KTU NOTES

The learning companion.

KTU STUDY MATERIALS | SYLLABUS | LIVE NOTIFICATIONS | SOLVED QUESTION PAPERS

MODULE II

Functional and non-functional requirements, Requirements engineering processes. Requirements elicitation, Requirements validation, Requirements change, Traceability Matrix. Developing use cases, Software Requirements Specification Template, Personas, Scenarios, User stories, Feature identification. Design concepts - Design within the context of software engineering, Design Process, Design concepts, Design Model. Architectural Design - Software Architecture, Architectural Styles, Architectural considerations, Architectural Design Component level design - What is a component?, Designing Class-Based Components, Conducting Component level design, Component level design for web-apps. Template of a Design Document as per “IEEE Std 1016-2009 IEEE Standard for Information Technology Systems Design Software Design Descriptions”. Case study: The Ariane 5 launcher failure. 5

- The requirements for a system are the descriptions of what the system should do—the services that it provides and the constraints on its operation.
- The process of finding out, analyzing, documenting and checking these services and constraints is called **requirements engineering (RE).**
- A clear separation should be made between different level of requirements.

❑**User requirements** are statements, in a natural language plus diagrams, of what services the system is expected to provide to system users and the constraints under which it must operate.

❑**System requirements** are more detailed descriptions of the software system's functions, services, and operational constraints.

- Different levels of requirements are useful because they communicate information about the system to different types of reader

User Requirement Definition

1. The MHC-PMS shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.

System Requirements Specification

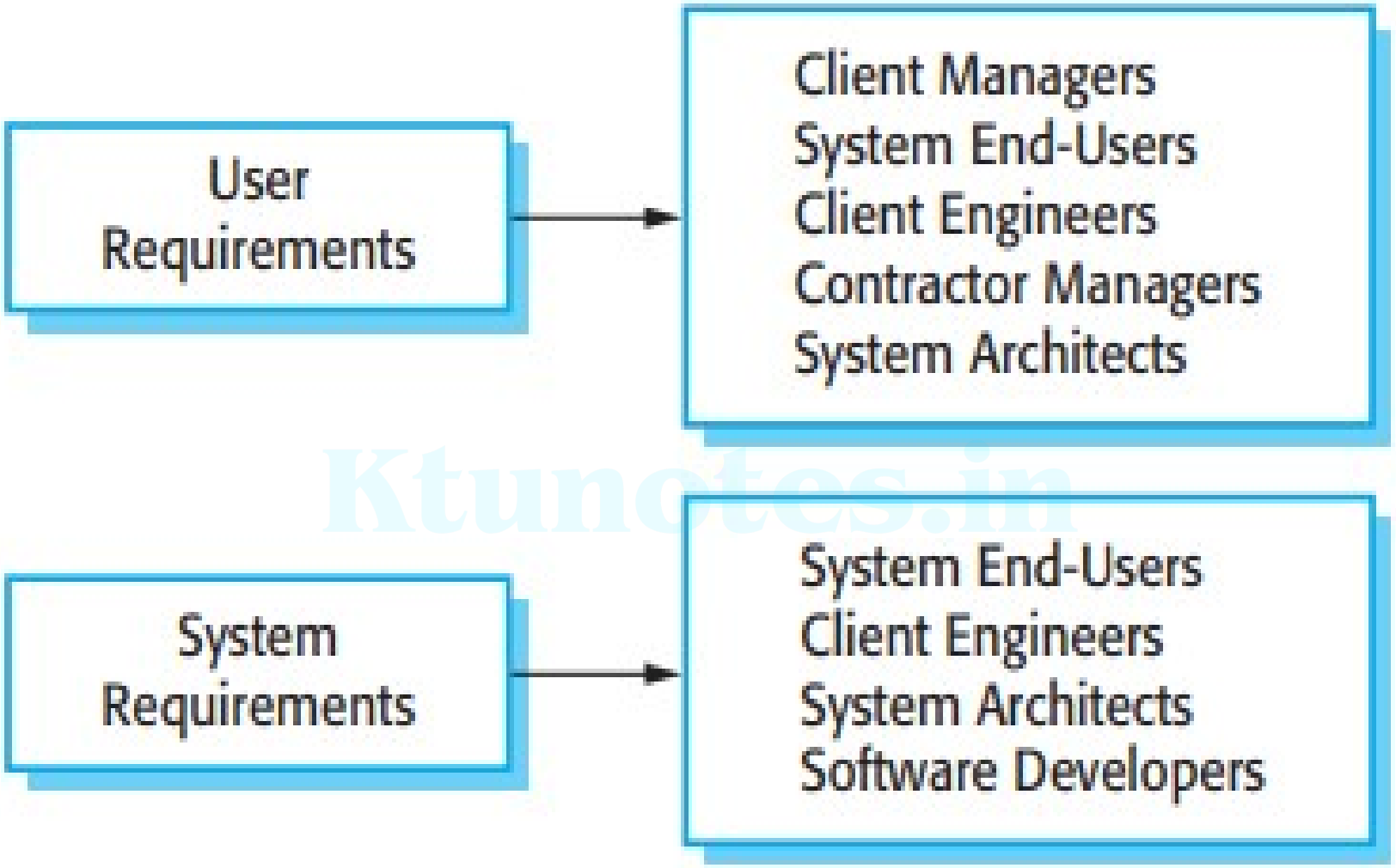
1.1 On the last working day of each month, a summary of the drugs prescribed, their cost, and the prescribing clinics shall be generated.

1.2 The system shall automatically generate the report for printing after 17.30 on the last working day of the month.

1.3 A report shall be created for each clinic and shall list the individual drug names, the total number of prescriptions, the number of doses prescribed, and the total cost of the prescribed drugs.

1.4 If drugs are available in different dose units (e.g., 10 mg, 20 mg) separate reports shall be created for each dose unit.

1.5 Access to all cost reports shall be restricted to authorized users listed on a management access control list.



- There is a clearly identifiable requirements engineering phase before the implementation of the system begins.
- The outcome is a requirements document, which may be part of the system development contract

Functional and non-functional requirements

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.
- state what the system should not do.

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 constraints imposed by standards.
- Non-functional requirements often apply to the system as a whole, rather than individual system features or service

Requirements are not independent and that one requirement often generates or constrains other requirements.

FUNCTIONAL REQUIREMENTS

WHAT THE SYSTEM SHOULD DO

PRODUCT FEATURES

USER REQUIREMENTS

NON-FUNCTIONAL REQUIREMENTS

HOW THE SYSTEM SHOULD DO IT

PRODUCT PROPERTIES

USER EXPECTATIONS

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 organization when writing requirements.
- When expressed as user requirements, functional requirements are usually described in an abstract way that can be understood by system users.
- More specific functional system requirements describe the system functions, its inputs and outputs, exceptions, etc., in detail.
- Functional system requirements vary from general requirements (covering what the system should do) to very specific requirements (reflecting local ways of working or an organization's existing systems).
- Imprecision in the requirements specification is the cause of many software engineering problems.
- the functional requirements specification of a system should be both complete and consistent. Completeness means that all services required by the user should be defined. Consistency means that requirements should not have contradictory definitions.

Requirements for the MHC-PMS system, used to maintain information about patients receiving treatment for mental health problems:

1. A user shall be able to search the appointments lists for all clinics.
2. The system shall generate each day, for each clinic, a list of patients who are expected to attend appointments that day.
3. Each staff member using the system shall be uniquely identified by his or her eight-digit employee number.

Non-functional requirements

- Requirements that are not directly concerned with the specific services delivered by the system to its users.
- relate to emergent system properties such as reliability, response time, and store occupancy.
- define constraints on the system implementation such as the capabilities of I/O devices or the data representations used in interfaces with other systems.
- Non-functional requirements, such as performance, security, or availability, usually specify or constrain characteristics of the system as a whole.
- Non-functional requirements are often more critical than individual functional requirement.
- failing to meet a non-functional requirement can mean that the whole system is unusable. For example, if an aircraft system does not meet its reliability requirements, it will not be certified as safe for operation

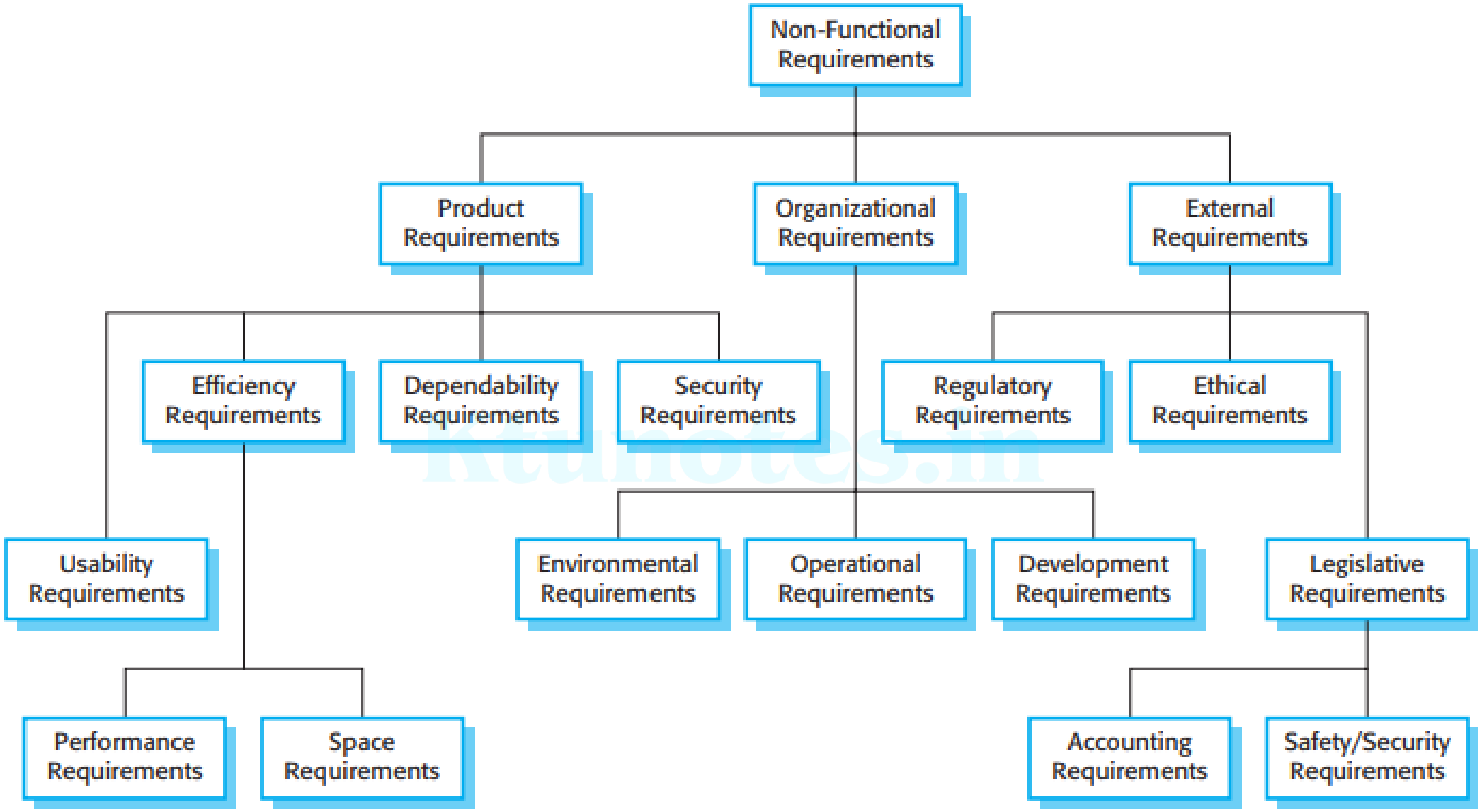
Non-functional requirements

The implementation of functional requirements may be diffused throughout the system. There are two reasons for this:

1. Non-functional requirements may affect the overall architecture of a system rather than the individual components

2. A single non-functional requirement, such as a security requirement, may generate a number of related functional requirements that define new system services that are required. In addition, it may also generate requirements that restrict existing requirements

Non-functional requirements arise through user needs, because of budget constraints, organizational policies, the hardware systems, or external factors such as safety regulations or privacy legislation



Classification of non-functional requirements.

non-functional requirements may come from

- required characteristics of the software (product requirements)
- the organization developing the software (organizational requirements)
- from external

Product requirements :

specify or constrain the behavior of the software.

Examples :performance requirements on how fast the system must execute and how much memory it requires, security requirements, and usability requirements.

Organizational requirements :

derived from policies and procedures in the customer's and developer's organization.

Examples:

operational process requirements -how the system will be used

development process requirements- specify the programming language

environmental requirements -operating environment of the system.

External requirements

requirements that are derived from factors external to the system and its development process.

Examples:

regulatory requirements -what must be done for the system to be approved for use by a regulator, such as a central bank;

legislative requirements - to ensure that the system operates within the law;

ethical requirements- ensure that the system will be acceptable to its users and the general public

- product, organizational, and external requirements taken from the MHC-PMS

PRODUCT REQUIREMENT

The MHC-PMS shall be available to all clinics during normal working hours (Mon–Fri, 08.30–17.30). Downtime within normal working hours shall not exceed five seconds in any one day.

ORGANIZATIONAL REQUIREMENT

Users of the MHC-PMS system shall authenticate themselves using their health authority identity card.

EXTERNAL REQUIREMENT

The system shall implement patient privacy provisions as set out in HStan-03-2006-priv.

- A common problem with non-functional requirements is that users or customers often propose these requirements as general goals, such as ease of use, the ability of the system to recover from failure, or rapid user response.
- Goals set out good intentions but cause problems for system developers as they leave scope for interpretation and subsequent dispute once the system is delivered

- write non-functional requirements quantitatively so that they can be objectively tested.
- Metrics used to specify non-functional system properties.

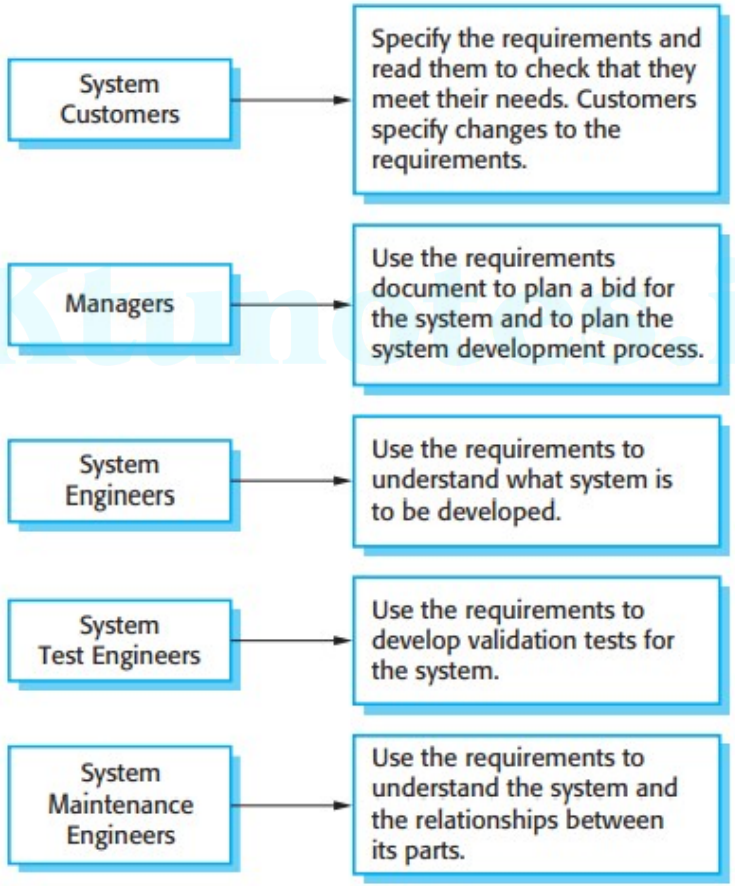
Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Mbytes Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

- customers for a system often find it difficult to translate their goals into measurable requirements.
- For some goals, such as maintainability, there are no metrics that can be used.
- Even when quantitative specification is possible, customers may not be able to relate their needs to these specifications.
- Non-functional requirements often conflict and interact with other functional or non-functional requirements.
- It is difficult, to separate functional and non-functional requirements in the requirements document.
- Non-functional requirements such as reliability, safety, and confidentiality requirements are particularly important for critical systems.

The software requirements document(software requirements specification or SRS)

- 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**.
- Requirements documents are essential when an outside contractor is developing the software system.
- Agile development methods argue that requirements change so rapidly that a requirements document is out of date as soon as it is written, so the effort is largely wasted.
- Extreme Programming (Beck, 1999) collect user requirements incrementally and write these on cards as user stories. The user then prioritizes requirements for implementation in the next increment of the system.
- For business systems where requirements are unstable, this approach is a good one

Users of a requirements document



- 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.
- The level of detail that should be included in a requirements document depends on the type of system that is being developed and the development process used.
- Critical systems need to have detailed requirements because safety and security have to be analyzed in detail.
- When the system is to be developed by a separate company (e.g., through outsourcing), the system specifications need to be detailed and precise.
- If an inhouse, iterative development process is used, the requirements document can be much less detailed and any ambiguities can be resolved during development of the system.

