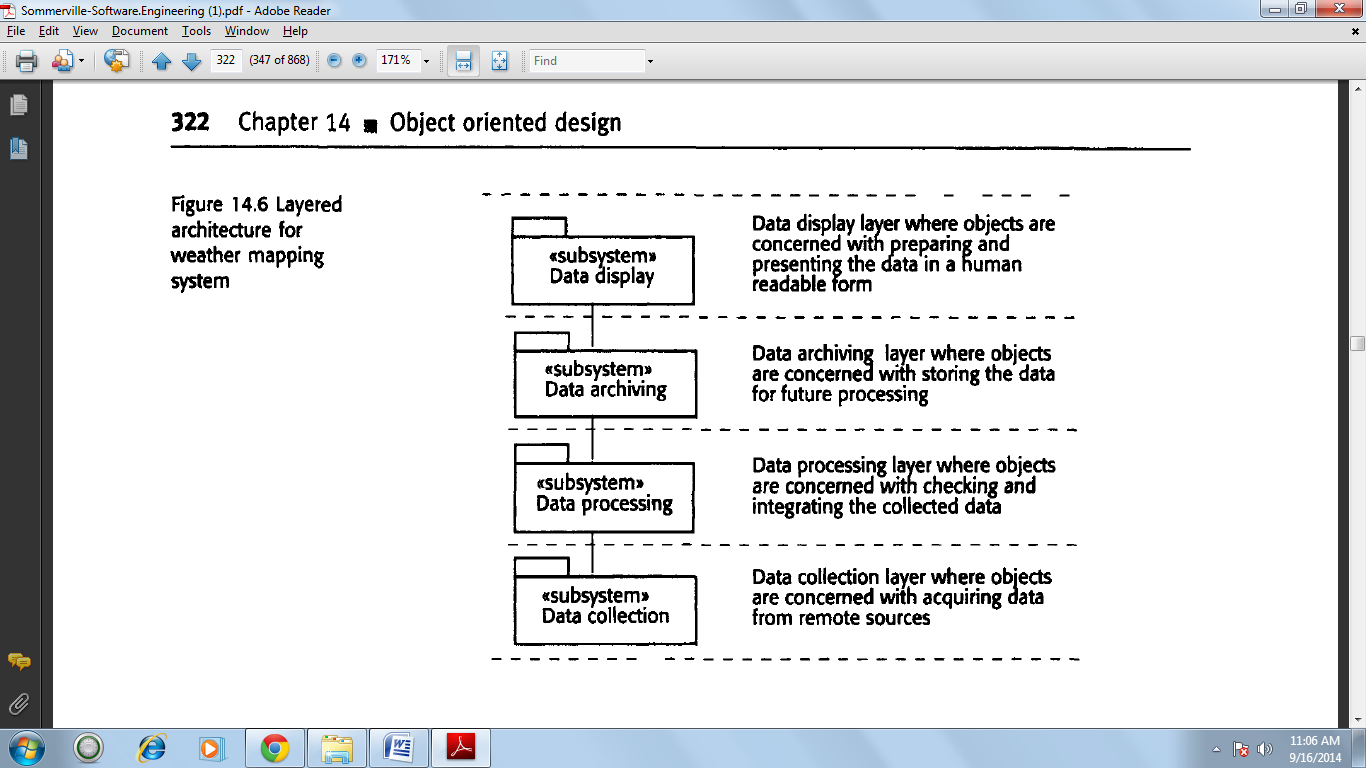
*A weather mapping system is required to generate weather maps on a regular basis using data collected from remote, unattended weather stations and other data sources such as weather observers, balloons and satellites.*

*Weather stations transmit their data to the area computer in response to a request from that machine.*

