MODULE 2

Requirement Analysis and Design (8 hours)

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 Arianne 5 launcher failure.

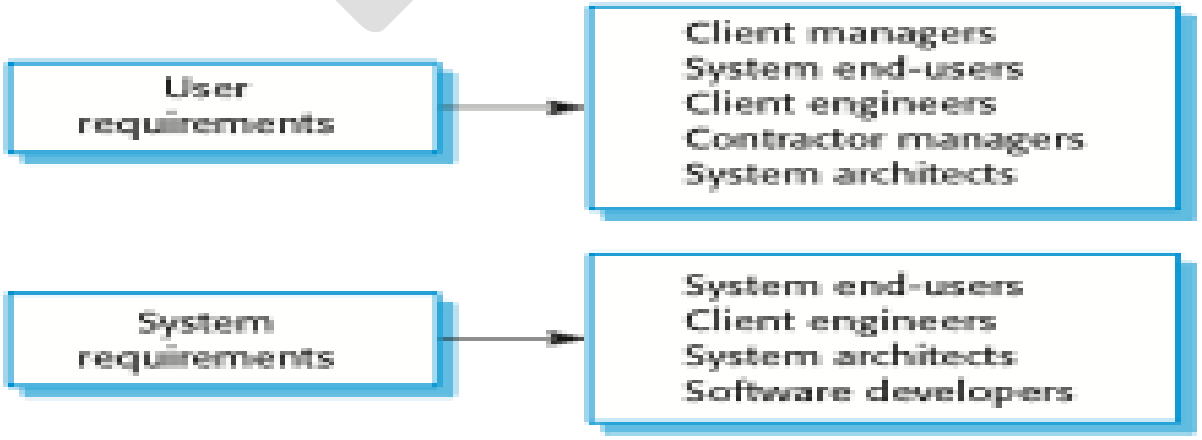
Requirements Engineering

The process of establishing the services that the customer requires from a system and the constraints under which it operates and is developed. The requirements themselves are the descriptions of the system services and constraints that are generated during the requirements engineering process. It may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification.

Types of requirement

- **User requirements-**
  Statements in natural language plus diagrams of the services the system provides and its operational constraints. Written for customers.
- **System requirements-**
  A structured document setting out detailed descriptions of the system's functions, services and operational constraints. Defines what should be implemented so may be part of a contract between client and contractor**.**

Readers of different types of requirements specification



Functional and non-functional requirements

- **Functional requirements-**
  Statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.

  May state what the system should not do.

- **Non-functional requirements-**

  Constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.

  Often apply to the system as a whole rather than individual features or services.