- Structure of a requirements document

Chapter	Description
Preface	This should define the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version.
Introduction	This should describe the need for the system. It should briefly describe the system's functions and explain how it will work with other systems. It should also describe how the system fits into the overall business or strategic objectives of the organization commissioning the software.
Glossary	This should define the technical terms used in the document. You should not make assumptions about the experience or expertise of the reader.
User requirements definition	Here, you describe the services provided for the user. The non-functional system requirements should also be described in this section. This description may use natural language, diagrams, or other notations that are understandable to customers. Product and process standards that must be followed should be specified.
System architecture	This chapter should present a high-level overview of the anticipated system architecture, showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.
System requirements specification	This should describe the functional and non-functional requirements in more detail. If necessary, further detail may also be added to the non-functional requirements. Interfaces to other systems may be defined.
System models	This might include graphical system models showing the relationships between the system components, the system, and its environment. Examples of possible models are object models, data-flow models, or semantic data models.
System evolution	This should describe the fundamental assumptions on which the system is based, and any anticipated changes due to hardware evolution, changing user needs, and so on. This section is useful for system designers as it may help them avoid design decisions that would constrain likely future changes to the system.
Appendices	These should provide detailed, specific information that is related to the application being developed; for example, hardware and database descriptions. Hardware requirements define the minimal and optimal configurations for the system. Database requirements define the logical organization of the data used by the system and the relationships between data.
Index	Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, and

Requirements engineering process

requirements engineering processes may include four high-level activities.

- assessing if the system is useful to the business (**feasibility study**)
- discovering requirements (**elicitation and analysis**)
- converting these requirements into some standard form (**specification**)
- checking that the requirements actually define the system that the customer wants (**validation**).

Requirements engineering is an iterative process in which the activities are interleaved.

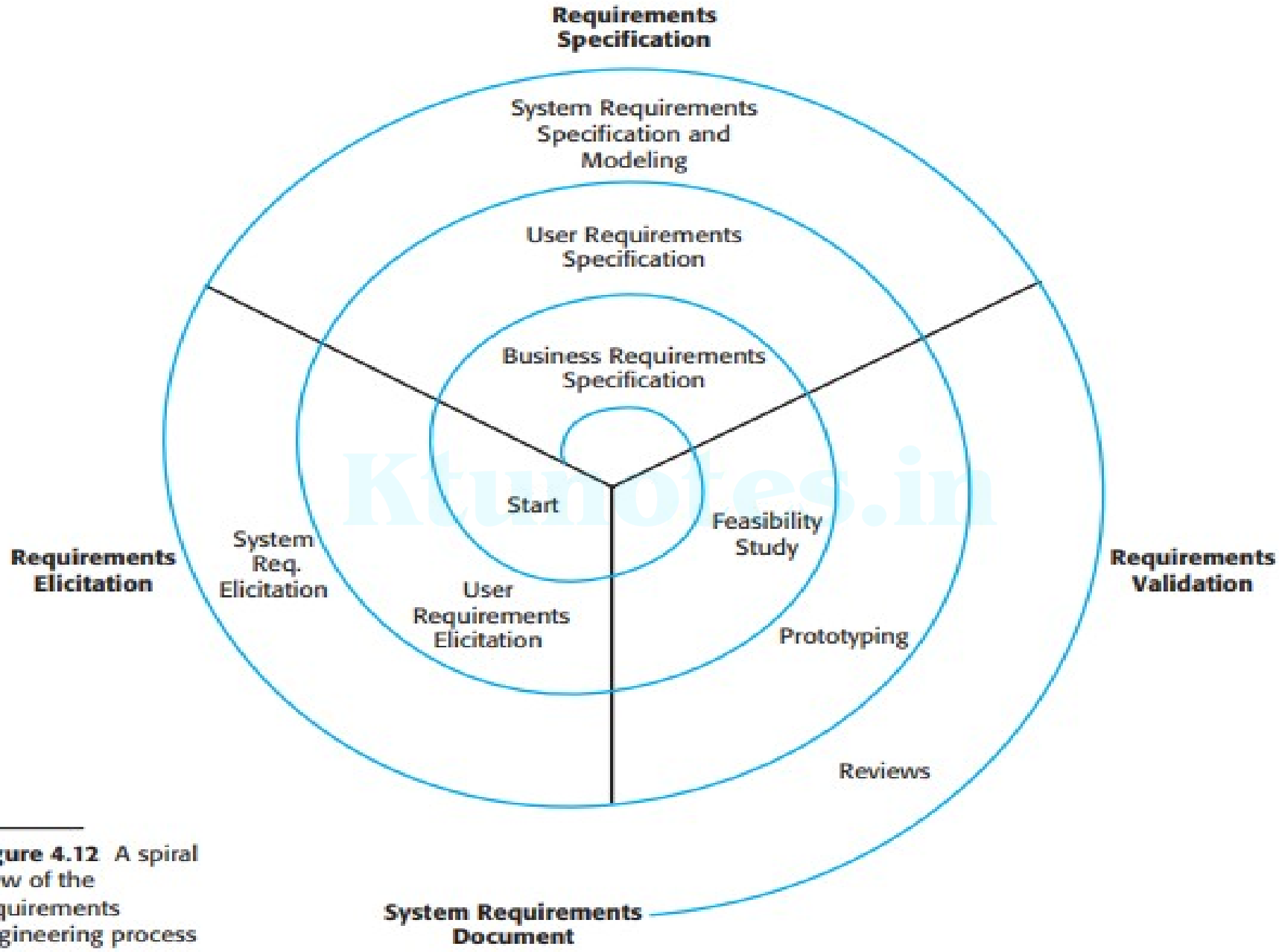


Figure 4.12 A spiral view of the requirements engineering process

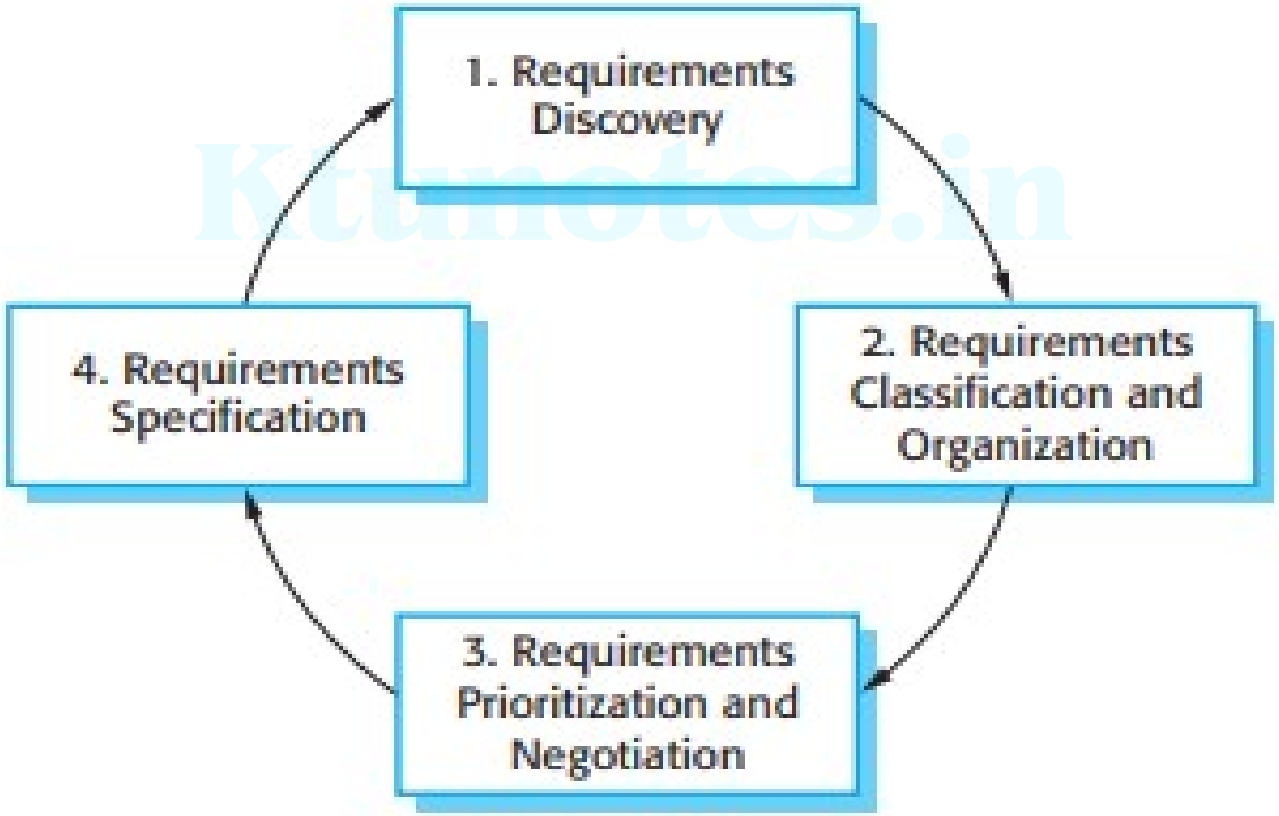
Requirements engineering process

- The activities are organized as an iterative process around a spiral, with the output being a system requirements document.
- The amount of time and effort devoted to each activity in each iteration depends on the stage of the overall process and the type of system being developed.
- Early in the process, most effort will be spent on understanding high-level business and non-functional requirements, and the user requirements for the system.
- Later in the process, in the outer rings of the spiral, more effort will be devoted to eliciting and understanding the detailed system requirements

Requirements elicitation and analysis

- After an initial feasibility study, 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 different kinds of people in an organization.
- A system stakeholder is anyone who should have some direct or indirect influence on the system requirements.
- Stakeholders include endusers, anyone else in an organization who will be affected by it,engineers who are developing or maintaining other related systems, business managers, domain experts, and trade union representatives.

Requirements elicitation and analysis



Requirements elicitation and analysis

- **Requirements discovery:**

process of interacting with stakeholders of the system to discover their requirements.

- **Requirements classification and organization :**

-This activity takes the unstructured collection of requirements, groups related requirements, and organizes them into coherent clusters.

- identify sub-systems and to associate requirements with each sub-system.

- **Requirements prioritization and negotiation:**

- concerned with prioritizing requirements and finding and resolving requirements conflicts through negotiation.

- **Requirements specification :**

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

Requirements elicitation and analysis

Eliciting and understanding requirements from system stakeholders is a difficult process for several reasons:

1. Stakeholders often don’t know what they want from a computer system except in the most general terms; they may find it difficult to articulate what they want the system to do; they may make unrealistic demands because they don’t know what is and isn’t feasible.

2. Stakeholders in a system naturally express requirements in their own terms and with implicit knowledge of their own work. Requirements engineers, without experience in the customer’s domain, may not understand these requirements.

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

4. Political factors may influence the requirements of a system. Managers may demand specific system requirements because these will allow them to increase their influence in the organization.

Requirements elicitation Techniques

- Requirements discovery
- Interviewing
- Scenarios
- Use cases
- Ethnography

Requirements discovery

- Requirements discovery (sometime called requirements elicitation) is the process of gathering information about the required system 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.
- Interact with stakeholders through interviews and observation and you may use scenarios and prototypes to help stakeholders understand what the system will be like.
- Stakeholders range from end-users of a system through managers to external stakeholders such as regulators, who certify the acceptability of the system.

system stakeholders for the mental healthcare patient information system include:

1. Patients whose information is recorded in the system.
2. Doctors who are responsible for assessing and treating patients.
3. Nurses who coordinate the consultations with doctors and administer some treatments.
4. Medical receptionists who manage patients' appointments.
5. IT staff who are responsible for installing and maintaining the system.
6. A medical ethics manager who must ensure that the system meets current ethical guidelines for patient care.
7. Healthcare managers who obtain management information from the system.
8. Medical records staff who are responsible for ensuring that system information can be maintained and preserved, and that record keeping procedures have been properly implemented.

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 currently use and the system to be developed.
- Requirements are derived from the answers to these questions.
- Interviews may be of two types:

❑ **Closed interviews**, where the stakeholder answers a pre-defined set of questions.

❑ **Open interviews**, in which there is no pre-defined agenda. The requirements engineering team explores a range of issues with system stakeholders and hence develop a better understanding of their needs.

- Interviews with stakeholders are normally a mixture of both of these.
- Completely open-ended discussions rarely work well.
- We have to ask some questions to get started and to keep the interview focused on the system to be developed
- Interviews are good for getting an overall understanding of what stakeholders do, how they might interact with the new system, and the difficulties that they face with current systems.

Interviewing

It can be difficult to elicit domain knowledge through interviews for two reasons:

1. All application specialists use terminology and jargon that are specific to a domain. 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.

Most people are generally reluctant to discuss political and organizational issues that may affect the requirements.

Interviewing

- Effective interviewers have two characteristics:

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

2. They prompt the interviewee to get discussions going using a springboard question, a requirements proposal, or by working together on a prototype system. They 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 rather than abstract descriptions.
- They can understand and criticize 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.
- Scenarios can be particularly useful for adding detail to an outline requirements description.
- They are descriptions of example interaction sessions.
- Each scenario usually covers one or a small number of possible interactions. Different forms of scenarios are developed and they provide different types of information at different levels of detail about the system

Scenarios

- A scenario starts with an outline of the interaction.
- During the elicitation process, details are added to this to create a complete description of that interaction.
- At its most general, a scenario may include:

1. A description of what the system and users expects 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.

INITIAL ASSUMPTION:
The patient has seen a medical receptionist who has created a record in the system and collected the patient's personal information (name, address, age, etc.). A nurse is logged on to the system and is collecting medical history.

NORMAL:
The nurse searches for the patient by family name. If there is more than one patient with the same surname, the given name (first name in English) and date of birth are used to identify the patient.

The nurse chooses the menu option to add medical history.

The nurse then follows a series of prompts from the system to enter information about consultations elsewhere on mental health problems (free text input), existing medical conditions (nurse selects conditions from menu), medication currently taken (selected from menu), allergies (free text), and home life (form).

WHAT CAN GO WRONG:
The patient's record does not exist or cannot be found. The nurse should create a new record and record personal information.

Patient conditions or medication are not entered in the menu. The nurse should choose the 'other' option and enter free text describing the condition/medication.

Patient cannot/will not provide information on medical history. The nurse should enter free text recording the patient's inability/unwillingness to provide information. The system should print the standard exclusion form stating that the lack of information may mean that treatment will be limited or delayed. This should be signed and handed to the patient.

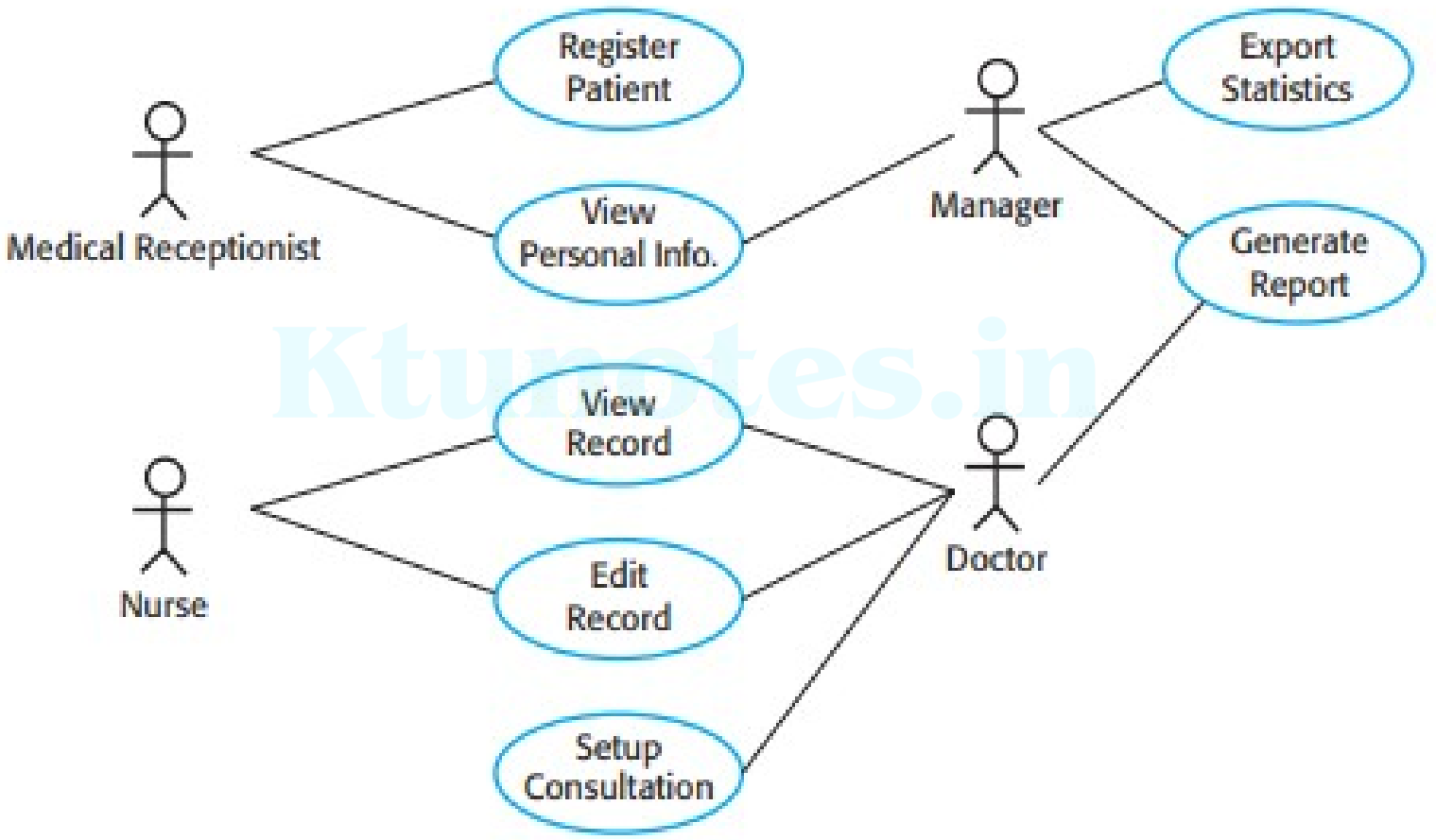
OTHER ACTIVITIES:
Record may be consulted but not edited by other staff while information is being entered.