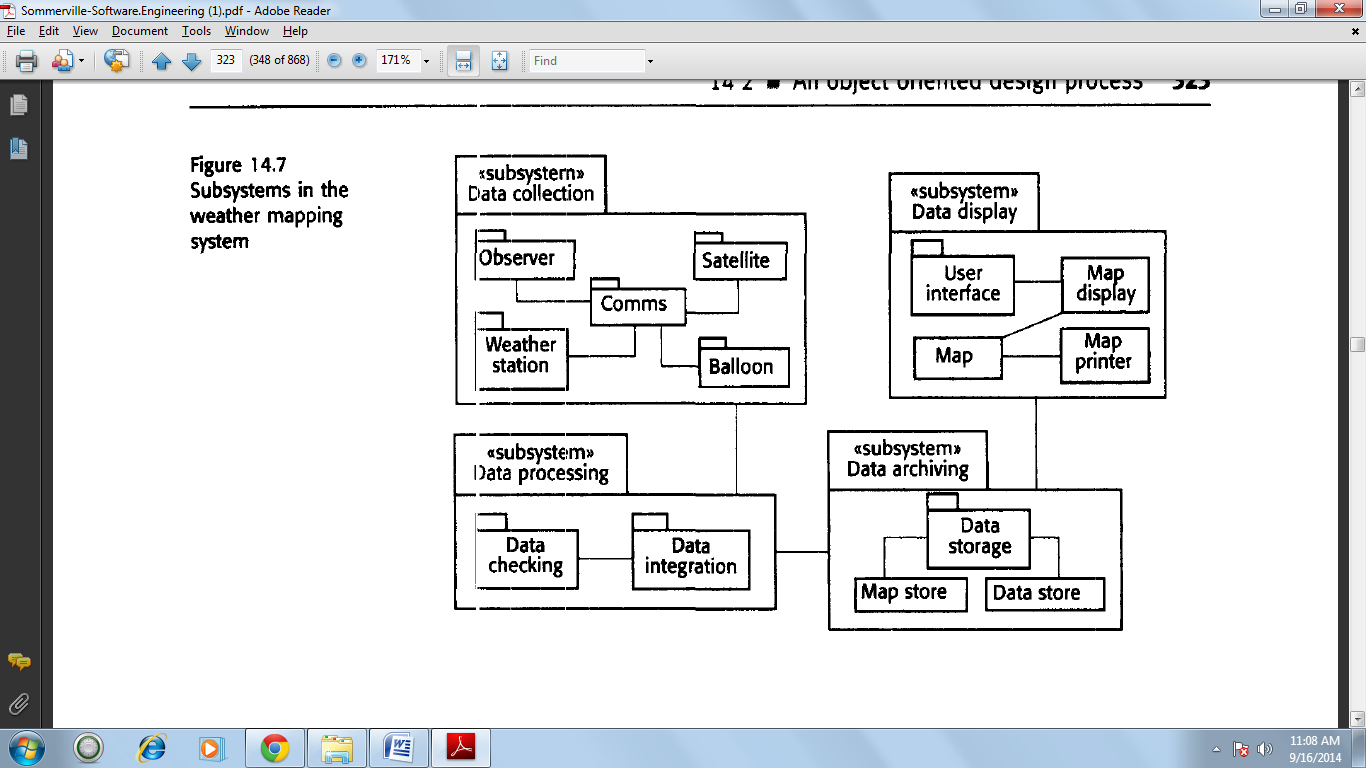
*The area computer system validates the collected data and integrates the data from different sources. The integrated data is archived and, using data from this archive and a digitized map database, a set of local weather maps is created.*

*Maps may be printed for distribution on a special-purpose map printer or may be displayed in a number of different formats.*