- **Domain requirements**

  Constraints on the system from the domain of operation

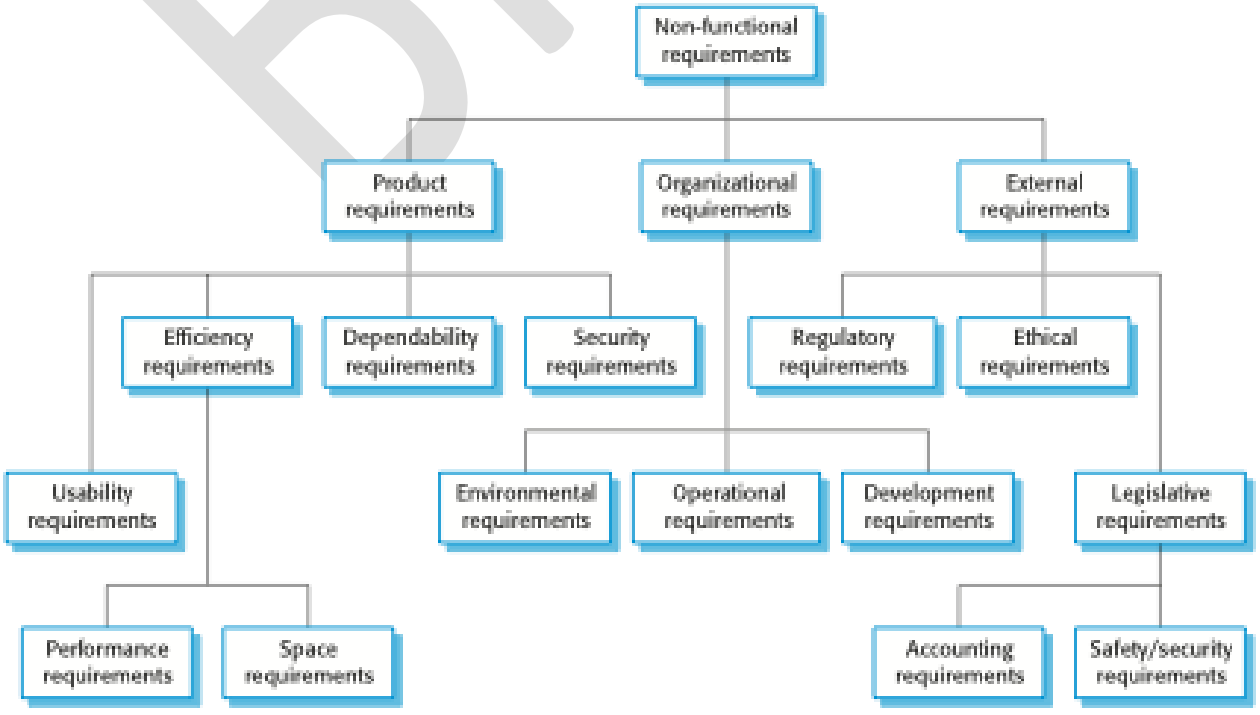
Functional requirements

- Describe functionality or system services.
- Depend on the type of software, expected users and the type of system where the software is used.
- Functional user requirements may be high-level statements of what the system should do.
- Functional system requirements should describe the system services in detail**.**

Non-functional requirements

These define system properties and constraints e.g. reliability, response time and storage requirements. Constraints are I/O device capability, system representations, etc. Process requirements may also be specified mandating a particular IDE, programming language or development method. Non-functional requirements may be more critical than functional requirements. If these are not met, the system may be useless.

Types of nonfunctional requirement



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

Non-functional classifications

- **Product requirements**
  Requirements which specify that the delivered product must behave in a particular way e.g. execution speed, reliability, etc.
- **Organisational requirements**
  Requirements which are a consequence of organisational policies and procedures e.g. process standards used, implementation requirements, etc.
- **External requirements**
  Requirements which arise from factors which are external to the system development process e.g. interoperability requirements, legislative requirements etc.
- **Usability requirements**

The system should be easy to use by medical staff and should be organized in such a way that user errors are minimized. (Goal). Medical staff shall be able to use all the system functions after four hours of training. After this training, the average number of errors made by experienced users shall not exceed two per hour of system use. (Testable non-functional requirement).

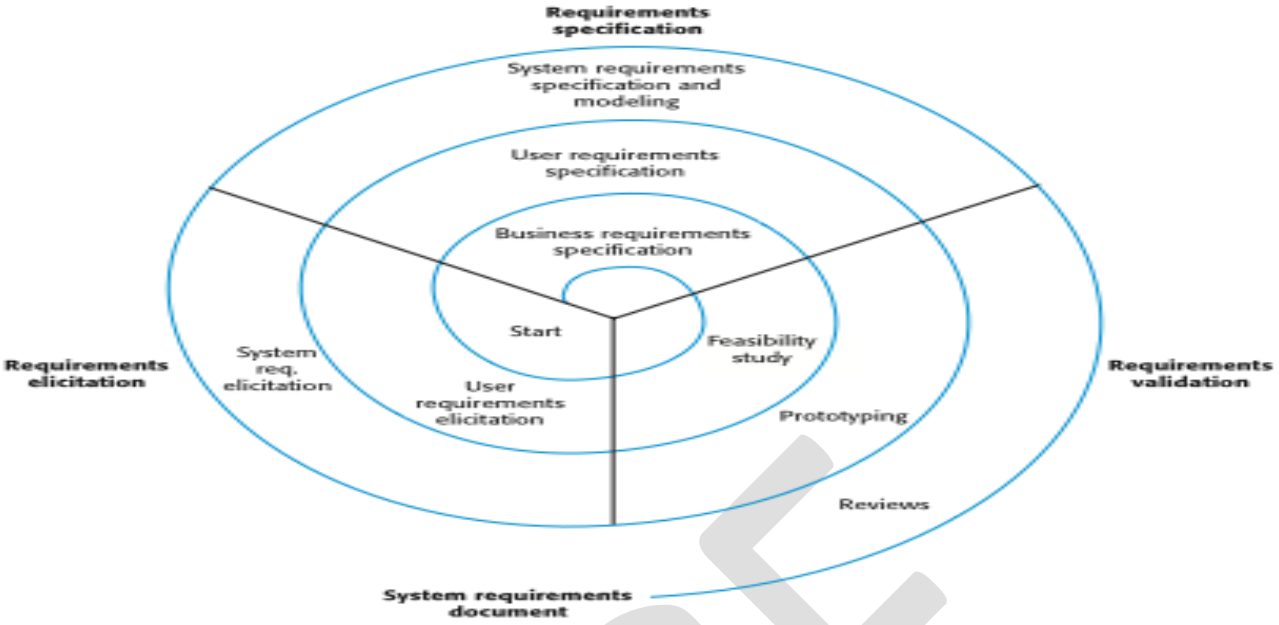
Metrics for specifying nonfunctional requirements