SYSTEM STATE ON COMPLETION:
User is logged on. The patient record including medical history is entered in the database, a record is added to the system log showing the start and end[illegible]e involved.

- Scenario-based elicitation involves working with stakeholders to identify scenarios and to capture details to be included in these scenarios.
- Scenarios may be written as text, supplemented by diagrams, screen shots, etc.
- Alternatively, a more structured approach such as event scenarios or use cases may be used

Use cases

- a fundamental feature of the unified modeling language.
- a use case identifies the actors involved in an interaction and names the type of interaction.
- This is then supplemented by additional information describing the interaction with the system. The additional information may be a textual description or one or more graphical models such as UML sequence or state charts.
- Use cases are documented using a high-level use case diagram.
- The set of use cases represents all of the possible interactions that will be described in the system requirements.
- Actors in the process, who may be human or other systems, are represented as stick figures.
- Each class of interaction is represented as a named ellipse. Lines link the actors with the interaction.
- arrowheads may be added to lines to show how the interaction is initiated

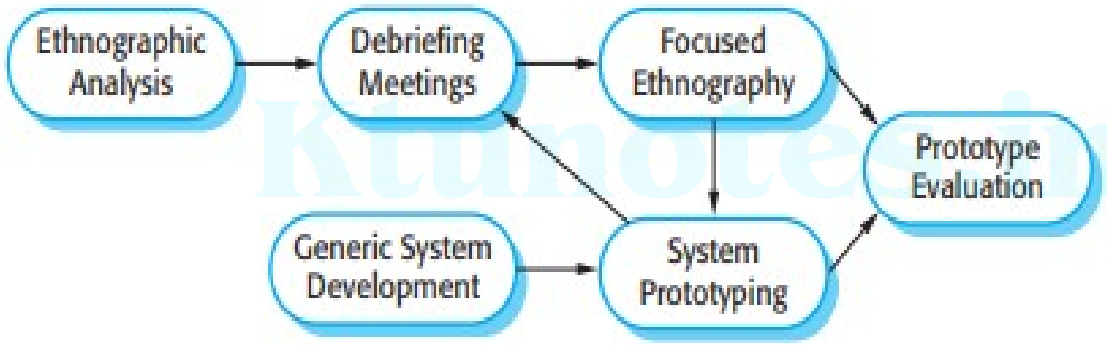


cases

5

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.
- Ethnography is an observational technique that can be used to understand operational processes and help derive support requirements for these processes.
- An analyst immerses himself or herself in the working environment where the system will be used.
- The day-to-day work is observed and notes made of the actual tasks in which participants are involved.
- The value of ethnography is that it helps discover implicit system requirements that reflect the actual ways that people work, rather than the formal processes defined by the organization



- Ethnography is particularly effective for 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.

2. Requirements that are derived from cooperation and awareness of other people's activities

Requirements validation

- Requirements validation is the process of checking that requirements actually define the system that the customer really wants.
- It overlaps with analysis as 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 these problems are discovered during development or after the system is in service.

Requirements validation

During the requirements validation process, **different types of checks** should be carried out on the requirements in the requirements document.

1. Validity checks:

These checks that the requirements reflects the real needs of system users.

2. Consistency checks :

Requirements in the document should not conflict.

3. Completeness checks :

The requirements document should include requirements that define all functions and the constraints intended by the system user.

4. Realism checks:

Using knowledge of existing technology, the requirements should be checked to ensure that they can actually be implemented.

-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.

Requirements validation

- There are a number of **requirements validation techniques** that can be used individually or in conjunction with one another:

1. Requirements reviews :

The requirements are analyzed systematically by a team of reviewers who check for errors and inconsistencies.

2. Prototyping :

In this approach to validation, an executable model of the system in question 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.

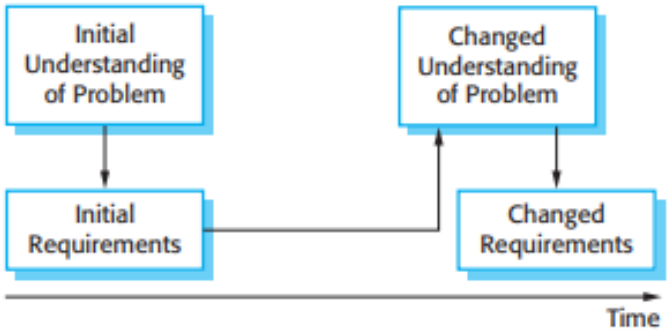
Requirements change

- The requirements for large software systems are always changing.
- One reason for this is that these systems are usually developed to address ‘wicked’ problems—problems that cannot be completely defined.
- Because the problem cannot be fully defined, the software requirements are bound to be incomplete.

Requirements change

- During the software process, the stakeholders' understanding of the problem is constantly changing.
- The system requirements must then also evolve to reflect this changed problem view

- Figure: Requirements evolution



Requirements change

- Most changes to system requirements arise because of changes to the business environment of the system:

1. The business and technical environment of the system always changes after installation. New hardware may be introduced and existing hardware updated.

2. The people who pay for a system and the users of that system are rarely the same people

3. Large systems usually have a diverse stakeholder community, with stakeholders having different requirements. Their priorities may be conflicting or contradictory.

Requirements change

- As requirements are evolving, you need to keep track of individual requirements and maintain links between dependent requirements so that you can assess the impact of requirements changes.
- You therefore need a formal process for making change proposals and linking these to system requirements.
- This process of “requirements management” should start as soon as a draft version of the requirements document is available

Requirements change

- Requirements management planning
- Requirements change management

Requirements management planning

Requirements management planning is concerned with establishing how a set of evolving requirements will be managed.

During the planning stage, you have to decide on a number of issues:

1. Requirements identification:

Each requirement must be uniquely identified so that it can be cross-referenced with other requirements and used in traceability assessments.

2. A change management process:

This is the set of activities that assess the impact and cost of changes.

3. Traceability policies:

These policies define the relationships between each requirement and between the requirements and the system design that should be recorded. The traceability policy should also define how these records should be maintained.

4. 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 shared spreadsheets and simple database systems.

Requirements management planning

- Requirements management needs automated support, and the software tools for this should be chosen during the planning phase. Tool support is needed for:

1. 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:

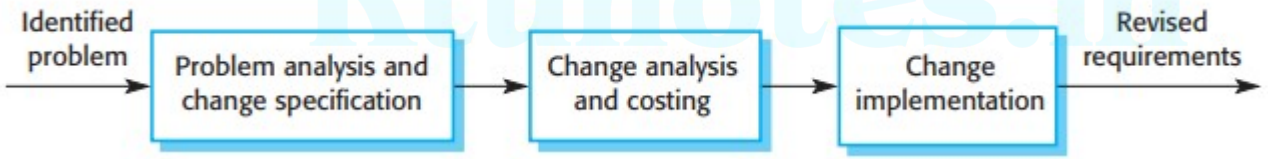
The process of change management is simplified if active tool support is available. Tools can keep track of suggested changes and responses to these suggestions.

3. Traceability management:

Tool support for traceability allows related requirements to be discovered.

Requirements change management

- Requirements change management should be applied to all proposed changes to a system's requirements after the requirements document has been approved.



Requirements change management

- There are three principal stages to a change management process:

1. Problem analysis and change specification:

During this stage, the problem or the change proposal is analyzed to check that it is valid. [During this stage, the problem or the change proposal is analyzed to check that it is valid. This analysis is fed back to the change requestor who may respond with a more specific requirements change proposal, or decide to withdraw the request]

2. Change analysis and costing:

The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements. The cost of making the change is estimated in terms of modifications to the requirements document and, if appropriate, to the system design and implementation. Once this analysis is completed, a decision is made as to whether or not to proceed with the requirements change.

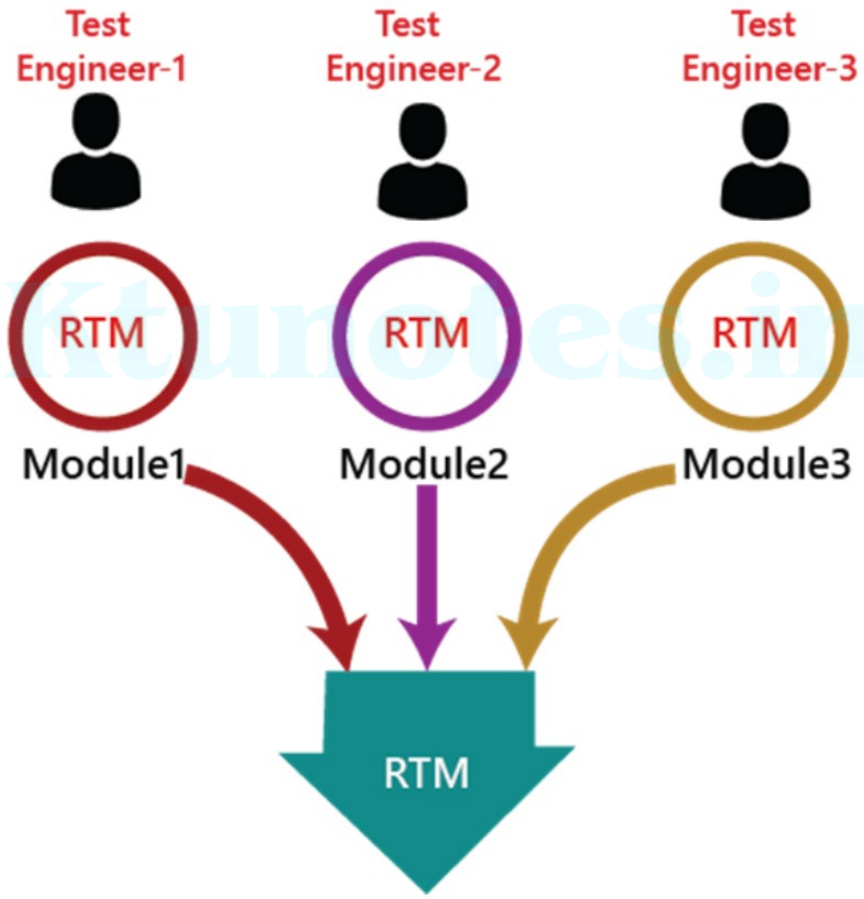
3. Change implementation :

The requirements document and, where necessary, the system design and implementation, are modified.[You should organize the requirements document so that you can make changes to it without extensive rewriting or reorganization. As with programs, changeability in documents is achieved by minimizing external references and making the document sections as modular as possible. Thus, individual sections can be changed and replaced without affecting other parts of the document]

Traceability Matrix

- Traceability matrix is a table type document that is used in the development of software application to trace requirements.
- It can be used for both forward (from Requirements to Design or Coding) and backward (from Coding to Requirements) tracing.
- It is also known as **Requirement Traceability Matrix (RTM) or Cross Reference Matrix (CRM).**
- It is prepared before the test execution process to make sure that every requirement is covered in the form of a Test case so that we don't miss out any testing.
- In the RTM document, we map all the requirements and corresponding test cases to ensure that we have written all the test cases for each condition.
- **The test engineer** will prepare RTM for their respective assign modules, and then it will be sent to the Test Lead. The Test Lead will go repository to check whether the Test Case is there or not and finally Test Lead consolidate and prepare one necessary RTM document.

Traceability Matrix



Traceability Matrix

- This document is designed to make sure that each requirement has a test case, and the test case is written based on business needs, which are given by the client.
- . The traceability is written to make sure that the entire requirement is covered.

Traceability Matrix

RTM Template

Below is the sample template of requirement traceability matrix (RTM):

Requirement no	Module name	High level requirement	Low level requirement	Test case name

Traceability Matrix

	A	B	C	D	E
1	RTM Template				
2	**Requirement number**	**Module number**	**High level requirement**	**Low level requirement**	**Test case name**
3	2	Loan	**2.1 Personal loan**	2.1.1--> personal loan for private employee	beta-2.0-personal loan
4				2.1.2--> personal loan for government employee	
5				2.1.3--> personal loan for jobless people	
6			**2.2 Car loan**	2.2.1--> car loan for private employee	
7				——	
8			**2.3 Home loan**	——	
9				——	
10				——	
11					

Traceability Matrix

Goals of Traceability Matrix

- It helps in tracing the documents that are developed during various phases of SDLC.
- It ensures that the software completely meets the customer's requirements.
- It helps in detecting the root cause of any bug.

Traceability Matrix

Types of Traceability Test Matrix

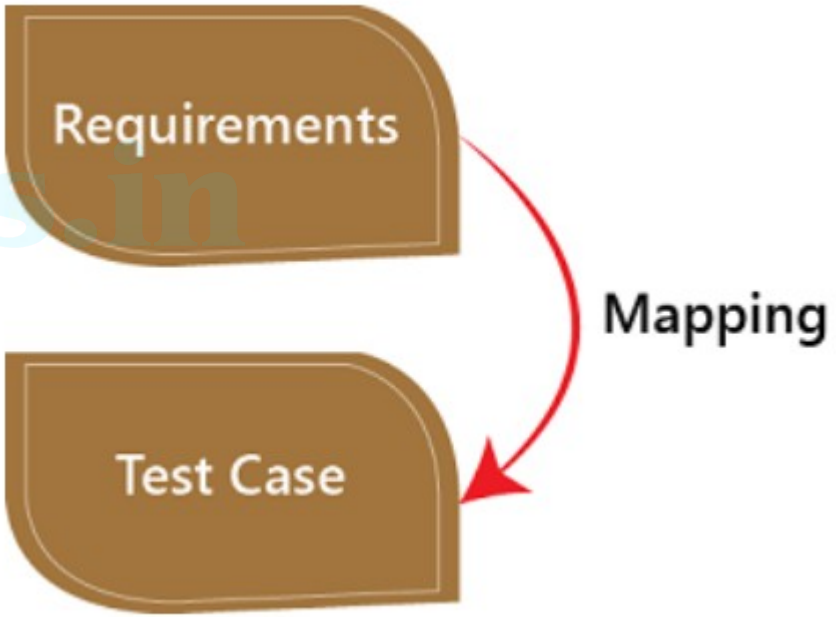
The traceability matrix can be classified into three different types which are as follows:

- Forward traceability
- Backward or reverse traceability
- Bi-directional traceability

Traceability Matrix

Forward traceability

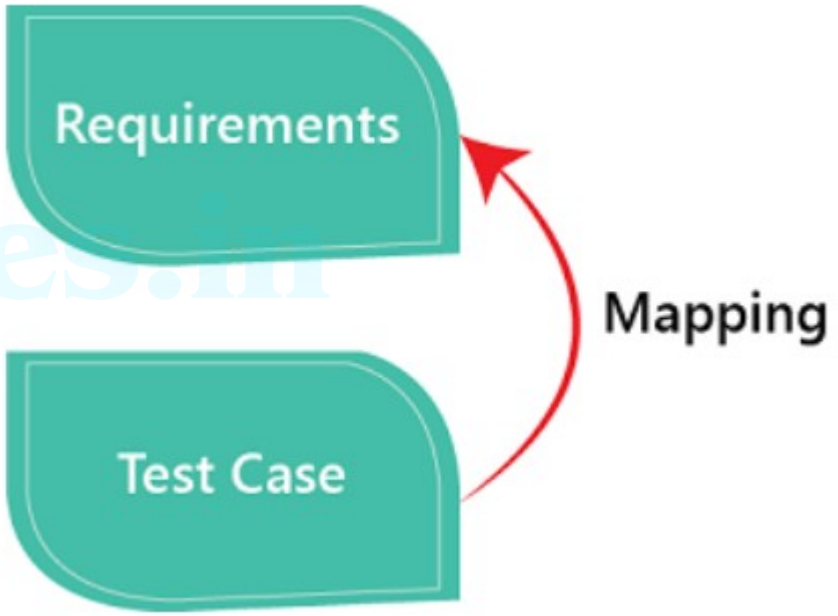
- The forward traceability test matrix is used to ensure that every business's needs or requirements are executed correctly in the application and also tested rigorously.
- The main objective of this is to verify whether the product developments are going in the right direction.
- In this, the requirements are mapped into the forward direction to the test cases.



Traceability Matrix

Backward or reverse traceability

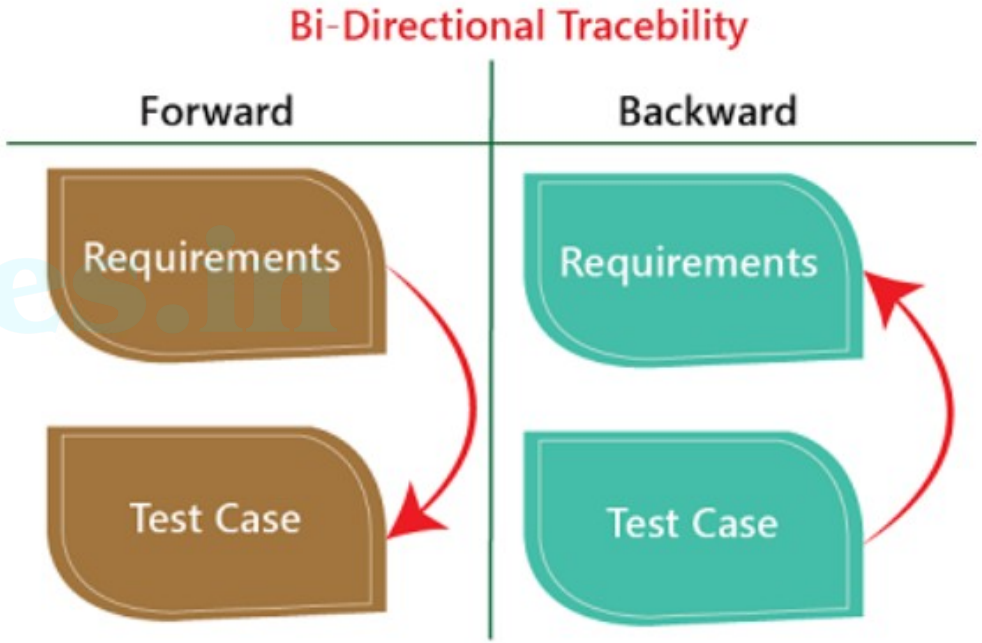
- The reverse or backward traceability is used to check that we are not increasing the space of the product by enhancing the design elements, code, test other things which are not mentioned in the business needs.
- And the main objective of this that the existing project remains in the correct direction.
- In this, the requirements are mapped into the backward direction to the test cases.