**Layered architecture for weather mapping system**



**Subsystems in the weather mapping system**



**System context and models of use**

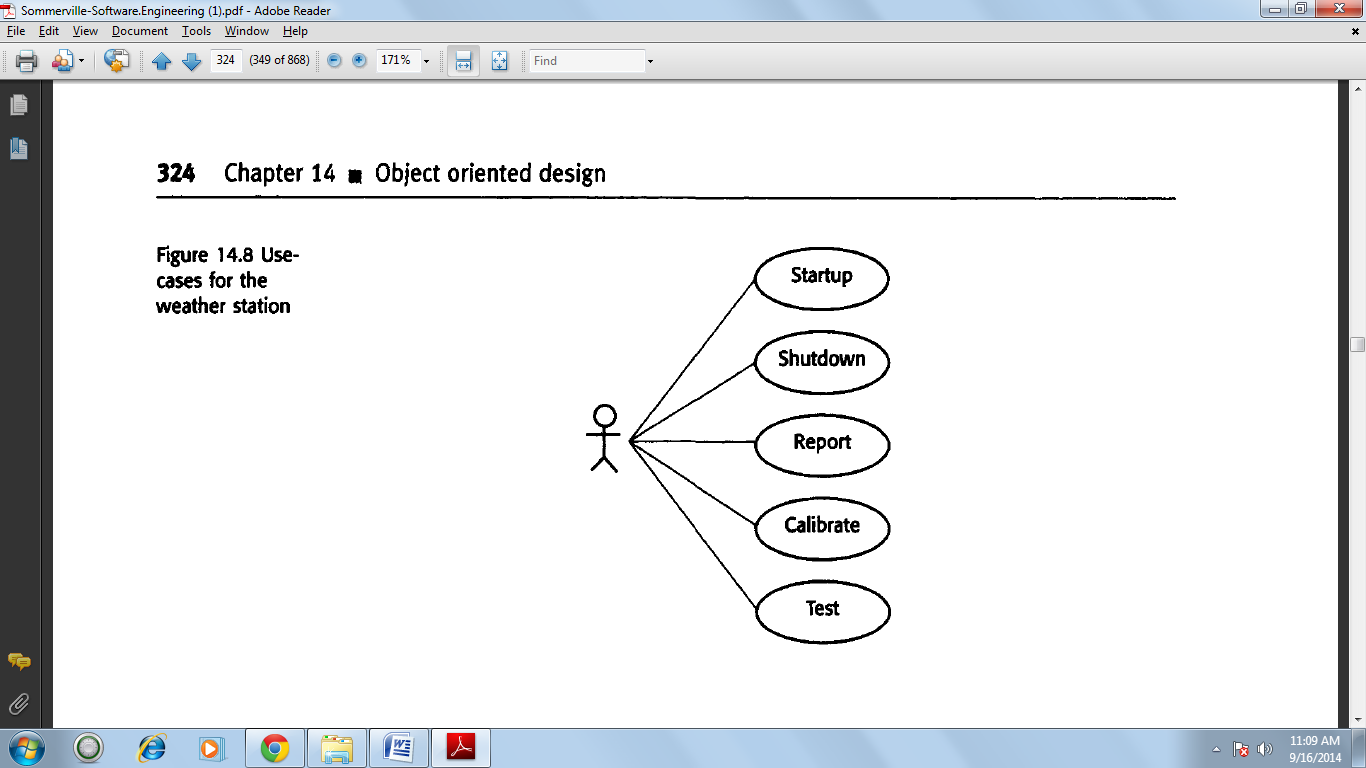
The first stage in any software design process is to develop an understanding of the relationships between the software that is being designed and its external environment. You need this understanding to help you decide how to provide the required system functionality and how to structure the system to communicate with its environment.

The system context and the model of system use represent two complementary models of the relationships between a system and its environment:

1. The system context is a static model that describes the other systems in that environment.

2. The model of the system use is a dynamic model that describes how the system actually interacts with its environment.

**Use cases for the weather station**



**Architectural design**

Once the interactions between the software system that is being designed and the systems, environment have been defined, you can use this information as a basis for designing the system architecture. Of course, you need to combine this with your general knowledge of the principles of architectural design and with more detailed domain knowledge.