Traceability Matrix

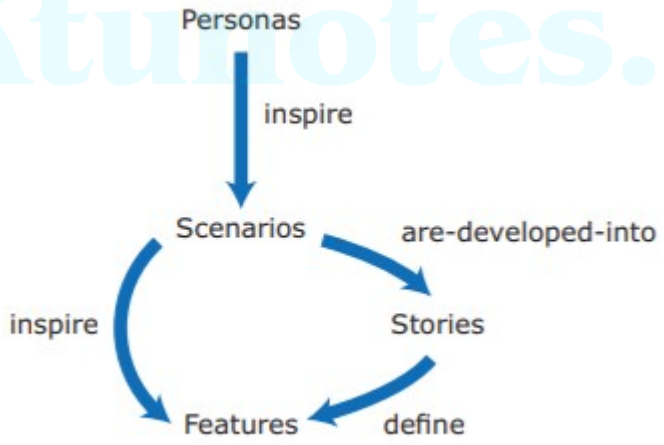
Bi-directional traceability

- It is a combination of forwarding and backward traceability matrix, which is used to make sure that all the business needs are executed in the test cases.
- It also evaluates the modification in the requirement which is occurring due to the bugs in the application.



Personas, Scenarios, User stories, Feature identification

- personas, scenarios, and user stories lead to features that might be implemented in a software product.



- a feature is a fragment of functionality that implements some user or system need.
- A feature is something that the user needs or wants.
- A standard template can be used where you define the feature by its input, its functionality, its output, and how it is activated.

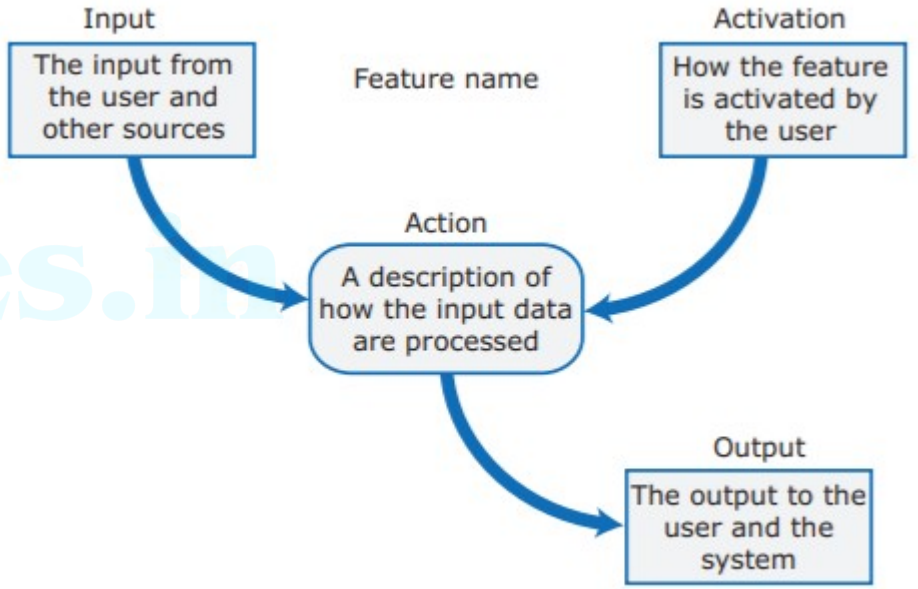
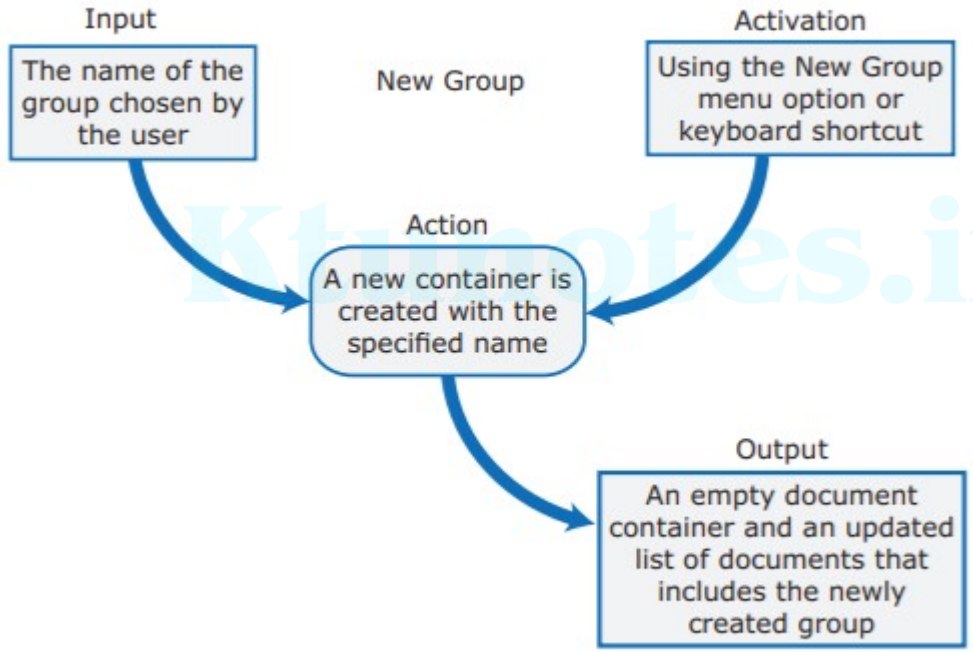


Figure: Feature description

The New Group feature description



Personas

- Personas are about “imagined users,” character portraits of types of user that you think might adopt your product.
- Example: if your product is aimed at managing appointments for dentists, you might create a dentist persona, a receptionist persona, and a patient persona.
- Personas of different types of users help you imagine what these users may want to do with your software and how they might use it.
- They also help you envisage difficulties that users might have in understanding and using product features.
- A persona
  - should paint a picture of a type of product user.
  - should describe the users’ backgrounds and why they might want to use your product.
  - should also say something about their education and technical skills.

- There is no standard way of representing a persona

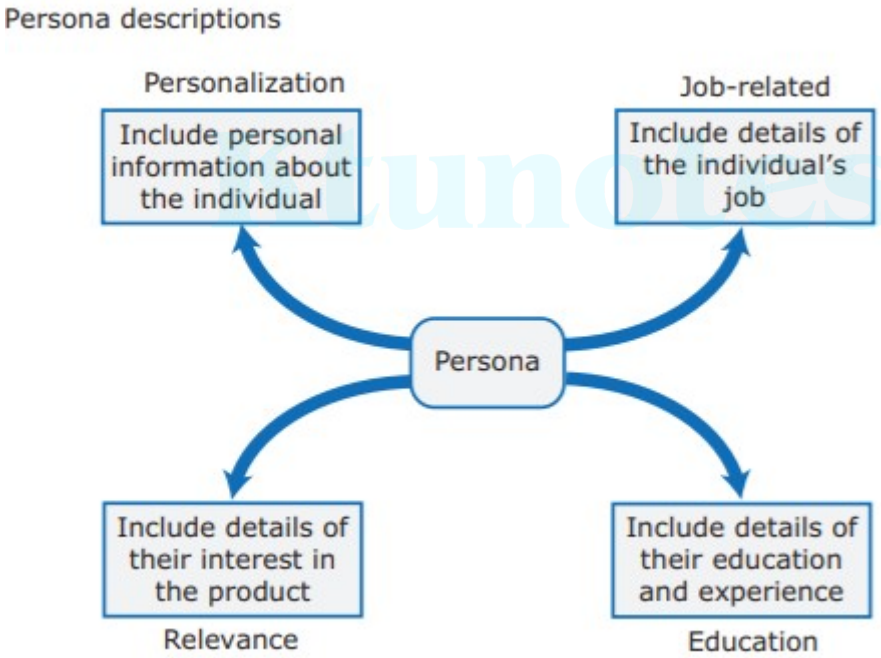


Table 3.2 Aspects of persona descriptions

Aspect	Description
Personalization	You should give them a name and say something about their personal circumstances. It is sometimes helpful to use an appropriate stock photograph to represent the person in the persona. Some studies suggest that this helps project teams use personas more effectively.
Job-related	If your product is targeted at business, you should say something about their job and (if necessary) what that job involves. For some jobs, such as a teacher where readers are likely to be familiar with the job, this may not be necessary.
Education	You should describe their educational background and their level of technical skills and experience. This is important, especially for interface design.
Relevance	If you can, you should say why they might be interested in using the product and what they might want to do with it.

- In general, you don’t need more than five personas to help identify the key features of a system.
- Personas should be relatively short and easy to read.

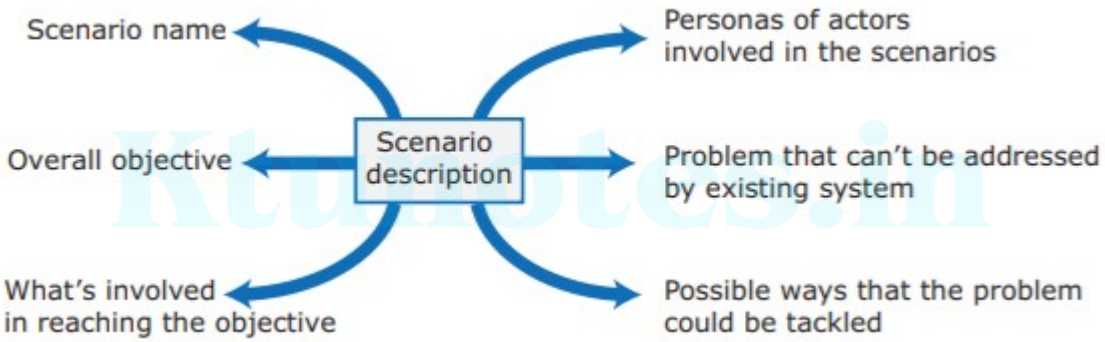
Table 3.3 A persona for a history teacher

Emma, a history teacher
Emma, age 41, is a history teacher in a secondary school (high school) in Edinburgh. She teaches students from ages 12 to 18. She was born in Cardiff in Wales, where both her father and her mother were teachers. After completing a degree in history from Newcastle University, she moved to Edinburgh to be with her partner and trained as a teacher. She has two children, aged 6 and 8, who both attend the local primary school. She likes to get home as early as she can to see her children, so often does lesson preparation, administration, and marking from home. Emma uses social media and the usual productivity applications to prepare her lessons, but is not particularly interested in digital technologies. She hates the virtual learning environment that is currently used in her school and avoids using it if she can. She believes that face-to-face teaching is most effective. She might use the iLearn system for administration and access to historical films and documents. However, she is not interested in a blended digital/face-to-face approach to teaching.

scenario

- A scenario is a narrative that describes a situation in which a user is using your product's features to do something that they want to do.
- The scenario should briefly explain the user's problem and present an imagined way that the problem might be solved.
- There is no need to include everything in the scenario; it isn't a detailed system specification.
- The most important elements of a scenario are:
  1. A brief statement of the overall objective.
  2. References to the persona involved so that you can get information about the capabilities and motivation of that user.
  3. Information about what is involved in doing the activity.
  4. If appropriate, an explanation of problems that can't be readily addressed using the existing system.
  5. A description of one way that the identified problem might be addressed

scenario



scenario

- Scenarios are effective in communication because they are understandable and accessible to users and to people responsible for funding and buying the system.
- Like personas, they help developers to gain a shared understanding of the system that they are creating.
- scenarios are not specifications. They lack detail, they may be incomplete, and they may not represent all types of user interactions.
- Scenarios are descriptions of situations in which a user is trying to do something with a software system.

scenario

Writing scenarios

- Starting point for scenario writing should be the personas you have created.
- Try to imagine several scenarios for each persona.
- Scenarios should always be written from the user’s perspective and should be based on identified personas or real users.
- Look at the roles of each of the personas that you have developed and write scenarios that cover the main responsibilities for that role.

User stories

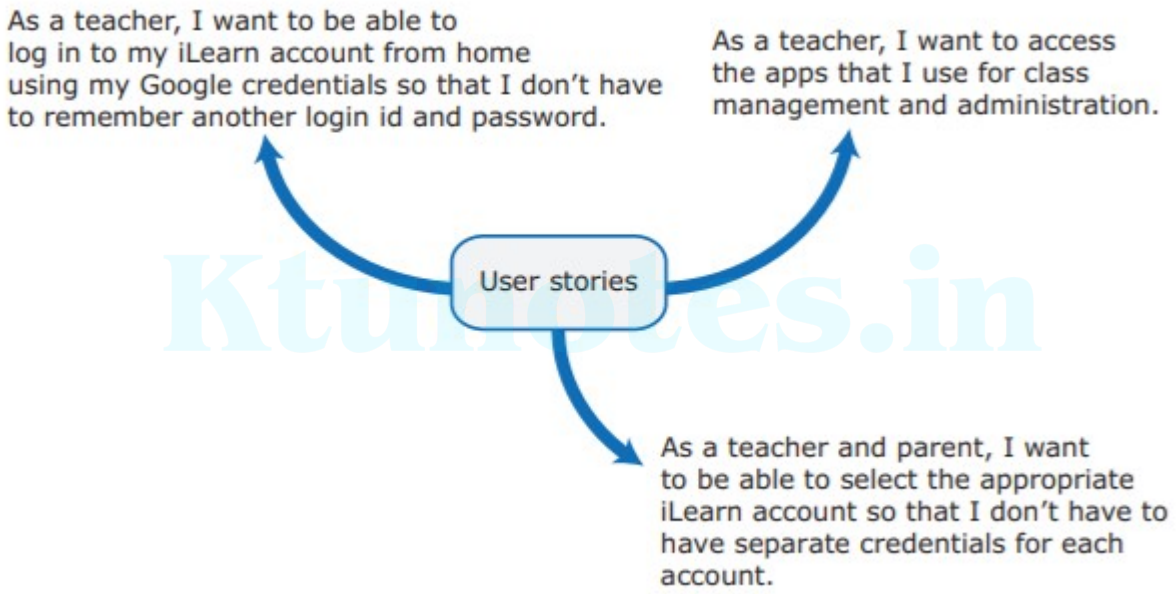
- User stories are finer-grain narratives that set out in a more detailed and structured way a single thing that a user wants from a software system.

As an author I need a way to organize the book that I'm writing into chapters and sections.

This story reflects what has become the standard format of a user story:

As a <role>, I <want / need> to <do something>

User stories



User stories

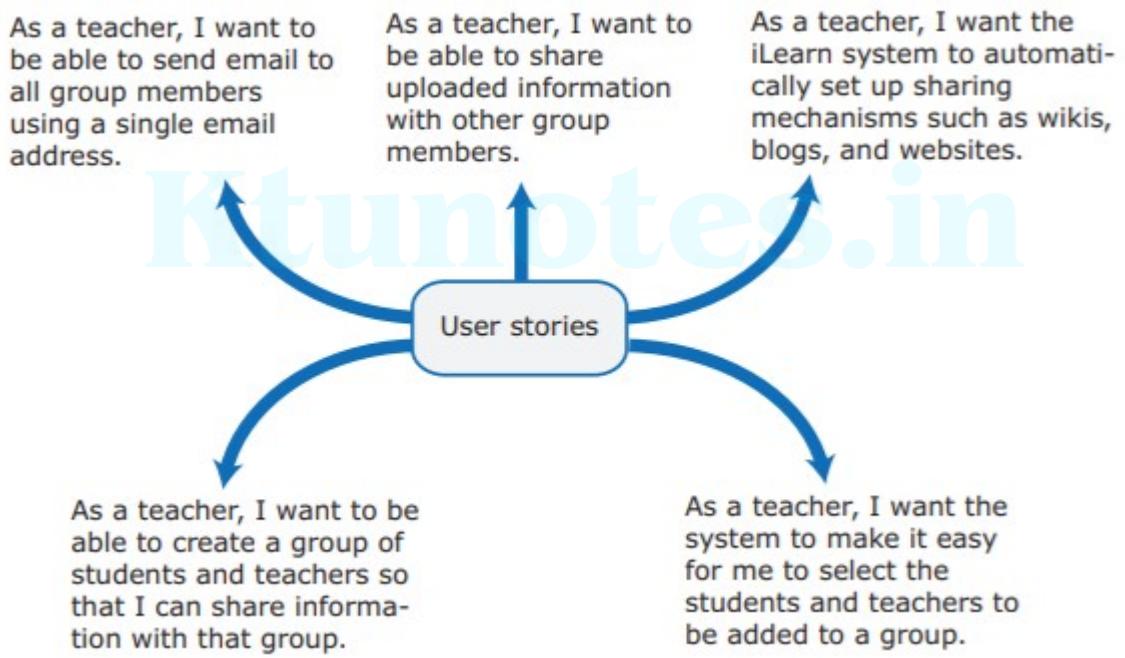
- An important use of user stories is in planning, and many users of the Scrum method represent the product backlog as a set of user stories.
- For this purpose, user stories should focus on a clearly defined system feature or aspect of a feature that can be implemented within a single sprint.
- If the story is about a more complex feature that might take several sprints to implement, then it is called an “epic.”

User stories

As a system manager, I need a way to back up the system and restore individual applications, files, directories, or the whole system.

- A lot of functionality is associated with this user story. For implementation, it should be broken down into simpler stories, with each story focusing on a single aspect of the backup system.
- develop stories that are helpful in one of two ways:
  - as a way of extending and adding detail to a scenario;
  - as part of the description of the system feature that you have identified.

User stories



scenarios are more natural and are helpful for the following reasons:

- Scenarios read more naturally because they describe what a user of a system is actually doing with that system. People often find it easier to relate to this specific information rather than to the statement of wants or needs set out in a set of user stories.
- When you are interviewing real users or checking a scenario with real users, they don’t talk in the stylized way that is used in user stories. People relate better to the more natural narrative in scenarios.
- Scenarios often provide more context—information about what users are trying to do and their normal ways of working

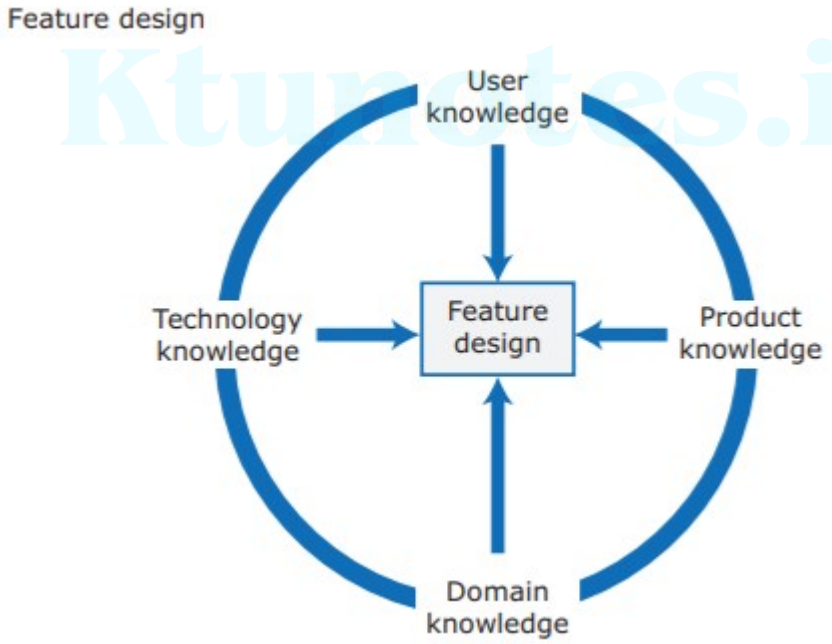
Feature identification

- A feature is a way of allowing users to access and use your product's functionality so that the feature list defines the overall functionality of the system.
- Identify product features that are independent, coherent and relevant:

1. Independence :A feature should not depend on how other system features are implemented and should not be affected by the order of activation of other features.
2. Coherence: Features should be linked to a single item of functionality. They should not do more than one thing, and they should never have side effects.
3. Relevance: System features should reflect the way users normally carry out some task. They should not offer obscure functionality that is rarely required

Feature identification

- There is no definitive method for feature selection and design.
- Rather, the four important knowledge sources shown in Figure can help with this.

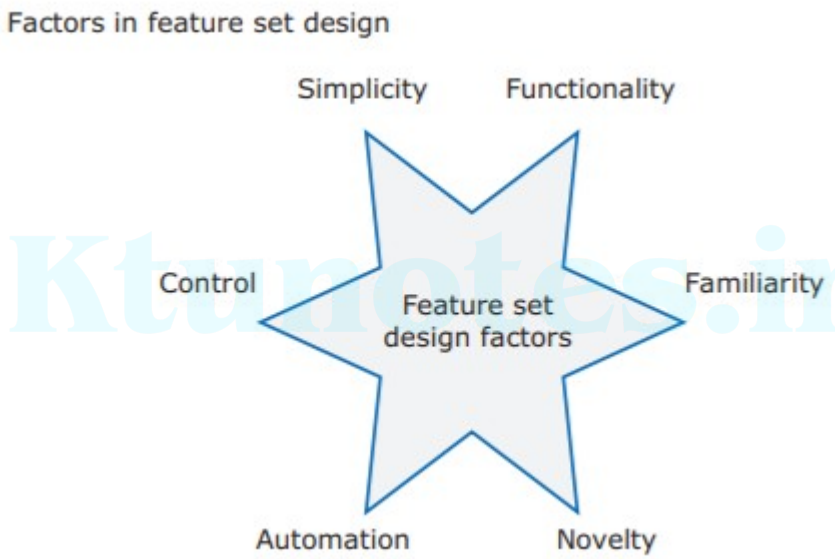


Feature identification

Table 3.8 Knowledge required for feature design

Knowledge	Description
User knowledge	You can use user scenarios and user stories to inform the team of what users want and how they might use the software features.
Product knowledge	You may have experience of existing products or decide to research what these products do as part of your development process. Sometimes your features have to replicate existing features in these products because they provide fundamental functionality that is always required.
Domain knowledge	This is knowledge of the domain or work area (e.g., finance, event booking) that your product aims to support. By understanding the domain, you can think of new innovative ways of helping users do what they want to do.
Technology knowledge	New products often emerge to take advantage of technological developments since their competitors were launched. If you understand the latest technology, you can design features to make use of it.

Feature identification



Feature creep

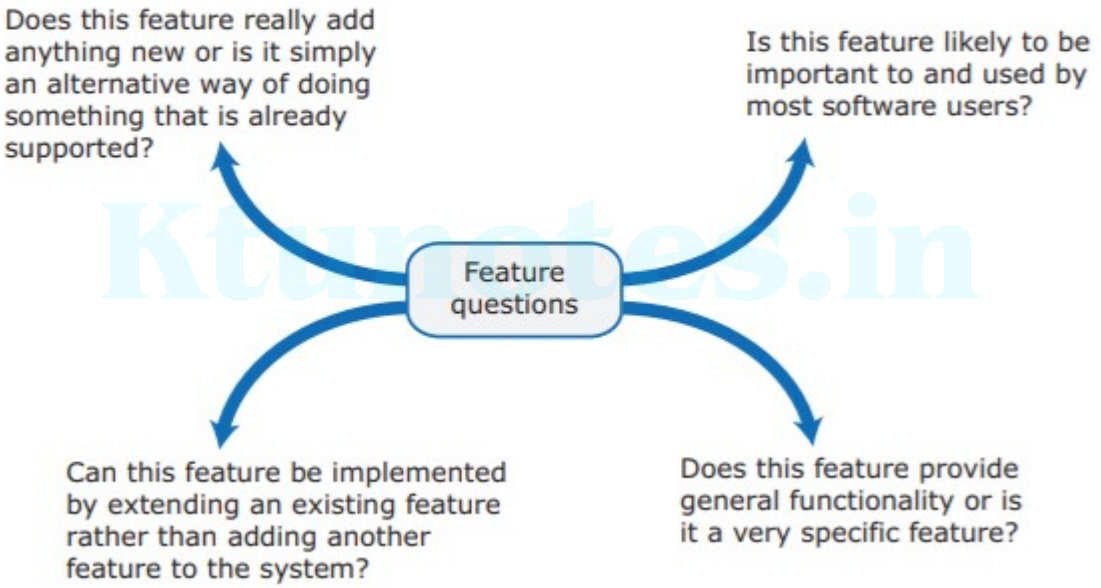
- Feature creep means that the number of features in a product creeps up as new potential uses of the product are envisaged.
- The size and complexity of many large software products such as Microsoft Office and Adobe Photoshop are a consequence of feature creep.
- Most users use only a relatively small subset of the features of these products.
- Rather than stepping back and simplifying things, developers continually added new features to the software.

Feature creep

- Feature creep happens for three reasons:
- 1. Product managers and marketing executives discuss the functionality they need with a range of different product users.
- 2. Competitive products are introduced with slightly different functionality to your product.
- 3. The product tries to support both experienced and inexperienced users.

Feature creep

Figure 3.10 Avoiding feature creep



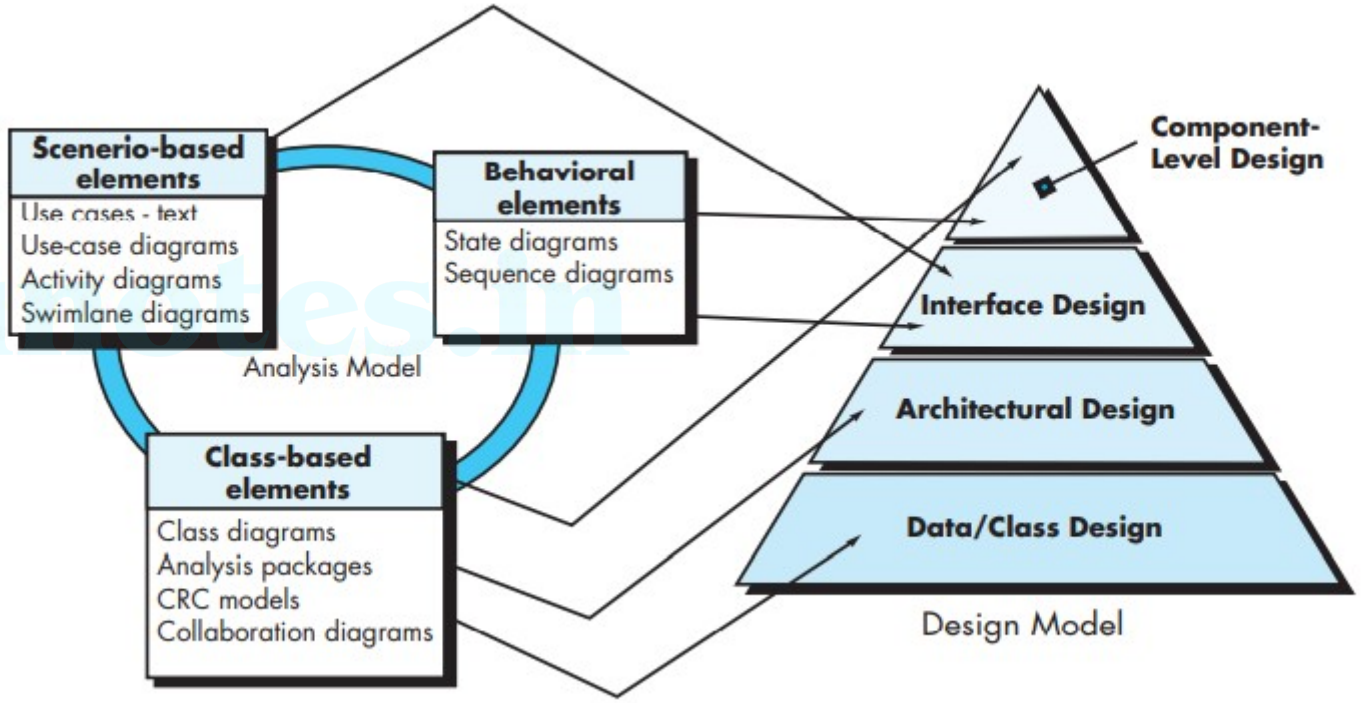
- Simplicity and functionality :
- Everyone says they want software to be as simple as possible to use. At the same time, they demand functionality that helps them do what they want to do. You need to find a balance between providing a simple, easy-to-use system and including enough functionality to attract users with a variety of needs.
- Familiarity and novelty :Users prefer that new software should support the familiar everyday tasks that are part of their work or life. However, if you simply replicate the features of a product that they already use, there is no real motivation for them to change. To encourage users to adopt your system, you need to include new features that will convince users that your product can do more than its competitors.
- 3. Automation and control: You may decide that your product can automatically do things for users that other products can't. However, users inevitably do things differently from one another. Some may like automation, where the software does things for them. Others prefer to have control. You therefore have to think carefully about what can be automated, how it is automated, and how users can configure the automation so that the system can be tailored to their preferences.

Design Concepts

- Design creates a representation or model of the software.
- Design model provides detail about software architecture, data structures, interfaces, and components that are necessary to implement the system.
- The goal of design is to produce a model or representation that exhibits firmness, commodity, and delight.
- Software design is an iterative process through which requirements are translated into a “blueprint” for constructing the software.
- design produces a data/class design, an architectural design, an interface design, and a component design

DESIGN WITHIN THE CONTEXT OF SOFTWARE E NGINEERING

- Software design sits at the technical kernel of software engineering and is applied regardless of the software process model that is used.
- Beginning once software requirements have been analyzed and modeled, software design is the last software engineering action within the modeling activity and sets the stage for construction (code generation and testing).
- The flow of information during software design is illustrated in the figure.



- The data/class design transforms class models into design class realizations and the requisite data structures required to implement the software.
- The architectural design defines the relationship between major structural elements of the software, the architectural styles and patterns.
- The interface design describes how the software communicates with systems that interoperate with it, and with humans who use it.
- The component-level design transforms structural elements of the software architecture into a procedural description of software components.

- The importance of software design can be stated with a single word—quality.
- Design is the place where quality is fostered in software engineering.

THE DESIGN PROCESS

- Software design is an iterative process through which requirements are translated into a “blueprint” for constructing the software.
  - Software Quality Guidelines and Attributes.
  - The Evolution of Software Design

Software Quality Guidelines and Attributes.

- the quality of the evolving design is assessed with a series of technical reviews.
- Three characteristics that serve as a guide for the evaluation of a good design :
  - ❑The design should implement all of the explicit requirements contained in the requirements model, and it must accommodate all of the implicit requirements desired by stakeholders.
  - ❑The design should be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
  - ❑The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

- Quality Guidelines
  1. A design should exhibit an architecture that (1) has been created using recognizable architectural styles or patterns, (2) is composed of components that exhibit good design characteristics, and (3) can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.
  2. A design should be modular; that is, the software should be logically partitioned into elements or subsystems.
  3. A design should contain distinct representations of data, architecture, interfaces, and components.
  4. A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
  5. A design should lead to components that exhibit independent functional characteristics.
  6. A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
  7. . A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
  8. A design should be represented using a notation that effectively communicates its meaning.

Quality Attributes

- FURPS—functionality, usability, reliability, performance, and supportability.
  - Functionality is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.
  - Usability is assessed by considering human factors ,overall aesthetics, consistency, and documentation.
  - Reliability is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure, and the predictability of the program.
  - Performance is measured using processing speed, response time, resource consumption, throughput, and efficiency.
  - Supportability combines extensibility, adaptability, and serviceability. These three attributes represent a more common term, maintainability— and in addition, testability, compatibility, configurability ,the ease with which a system can be installed, and the ease with which problems can be localized.

The Evolution of Software Design

- Early design work concentrated on criteria for the development of modular programs and methods for refining software structures in a top-down “structured” .
- Newer design approaches proposed an object-oriented approach to design derivation.
- More recent emphasis in software design has been on software architecture and the design patterns that can be used to implement software architectures and lower levels of design abstractions
- Growing emphasis on aspect-oriented methods ,model-driven development ,and test-driven development .

DESIGN CONCEPTS

Overview of fundamental software design concepts that provide the necessary framework for “getting it right.”

1. Abstraction
2. Architecture
3. Patterns
4. Separation of Concerns
5. Modularity
6. Information Hiding
7. Functional Independence
8. Refinement
9. Aspects
10. Refactoring
11. Object-Oriented Design Concepts
12. Design Classes
13. Dependency Inversion
14. Design for Tes

Abstraction

- many levels of abstraction can be posed.
- At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment.
- At lower levels of abstraction, a more detailed description of the solution is provided.
- A **procedural abstraction** refers to a sequence of instructions that have a specific and limited function.
- A **data abstraction** is a named collection of data that describes a data object.

Architecture

- architecture is the structure or organization of program components (modules), the manner in which these components interact, and the , structure of data that are used by the components.
- Components can be generalized to represent major system elements and their interactions.
- set of properties that should be specified as part of an architectural design.
  - Structural properties define “the components of a system (e.g., modules, objects, fi lters) and the manner in which those components are packaged and interact with one another.”
  - Extra-functional properties address “how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.
- Architectural design can be represented using one or more of a number of different models
  - Structural models represent architecture as an organized collection of program components.
  - Framework models increase the level of design abstraction by attempting to identify repeatable architectural design frameworks (patterns) that are encountered in similar types of applications.
  - Dynamic models address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events.
  - Process models focus on the design of the business or technical process that the system must accommodate.
  - functional models can be used to represent the functional hierarchy of a system.

Patterns

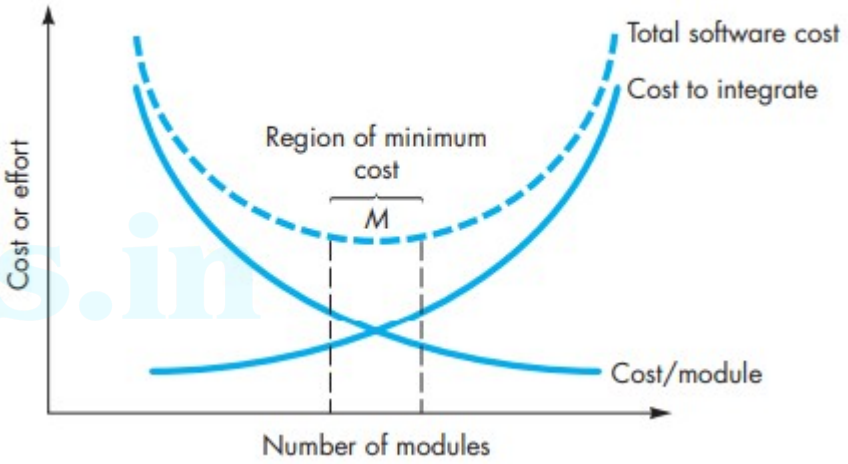
- a design pattern describes a design structure that solves a particular design problem within a specific context and amid “forces” that may have an impact on the manner in which the pattern is applied and used.
- The intent of each design pattern is to provide a description that enables a designer to determine
  - whether the pattern is applicable to the current work,
  - whether the pattern can be reused (hence, saving design time), and
  - whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

Separation of concerns

- Separation of concerns is a design concept that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently
- A concern is a feature or behavior that is specified as part of the requirements model for the software.
- By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.
- divide-and-conquer strategy—it's easier to solve a complex problem when you break it into manageable pieces.