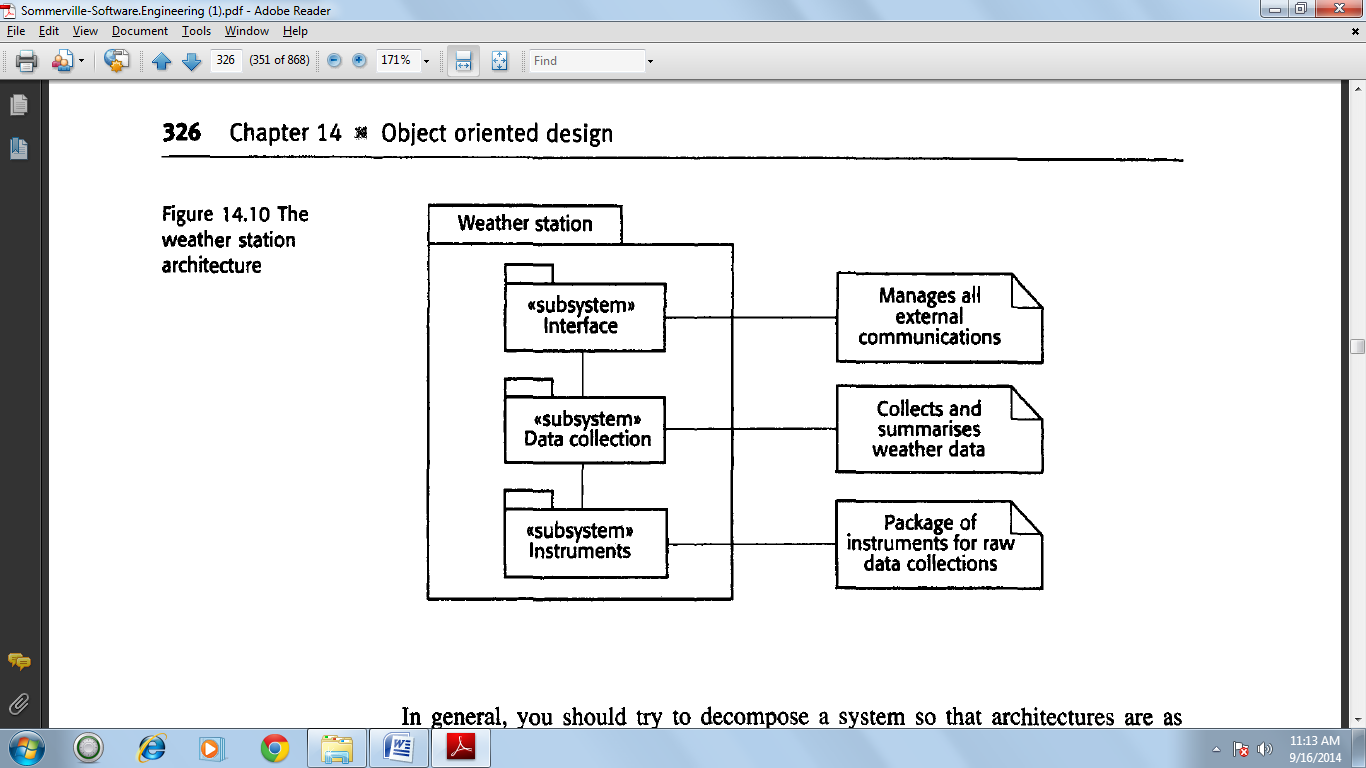
The three layers in the weather station software are:

1. ***The interface layer***that is concerned with all communications with other parts of the system and with providing the external interfaces of the system.

2. ***The data collection layer***that is concerned with managing the collection of data from the instruments and with summarising the weather data before transmission to the mapping system.

3. ***The instruments layer***that is an encapsulation of all of the instruments used to collect raw data about the weather conditions.

**The weather station architecture**



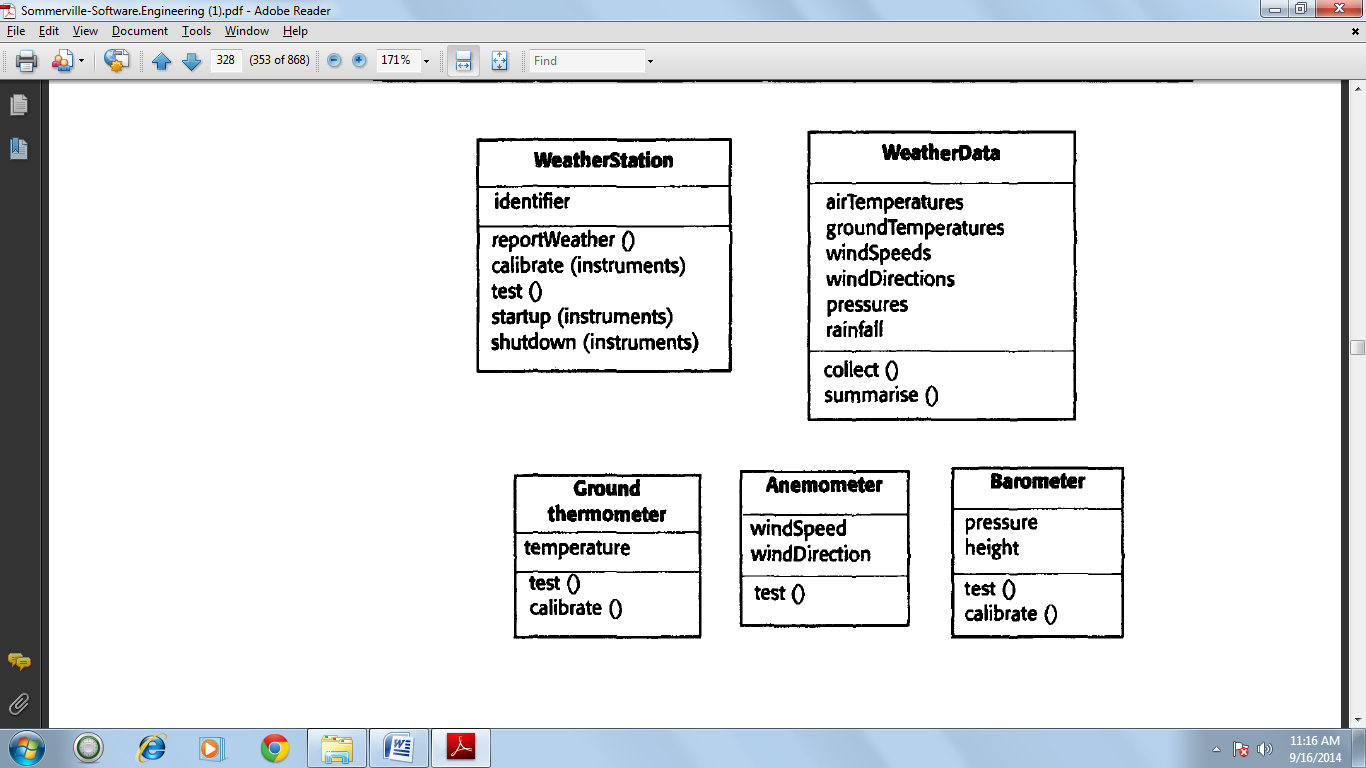
**Object identification**

These objects are related to the levels in the system architecture.

1. The Weather Station object class provides the basic interface of the weather station with its environment. I use a single object class to encapsulate all of these interactions, but in other designs you may chose to design the system interface: as several different classes.

2. The Weather Data object class encapsulates the summarised data from the instruments in the weather station. Its associated operations are concerned with collecting and summarising the data that is required.

3. The Ground thermometer, Anemometer and Barometer object classes are directly related to instruments in the system. They reflect tangible hardware entities in the system and the operations are concerned with controlling that hardware.



**Design models**

Design models show the objects or object classes in a system and, where appropriate, the relationships between these entities. Design models essentially are the design. They are the bridge between the requirements for the system and the system implementation. This means that there are conflicting requirements on these models. They have to be abstract so that unnecessary detail doesn't hide the relationships between them and the system requirements. However, they also have to include enough detail for programmers to make implementation decisions.

In general, you get around this conflict by developing models at different levels of detail. Where there are close links between requirements engineers, designers and organisation, then abstract models may be all that are required. Specific design decisions may be made as the system is implemented. When the links between system specifiers designers and programmers are indirect (e.g., where a system is being designed in one part of an organisation but implemented elsewhere), then more detailed models may be required.

There are two types of design models that should normally be produced to describe an object-oriented design:

*1. Static models* describe the static structure of the system using object classes and their relationships. Important relationships that may be documented at this stage are generalisation relationships, uses/used-by relationships and composition relationships.

*2. Dynamic models* describe the dynamic structure of the system and show the interactions between the system objects (not the object classes). Interactions that may be documented include the sequence of service requests made by objects and the way in which the state of the system is related to these object interactions.

**The models that I discuss in this section are:**

*1.* ***Subsystem models***that show logical groupings of objects into coherent sub-systems. These are represented using a form of class diagram where each sub-system is shown as a package. Subsystem models are static models.

*2.* ***Sequence models***that show the sequence of object interactions. These are represented using a UML sequence or a collaboration diagram. Sequence models are dynamic: models.

*3.* ***State machine models***that show how individual objects change their state in response to events. These are represented in the UML using state chart diagrams. State machine models are dynamic models.