Modularity

- Modularity is the most common manifestation of separation of concerns.
- Software is divided into separately named and addressable components, sometimes called modules, that are integrated to satisfy problem requirements.
- As the number of modules grows, the effort (cost) associated with integrating the modules also grows. These characteristics lead to a total cost or effort curve shown in the figure. There is a number, M, of modules that would result in minimum development cost.



Information Hiding

- The principle of information hiding suggests that modules be “characterized by design decisions that (each) hides from all others.”
- Modules should be specified and designed so that information (algorithms and data) contained within a module is inaccessible to other modules that have no need for such information.
- The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later during software maintenance - , inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

Functional Independence

- The concept of functional independence is a direct outgrowth of separation of concerns, modularity, and the concepts of abstraction and information hiding.
- Design software so that each module addresses a specific subset of requirements and has a simple interface when viewed from other parts of the program structure.
- Independence is assessed using two qualitative criteria: cohesion and coupling.
  - Cohesion is an indication of the relative functional strength of a module.
  - Coupling is an indication of the relative interdependence among modules
- functional independence is a key to good design, and design is the key to software quality.

- .

Refinement

- An application is developed by successively refining levels of procedural detail.
- Refinement is a process of elaboration.
- Begin with a statement of function (or description of information) that is defined at a high level of abstraction. This statement describes function or information conceptually but provides no indication of the internal workings of the function.
- Elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.
- Abstraction and refi nement are complementary concepts.

- As design begins, requirements are refined into a modular design representation.
- Consider two requirements, A and B. Requirement A crosscuts requirement B “if a software decomposition [refinement] has been chosen in which B cannot be satisfied without taking A into account”.
- An aspect is a representation of a crosscutting concern.
- an aspect is implemented as a separate module (component) rather than as software fragments that are “scattered” or “tangled” throughout many components

Refactoring

- Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure.
- When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design.
- [For example, a first design iteration might yield a component that exhibits low cohesion. After careful consideration, you may decide that the component should be refactored into three separate components, each exhibiting high cohesion. The result will be software that is easier to integrate, easier to test, and easier to maintain.]

- Object-Oriented Design Concepts

The object-oriented (OO) paradigm is widely used in modern software engineering.

Design Classes

- The analysis model defines a set of analysis classes- each of these classes describes some element of the problem domain.
- As the design model evolves, you will define a set of design classes that refine the analysis classes by providing design detail that will enable the classes to be implemented, and implement a software infrastructure that supports the business solution.
- Five different types of design classes, each representing a different layer of the design architecture, can be developed.
  - User interface classes define all abstractions that are necessary for human-computer interaction (HCI) .
  - Business domain classes identify the attributes and services (methods) that are required to implement some element of the business domain.
  - Process classes implement lower-level business abstractions required to fully manage the business domain classes.
  - Persistent classes represent data stores (e.g., a database) that will persist beyond the execution of the software.
  - System classes implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.
- Four characteristics of a well formed design class
  - Complete and sufficient
  - Primitiveness
  - High cohesion.
  - Low coupling.

Dependency inversion

- High-level modules (classes) should not depend [directly] upon low-level modules.
- Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

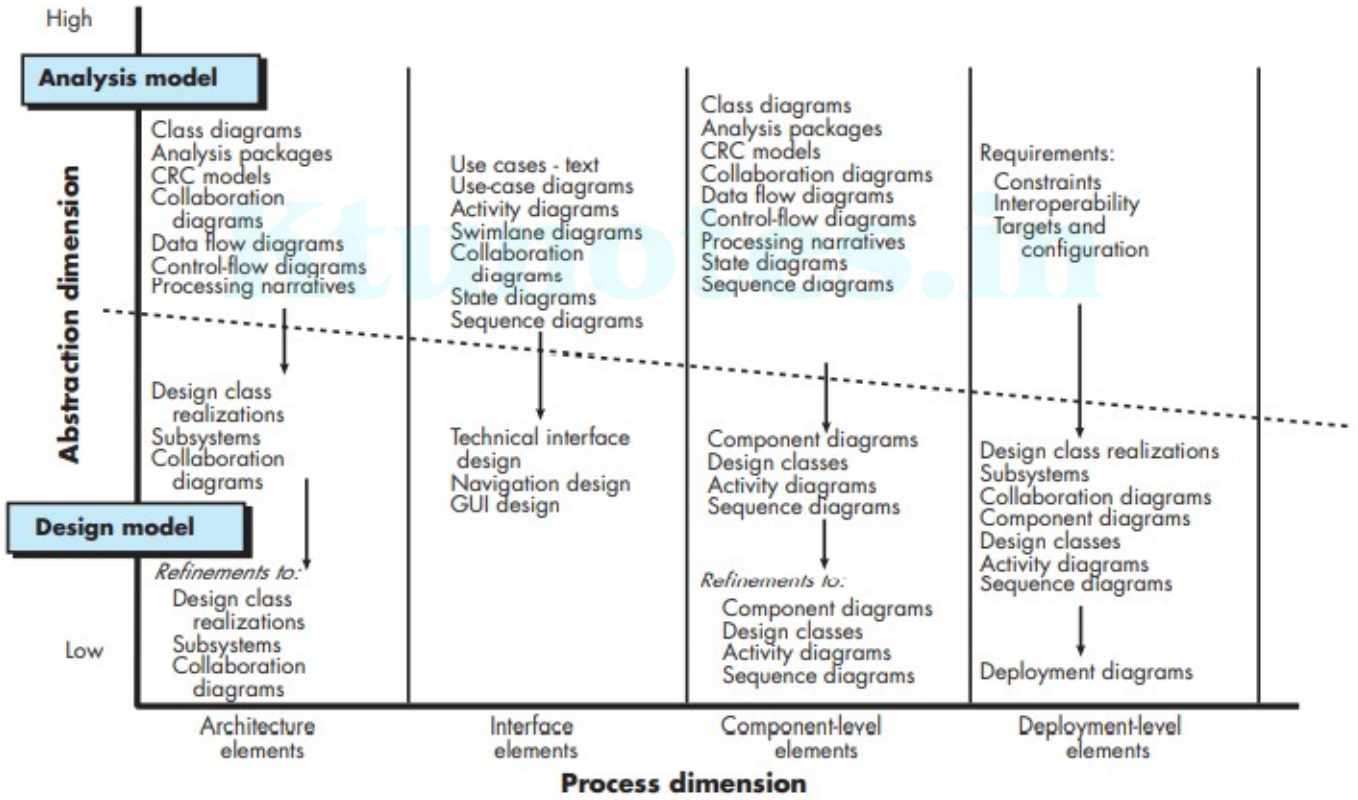
Design for test

- Whether to design and test or test before implementing a code.

THE DESIGN MODEL

- The design model can be viewed in two different dimensions
  - The **process dimension** indicates the evolution of the design model as design tasks are executed as part of the software process.
  - The **abstraction dimension** represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively
    - Data Design Elements
    - Architectural Design Elements
    - Interface Design Elements
    - Deployment-Level Design Elements

FIGURE 12.4 Dimensions of the design model



Data Design Elements

- data design (sometimes referred to as data architecting) creates a model of data and/or information that is represented at a high level of abstraction (the customer/user's view of data).
- This data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system.
- The structure of data has always been an important part of software design.
- At the program-component level, the design of data structures and the associated algorithms required to manipulate them is essential to the creation of high- quality applications.
- At the application level, the translation of a data model (derived as part of requirements engineering) into a database is pivotal to achieving the business objectives of a system.
- At the business level, the collection of information stored in disparate databases and reorganized into a "data warehouse" enables data mining or knowledge discovery that can have an impact on the success of the business itself.

Architectural Design Elements

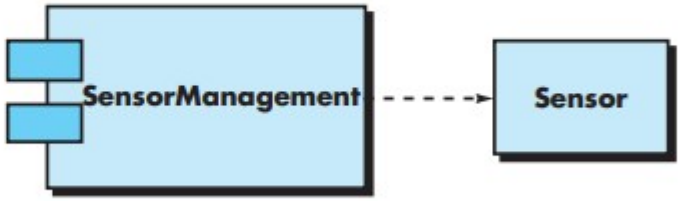
- The architectural design for software is the equivalent to the floor plan of a house.
- The architectural model is derived from three sources:
  - (1) information about the application domain for the software to be built;
  - (2) specific requirements model elements such as use cases or analysis classes, their relationships and collaborations for the problem at hand; and
  - (3) the availability of architectural styles and patterns

The architectural design element is usually depicted as a set of interconnected subsystems

Interface Design Elements

- The interface design for software is analogous to a set of detailed drawings (and specifications) for the doors, windows, and external utilities of a house.
- There are three important elements of interface design:
  - (1) the user interface (UI),
  - (2) external interfaces to other systems, devices, networks, or other producers or consumers of information, and
  - (3) internal interfaces between various design components.
- These interface design elements allow the software to communicate externally and enable internal communication and collaboration among the components that populate the software architecture.

FIGURE 12.6
A UML component diagram



Component-Level Design Elements

- The component-level design for software is the equivalent to a set of detailed drawings (and specifications) for each room in a house.
- The component-level design for software fully describes the internal detail of each software component.
- To accomplish this, the component-level design defines data structures for all local data objects and algorithmic detail for all processing that occurs within a component and an interface that allows access to all component operations (behaviors).

- Deployment-Level Design Elements
- Deployment-level design elements indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software.

Architectural Design

Architectural Design

- Architectural design represents the structure of data and program components that are required to build a computer-based system.
- It considers the architectural style that the system will take, the structure and properties of the components that constitute the system, and the interrelationships that occur among all architectural components of a system.

software architecture

- The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.
- The architecture is not the operational software.
- Rather, it is a representation that enables you to
  (1) analyze the effectiveness of the design in meeting its stated requirements,
  (2) consider architectural alternatives at a stage when making design changes is still relatively easy, and
  (3) reduce the risks associated with the construction of the software.
- a software component can be something as simple as a program module or an object-oriented class, but it can also be extended to include databases and “middleware” that enable the configuration of a network of clients and servers.
- The relationships between components can be as simple as a procedure call from one module to another or as complex as a database access protocol.

software architecture

- There is a distinct difference between the terms architecture and design. A design is an instance of an architecture similar to an object being an instance of a class.
- three key reasons that software architecture is important:
  - Software architecture provides a representation that facilitates communication among all stakeholders.
  - The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows.
  - • Architecture “constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together”
- **Architectural Descriptions**
- **Architectural Decisions**

software architecture

Architectural Descriptions

- architectural description is actually a set of work products that reflect different views of the system.
- An architectural description of a software-based system must exhibit characteristics that combine these metaphors.
- multiple metaphors are there representing different views of the same architecture, that stakeholders use to understand the term software architecture.
  - The blueprint metaphor - Developers regard architecture descriptions as a means of transferring explicit information from architects to designers to software engineers charged with producing the system components.
  - The language metaphor views architecture as a facilitator of communication across stakeholder groups.
  - The decision metaphor represents architecture as the product of decisions involving trade-offs among properties such as cost, usability, maintainability, and performance.
  - The literature metaphor is used to document architectural solutions constructed in the past.
- The IEEE Computer Society has proposed IEEE-Std-1471-2000, Recommended Practice for Architectural Description of Software-Intensive Systems, [IEE00], with the following objectives:
  (1) to establish a conceptual framework and vocabulary for use during the design of software architecture,
  (2) to provide detailed guidelines for representing an architectural description, and
  (3) to encourage sound architectural design practices.

An architectural description (AD) represents multiple views, where each view is “a representation of a whole system from the perspective of a related set of [stakeholder] concerns.”

software architecture

Architectural Decisions

- Each view developed as part of an architectural description addresses a specific stakeholder concern.
- To develop each view (and the architectural description as a whole) the system architect considers a variety of alternatives and ultimately decides on the specific architectural features that best meet the concern.
- Therefore, architectural decisions themselves can be considered to be one view of the architecture.

Architectural style

- An architectural style is a transformation that is imposed on the design of an entire system. The intent is to establish a structure for all components of the system.
- The software that is built for computer-based systems exhibits one of many architectural styles.
- Each style describes a system category that encompasses
  (1) a set of components (e.g., a database, computational modules) that perform a function required by a system,
  (2) a set of connectors that enable “communication, coordination and cooperation” among components,
  (3) constraints that define how components can be integrated to form the system, and
  (4) semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts

Architectural Styles

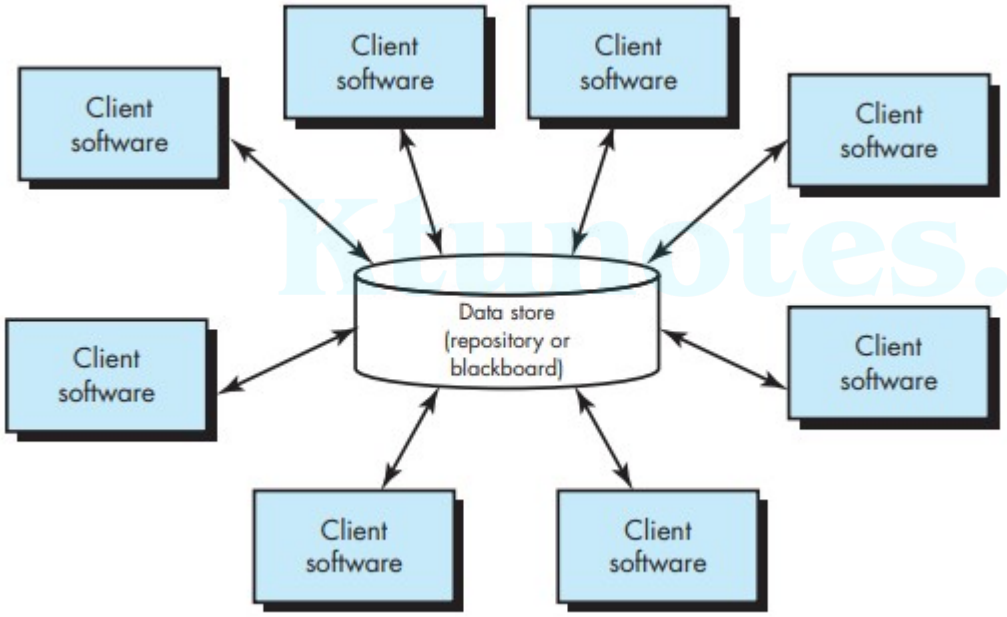
A Brief Taxonomy of Architectural Styles

- computer-based systems can be categorized into one of a relatively small number of architectural styles:
  - **Data-Centered Architectures**
  - **Data-Flow Architectures.**
  - **Call and Return Architectures.**
  - **Object-Oriented Architectures.**
  - **Layered Architectures.**

Data-Centered Architectures

- A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store.
- Client software accesses a central repository.
- In some cases the data repository is passive. That is, client software accesses the data independent of any changes to the data or the actions of other client software.
- A variation on this approach transforms the repository into a “blackboard” that sends notifications to client software when data of interest to the client changes.
- Data-centered architectures promote integrability . That is, existing components can be changed and new client components added to the architecture without concern about other clients (because the client components operate independently).
- data can be passed among clients using the blackboard mechanism (i.e., the blackboard component serves to coordinate the transfer of information between clients). Client components independently execute processes.

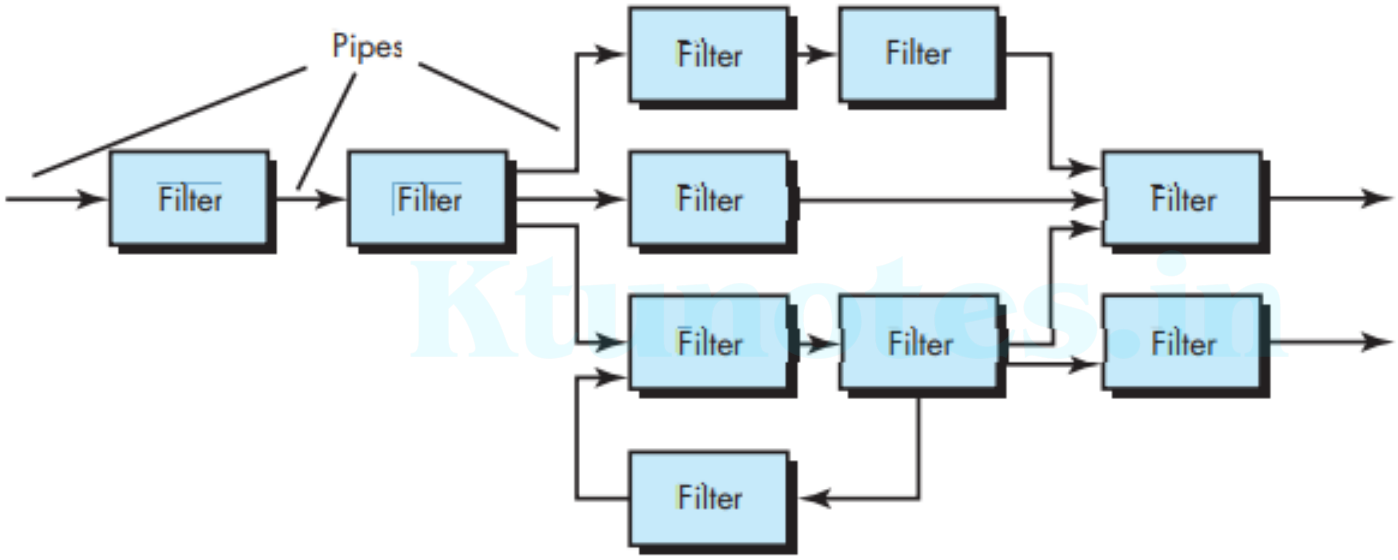
Data-Centered Architectures



Data-Flow Architectures

- This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data.
- A pipe-and-filter pattern has a set of components, called filters, connected by pipes that transmit data from one component to the next. Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form.
- The filter does not require knowledge of the workings of its neighboring filters.
- If the data flow degenerates into a single line of transforms, it is termed **batch sequential**. This structure accepts a batch of data and then applies a series of sequential components (filters) to transform it.

Data-Flow Architectures

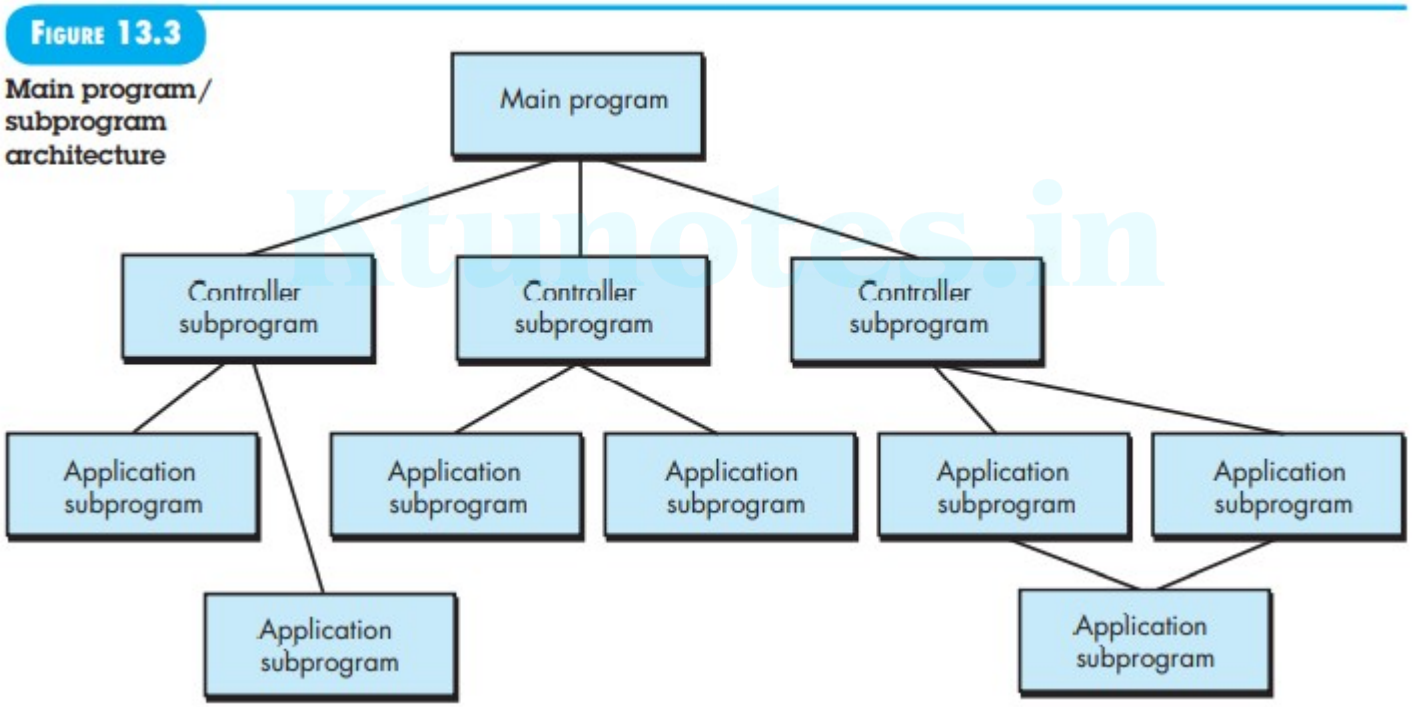


Pipes and filters

Call and Return Architectures

- This architectural style enables you to achieve a program structure that is relatively easy to modify and scale. A number of substyles exist within this category:
  - Main program/subprogram architectures. This classic program structure decomposes function into a control hierarchy where a “main” program invokes a number of program components, which in turn may invoke still other components.
  - Remote procedure call architectures. The components of a main program/ subprogram architecture are distributed across multiple computers on a network.

Call and Return Architectures



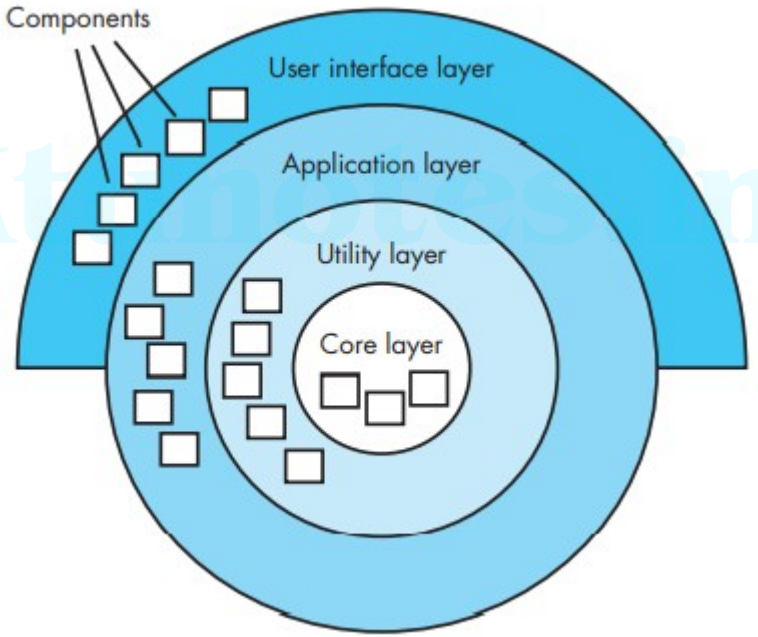
Object-Oriented Architectures

- The components of a system encapsulate data and the operations that must be applied to manipulate the data.
- Communication and coordination between components are accomplished via message passing.

Layered Architectures

- A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set.
- At the outer layer, components service user interface operations.
- At the inner layer, components perform operating system interfacing.
- Intermediate layers provide utility services and application software functions.

Layered Architectures



Architecture styles

- Choosing the right architecture style can be tricky.
- two complementary concepts provide some guidance.
  - Problem frames - describe characteristics of recurring problems, without being distracted by references to details of domain knowledge or programming solution implementations.
  - Domain-driven design - suggests that the software design should reflect the domain and the domain logic of the business problem you want to solve with your application

ARCHITECTURAL CONSIDERATIONS

- architectural considerations that can provide software engineers with guidance as architecture decisions are made:
  - Economy
  - Visibility
  - Spacing
  - Symmetry
  - Emergence

ARCHITECTURAL CONSIDERATIONS

- Economy: Many software architectures suffer from unnecessary complexity driven by the inclusion of unnecessary features or nonfunctional. The best software is uncluttered and relies on abstraction to reduce unnecessary detail.
- Visibility: As the design model is created, architectural decisions and the reasons for them should be obvious to software engineers who examine the model at a later time. Poor visibility arises when important design and domain concepts are poorly communicated to those who must complete the design and implement the system.
- Spacing: Separation of concerns in a design without introducing hidden dependencies is a desirable design concept that is sometimes referred to as spacing. Sufficient spacing leads to modular designs, but too much spacing leads to fragmentation and loss of visibility.
- Symmetry: Architectural symmetry implies that a system is consistent and balanced in its attributes. Symmetric designs are easier to understand, comprehend, and communicate.
- Emergence: Emergent, self-organized behavior and control are often the key to creating scalable, efficient, and economic software architectures.

ARCHITECTURAL DESIGN

- **Representing the System in Context**
- **Defining Archetypes**
- **Refining the Architecture into Components:**
- **Describing Instantiations of the System**

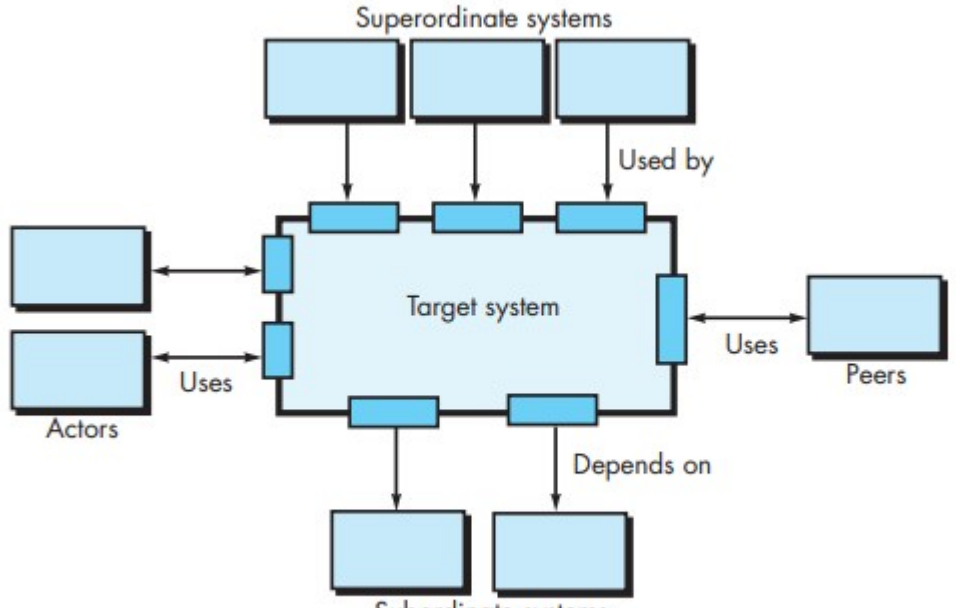
- **Architectural Design for Web Apps**
- **Architectural Design for Mobile Apps**

ARCHITECTURAL DESIGN

- As architectural design begins, context must be established.
- To accomplish this, the external entities (e.g., other systems, devices, people) that interact with the software and the nature of their interaction are described. This information can generally be acquired from the requirements model.
- Once context is modeled and all external software interfaces have been described, identify a set of architectural archetypes. An archetype is an abstraction (similar to a class) that represents one element of system behavior.
- The set of archetypes provides a collection of abstractions that must be modeled architecturally if the system is to be constructed, but the archetypes themselves do not provide enough implementation detail.
- Therefore, the designer specifies the structure of the system by defining and refining software components that implement each archetype.
- This process continues iteratively until a complete architectural structure has been derived.
- A number of questions must be asked and answered as a software engineer creates meaningful architectural diagrams. Does the diagram show how the system responds to inputs or events? Etc

Representing the System in Context

- At the architectural design level, a software architect uses an architectural context diagram (ACD) to model the manner in which software interacts with entities external to its boundaries



Representing the System in Context

systems that interoperate with the target system (the system for which an architectural design is to be developed) are represented as:

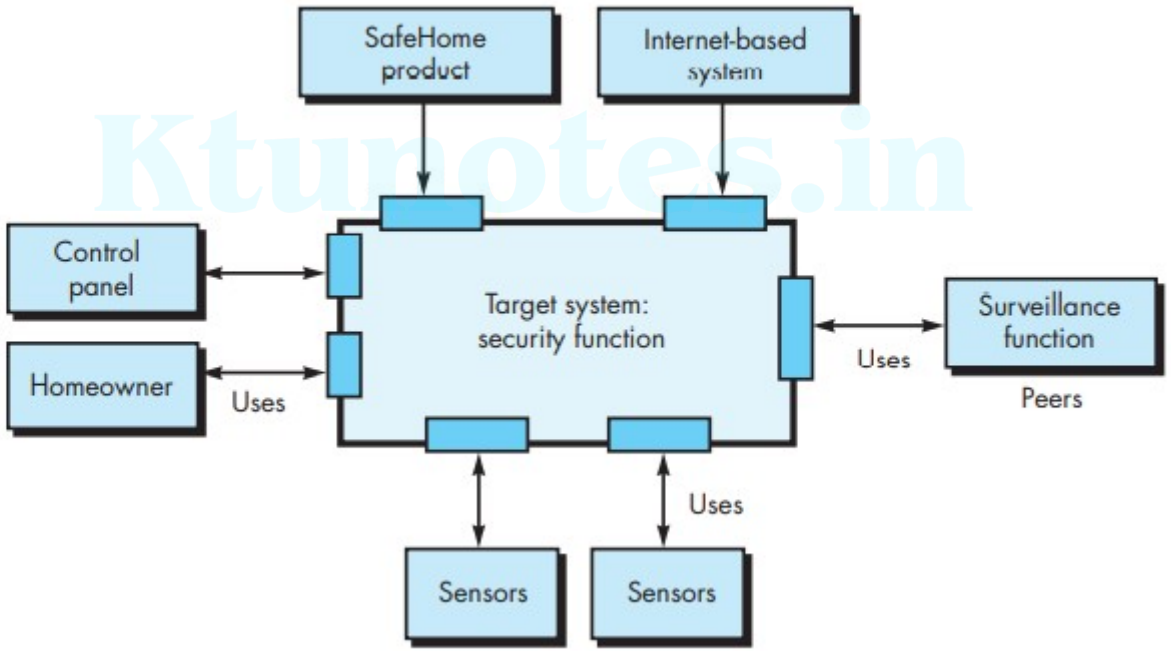
- Superordinate systems—those systems that use the target system as part of some higher-level processing scheme.
- Subordinate systems—those systems that are used by the target system and provide data or processing that are necessary to complete target system functionality.
- Peer-level systems—those systems that interact on a peer-to-peer basis (i.e., information is either produced or consumed by the peers and the target system.
- Actors—entities (people, devices) that interact with the target system by producing or consuming information that is necessary for requisite processing.

Each of these external entities communicates with the target system through an interface (the small shaded rectangles).

Representing the System in Context

FIGURE 13.6

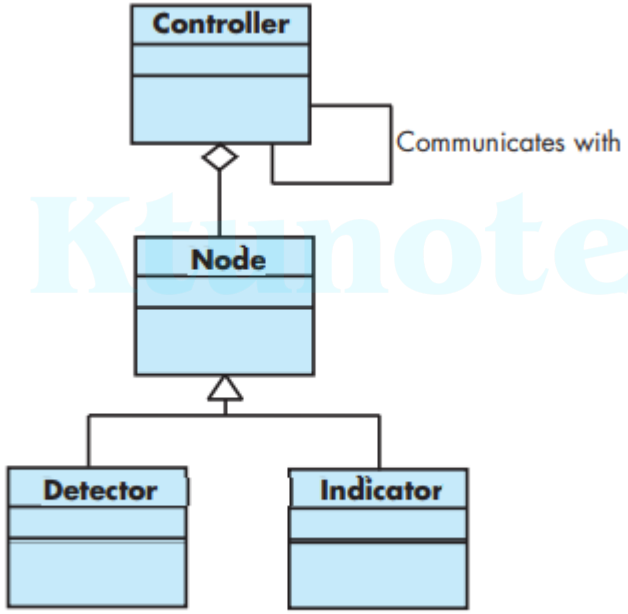
Architectural context diagram for the *SafeHome* security function



Defining Archetypes

- An archetype is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system.
- a relatively small set of archetypes is required to design even relatively complex systems.
- The target system architecture is composed of these archetypes, which represent stable elements of the architecture but may be instantiated many different ways based on the behavior of the system.
- In many cases, archetypes can be derived by examining the analysis classes defined as part of the requirements model.
- For the SafeHome home security function, we might define the following archetypes:
  - Node: Represents a cohesive collection of input and output elements of the home security function. For example, a node might be composed of (1) various sensors and (2) a variety of alarm (output) indicators.
  - Detector: An abstraction that encompasses all sensing equipment that feeds information into the target system.
  - Indicator. An abstraction that represents all mechanisms (e.g., alarm siren, flashing lights, bell) for indicating that an alarm condition is occurring.
  - Controller. An abstraction that depicts the mechanism that allows the arming or disarming of a node. If controllers reside on a network, they have the ability to communicate with one another.

- Each of these archetypes is depicted using UML notation



Refining the Architecture into Components:

- As the software architecture is refined into components, the structure of the system begins to emerge
- Application domain is one source for the derivation and refinement of components.
- Another source is the infrastructure domain.
- In some cases (e.g., a graphical user interface), a complete subsystem architecture with many components must be designed.