**Object interface specification**

An important part of any design process is the specification of the interfaces between the components in the design. You need to specify interfaces so that objects and sub-systems can be designed in parallel. Once an interface has been specified, the developers of other objects may assume that interface will be implemented. You should try to avoid including details of the interface representation in an interface design. The representation should be hidden and object operations provided to access and update the data. If the representation is hidden, it can be changed without affecting the objects that use these attributes. For example, an array representation of a stack may be changed to a list representation without affecting other objects that use the stack. By contrast, it often makes sense to expose the attributes in a static design model, as this is the most compact way of illustrating essential characteristics of the objects.

**User interface design**

Good user interface design is critical to the success of a system. An interface that is difficult to use will, at best, result in a high level of user errors. At worst users will simply refuse to use the software system irrespective of its functionality If information is presented in a confusing or misleading way, users may misunderstand the meaning of information. They may initiate a sequence of actions that corrupt data or even cause catastrophic system failure.

**The User-Interface design process**

Analyses and understand user activities

Produce paper based design prototype

Evaluate design with end-users

Design Prototype

Produce dynamic design prototype

Evaluate design with end-users

Executable prototype

Implement final user interface

**User Interface design principles**

1. **User familiarity**

The interface should use terms and concepts which are drawn from the experience of the people who will make most use of the system.

1. **Consistency**

The interface should be consistent in that wherever possible comparable operations should be activated in the same way.

1. **Minimal Surprise**

User should never be surprised by the behavior of a system.

1. **Recoverability**

The interface should include mechanisms to allow users to recover from errors.

1. **User guidance**

The interface should provide meaningful feedback when errors occur and provide context sensitive user help facilities.

1. **User diversity**

The interface should provide appropriate interaction facilities for different types of system user.

User interfaces should contain facilities allowing users to recover from their mistakes. These can be of two kinds.

1. Confirmation of destructive actions

If users specify an action which is potentially destructive, they should be asked to confirm that this is really what they want before any information is destroyed.

1. The provision of an undo facility

Undo restores the system to a state before the action occurred. Multiple levels of undo are useful as users don’t always recognize immediately that a mistake has been made.

**User interaction**

User interaction means issuing commands and associated data to the computer system the only way to do this was through a command line interface where a special purpose language was used to communicate with the machine. User interaction is classified to five primary styles.

1. **Direct Manipulation**

Where the user interacts directly with objects on the screen.

1. **Menu selection**

Where a user selects a command from a list of possibilities. It is often the case that another screen object is selected at the same time and the command operates on that object. In this approach to delete a file, the user selects the file, and then selects the delete command.

1. **Form\_fill\_in**

Where a user fills in the fields of a form. Some fields may have action “buttons” that, when pressed, cause some action to be initiated. It would be artificial to delete a file using a form based interface. It would involve filling in the name of the file, then “pressing “ a delete button.

1. **Command Language**

Where the user issues a special command and associated parameters to instruct the system what to do. To delete a file the user issues a delete command with the filename as a parameter.

1. **Natural Language**

Where the user issues a command in Natural Language. To delete a file the user might therefore type delete the file named XXX.

Information Presentation

All interactive systems have to provide some way of presenting information to users. The information presentation may simply be a direct representation of the input information.

Users can interact with each presentation using a style that is appropriate to it. The data to be displayed is encapsulated in a model object. Each model object may have a number of separate view objects associated with it where each view is a different display representation of the model. Information that does not change during a session may be presented either graphically or a text depending on the application. Textual presentation takes up less screen space but cannot be read at a glance.

Information should be represented as text when precise numeric information is required and the information changes relatively slowly.

**Colour in interface design**

Colour can improve user interfaces by helping users understand and manage complexity. As a general principle, user interface designers should be conservative in their use of colour in user displays. The most important of these are.

1. Limit the number of colours used and be conservative how these are used.
2. Use colour change to show a change in system status.
3. Use colour coding to support the task which users are trying to perform.
4. Use colur coding in a thoughtful and consistent way.

**User Support**

A design principle suggested in the first section of this chapter was that user interfaces should always provide some form of online help system. Help systems are one fact of a general part of user interface design, namely the provision of user guidance which covers three areas”

1. The messages produced by the system in response to user actions
2. The online help system
3. The documentation provided with the system.