Refining the Architecture into Components:

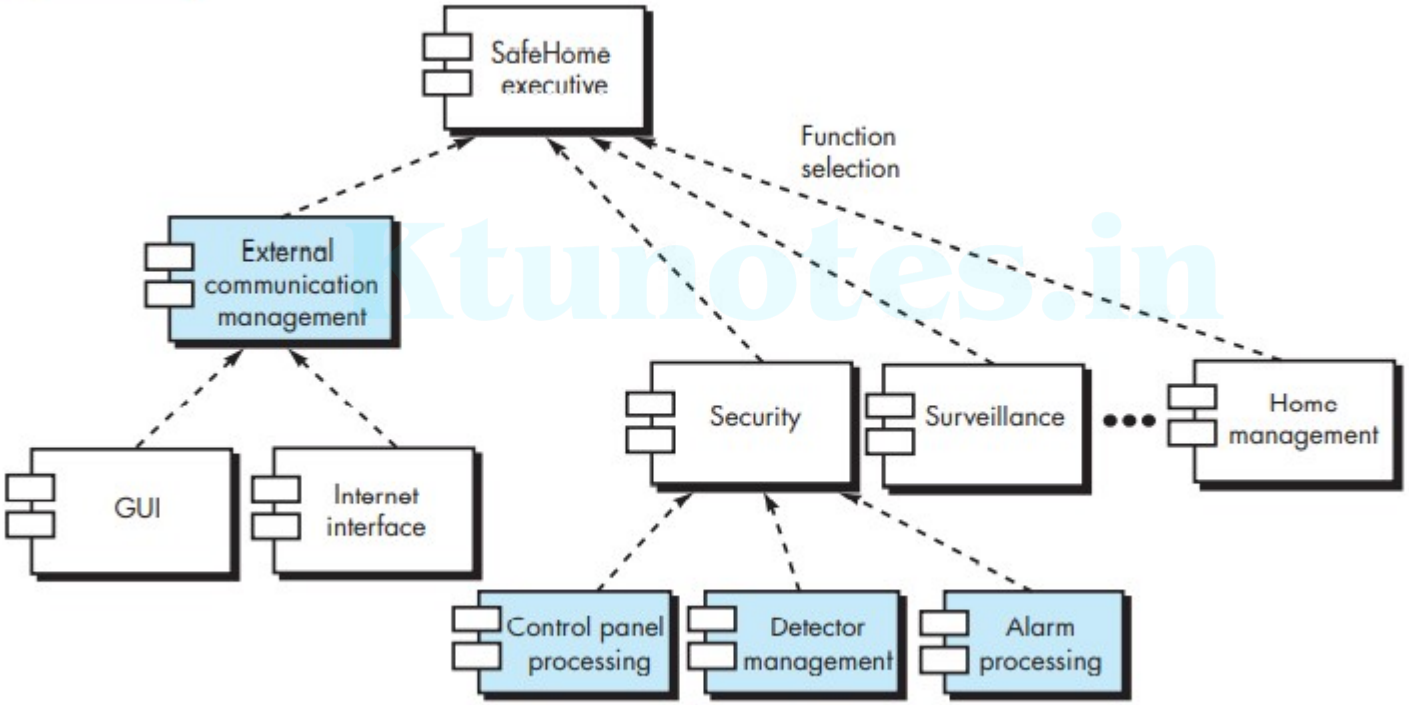
- SafeHome home security function example-define the set of top-level components that address the following functionality:
- External communication management —coordinates communication of the security function with external entities such as other Internet-based systems and external alarm notification.
- Control panel processing—manages all control panel functionality.
- Detector management —coordinates access to all detectors attached to the system.
- Alarm processing—verifies and acts on all alarm conditions.

Each of these top-level components would have to be elaborated iteratively and then positioned within the overall SafeHome architecture.

Design classes (with appropriate attributes and operations) would be defined for each

design details of all attributes and operations would not be specifi ed until component-level design

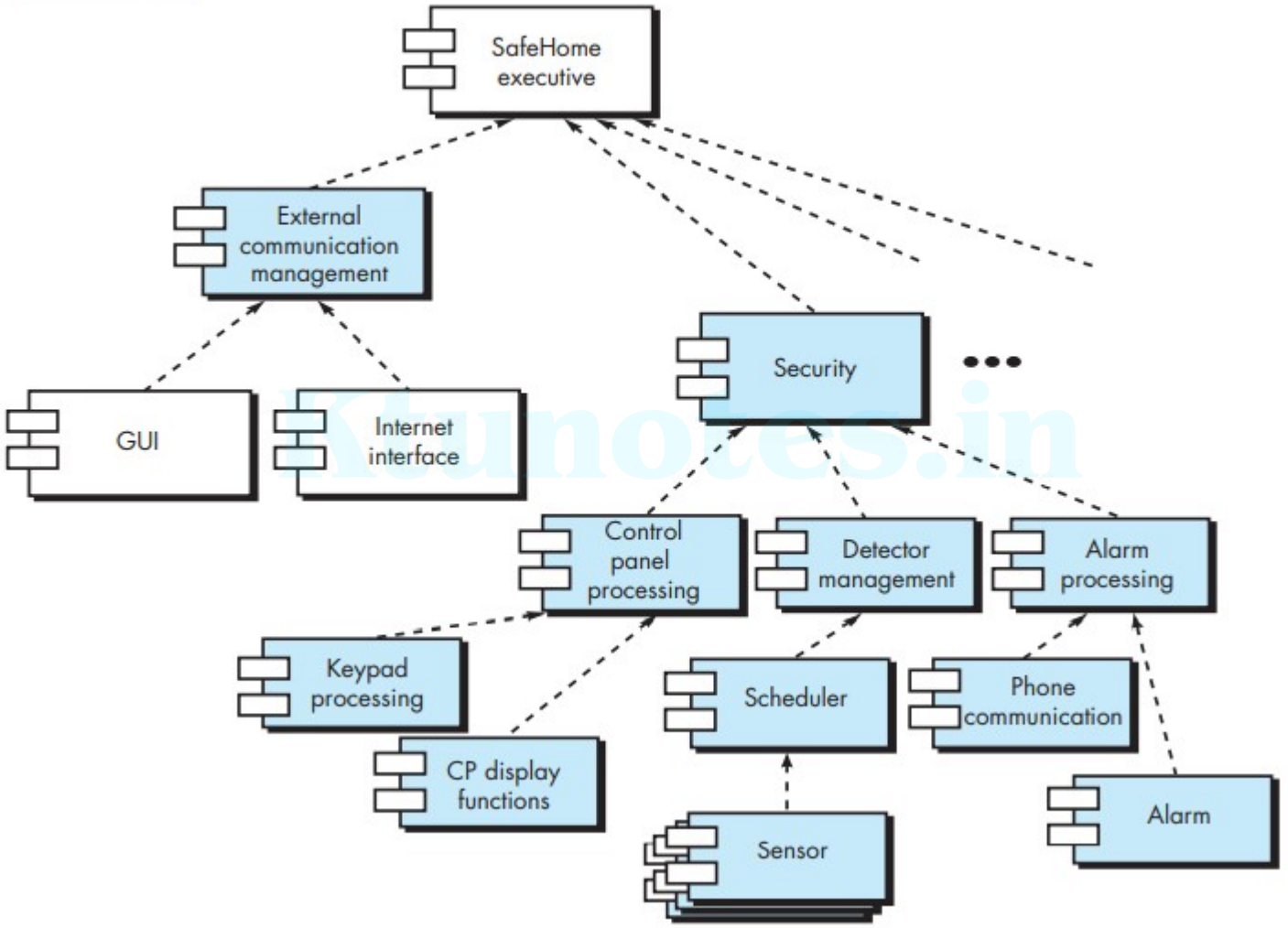
FIGURE 13.8 Overall architectural structure for *SafeHome* with top-level components



Describing Instantiations of the System

- an actual instantiation of the architecture is developed. the architecture is applied to a specific problem with the intent of demonstrating that the structure and components are appropriate.
- instantiation of the SafeHome architecture for the security system. Components shown in Figure(next slide) are elaborated to show additional detail.
- detector management component interacts with a scheduler infrastructure component that implements polling of each sensor object used by the security system.

FIGURE 13.9 An instantiation of the security function with component elaboration



Architectural Design for Web Apps

- WebApps are client-server applications typically structured using multilayered architectures, including a user interface or view layer, a controller layer which directs the flow of information to and from the client browser based on a set of business rules, and a content or model layer that may also contain the business rules for the WebApp.
- The user interface for a WebApp is designed around the characteristics of the web browser running on the client machine (usually a personal computer or mobile device).
- Data layers reside on a server. Business rules can be implemented using a server-based scripting language such as PHP or a client-based scripting language such as javascript.
- An architect will examine requirements for security and usability to determine which features should be allocated to the client or server. The architectural design of a WebApp is also influenced by the structure (linear or nonlinear) of the content that needs to be accessed by the client.

Architectural Design for Mobile Apps

- Mobile apps are typically structured using multilayered architectures, including a user interface layer, a business layer, and a data layer.
- With mobile apps you have the choice of building a thin Web-based client or a rich client. With a thin client, only the user interface resides on the mobile device, whereas the business and data layers reside on a server. With a rich client all three layers may reside on the mobile device itself.
- Mobile devices differ from one another in terms of their physical characteristics (e.g., screen sizes, input devices), software (e.g., operating systems, language support), and hardware (e.g., memory, network connections). Each of these attributes shapes the direction of the architectural alternatives that can be selected.
- considerations that can influence the architectural design of a mobile app:
  (1) the type of web client (thin or rich) to be built,
  (2) the categories of devices (e.g., smartphones, tablets) that are supported,
  (3) the degree of connectivity (occasional or persistent) required,
  (4) the bandwidth required,
  (5) the constraints imposed by the mobile platform,
  (6) the degree to which reuse and maintainability are important, and
  (7) device resource constraints (e.g., battery life, memory size, processor speed).

Component-level design

- A complete set of software components is defined during architectural design.
- Component-level design defi nes the data structures, algorithms, interface characteristics, and communication mechanisms allocated to each software component.

What is a component?

An object oriented view

The traditional view

A process related view

What is a component?

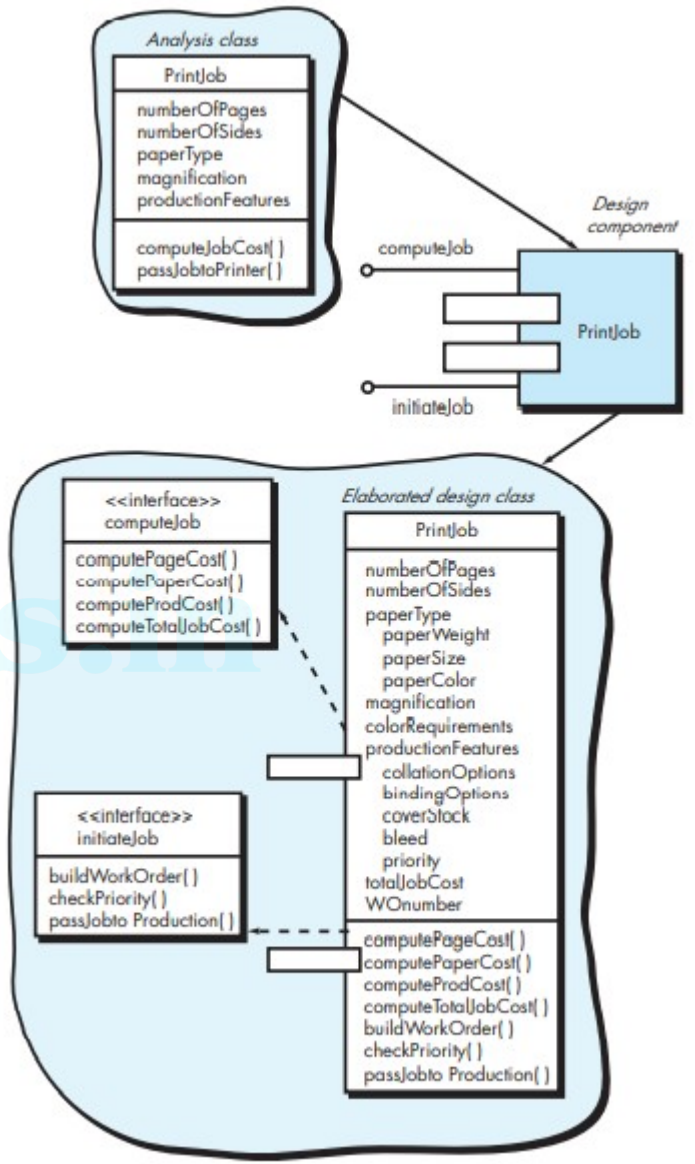
- A component is a modular building block for computer software.
- “a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces
- components populate the software architecture and, as a consequence, play a role in achieving the objectives and requirements of the system to be built.
- Because components reside within the software architecture, they must communicate and collaborate with other components and with entities (e.g., other systems, devices, people) that exist outside the boundaries of the software.

Component

An Object-Oriented View

- In the context of object-oriented software engineering, a component contains a set of collaborating classes.
- Each class within a component has been fully elaborated to include all attributes and operations that are relevant to its implementation.
- As part of the design elaboration, all interfaces that enable the classes to communicate and collaborate with other design classes must also be defined.
- begin with the analysis model and elaborate analysis classes (for components that relate to the problem domain) and infrastructure classes (for components that provide support services for the problem domain).

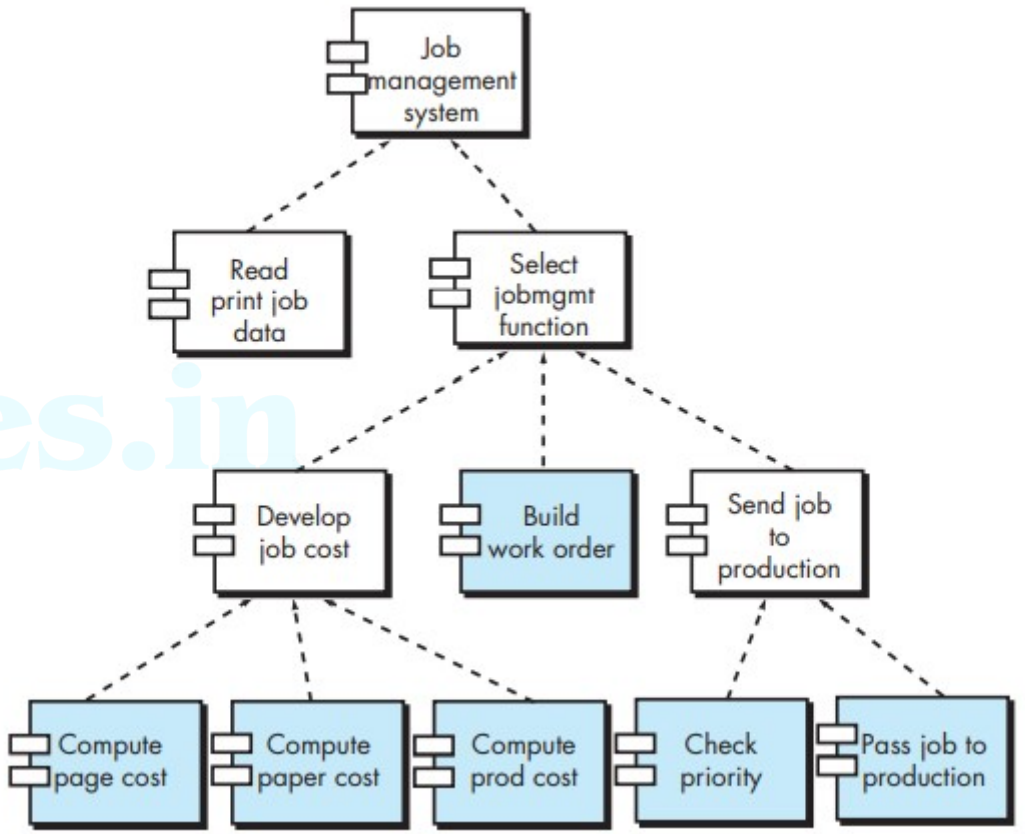
- **An Object-Oriented View**
- consider software to be built for a sophisticated print shop. The overall intent of the software is to collect the customer's requirements at the front counter, cost a print job, and then pass the job on to an automated production facility.



Component

- **The Traditional View**
- a component is a functional element of a program that incorporates processing logic, the internal data structures that are required to implement the processing logic, and an interface that enables the component to be invoked and data to be passed to it.
- A traditional component, also called a module, resides within the software architecture and serves one of three important roles: (1) a control component that coordinates the invocation of all other problem domain components, (2) a problem domain component that implements a complete or partial function that is required by the customer, or (3) an infrastructure component that is responsible for functions that support the processing required in the problem domain.
- traditional software components are derived from the analysis model.

Structure chart for a traditional system



Component

- **A Process-Related View**
- create a new component based on specifications derived from the requirements model.
- As the software architecture is developed, you choose components or design patterns from the catalog and use them to populate the architecture. Because these components have been created with reusability in mind, a complete description of their interface, the function(s) they perform, and the communication and collaboration they require are all available to you.

DESIGNING CLASS -BASED COMPONENTS

Basic Design Principles

Component-Level Design Guidelines

Cohesion

Coupling

DESIGNING CLASS -BASED COMPONENTS

- When an object-oriented software engineering approach is chosen, component-level design focuses on the elaboration of problem domain specific classes and the definition and refinement of infrastructure classes contained in the requirements model.
- The detailed description of the attributes, operations, and interfaces used by these classes is the design detail required as a precursor to the construction activity.

Basic Design Principles

- The Open-Closed Principle (OCP).

The Open-Closed Principle (OCP).

- A module [component] should be open for extension but closed for modification.
- specify the component in a way that allows it to be extended (within the functional domain that it addresses) without the need to make internal (code or logic-level) modifications to the component itself.
- To accomplish this, create abstractions that serve as a buffer between the functionality that is likely to be extended and the design class itself.
- [For example, assume that the SafeHome security function makes use of a Detector class that must check the status of each type of security sensor. It is likely that as time passes, the number and types of security sensors will grow. If internal processing logic is implemented as a sequence of if-then-else constructs, each addressing a different sensor type, the addition of a new sensor type will require additional internal processing logic (still another if-then-else). This is a violation of OCP. One way to accomplish OCP for the Detector class is illustrated in Figure. The sensor interface presents a consistent view of sensors to the detector component. If a new type of sensor is added no change is required for the Detector class (component). The OCP is preserved.]