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

Requirements engineering processes

The processes used for Requirement Engineering vary widely depending on the application domain, the people involved and the organisation developing the requirements. Requirement Engineering is an iterative activity in which these processes are interleaved.

A spiral view of the requirements engineering process



1. **Requirements elicitation and analysis**

- Sometimes called requirements elicitation or requirements discovery.
- Involves technical staff working with customers to find out about the application domain, the services that the system should provide and the system's operational constraints.
- May involve end-users, managers, engineers involved in maintenance, domain experts, trade unions, etc. These are called *stakeholders.*

Problems of requirements analysis

- Stakeholders don't know what they really want.
- Stakeholders express requirements in their own terms.
- Different stakeholders may have conflicting requirements.
- Organisational and political factors may influence the system requirements.
- The requirements change during the analysis process. New stakeholders may emerge and the business environment may change.

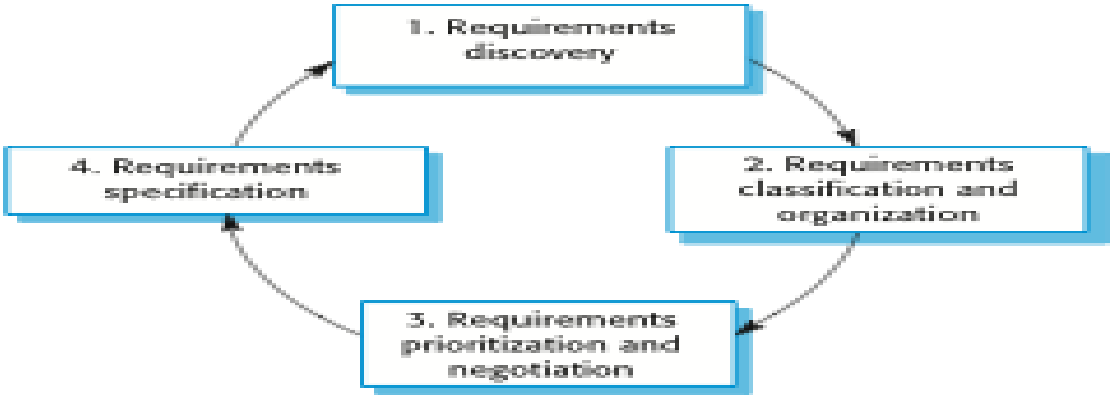
Requirements elicitation and analysis

Software engineers work with a range of system stakeholders to find out about the application domain, the services that the system should provide, the required system performance, hardware constraints, other systems, etc.

Stages include:

- **Requirements discovery and understanding:** process of interacting with stake holders to discover their requirements. Domain requirements from stakeholders and documentation are also discovered.
- **Requirements classification and organization:** this activity takes the unstructured collection of requirements, groups related requirements and organizes them into coherent clusters**.**
- **Requirements prioritization and negotiation:** when multiple stakeholders are involved, requirements will conflict. This activity is concerned with prioritizing requirements and finding and resolving requirements conflicts through negotiation.
- **Requirements specification (documentation):** requirements are documented and input into the next round of spiral.

The requirements elicitation and analysis process



Requirements discovery (elicitation techniques)

The process of gathering information about the required and existing systems and distilling the user and system requirements from this information. Interaction is with system stakeholders from managers to external regulators. Systems normally have a range of stakeholders.

- ✓ **Interviewing**

Formal or informal interviews with stakeholders are part of most RE processes.

Types of interview

- **Closed interviews : stakeholders answers based on pre-determined list of questions**
- **Open interviews : in which there is no predefined agenda, where various issues are explored with stakeholders**

Effective interviewing

- **Be open-minded, avoid pre-conceived ideas about the requirements and are willing to listen to stakeholders.**
- **Prompt the interviewee to get discussions going using a springboard question, a requirements proposal, or by working together on a prototype system.**

- ✓ **Scenarios**

Scenarios are real-life examples of how a system can be used.

They should include

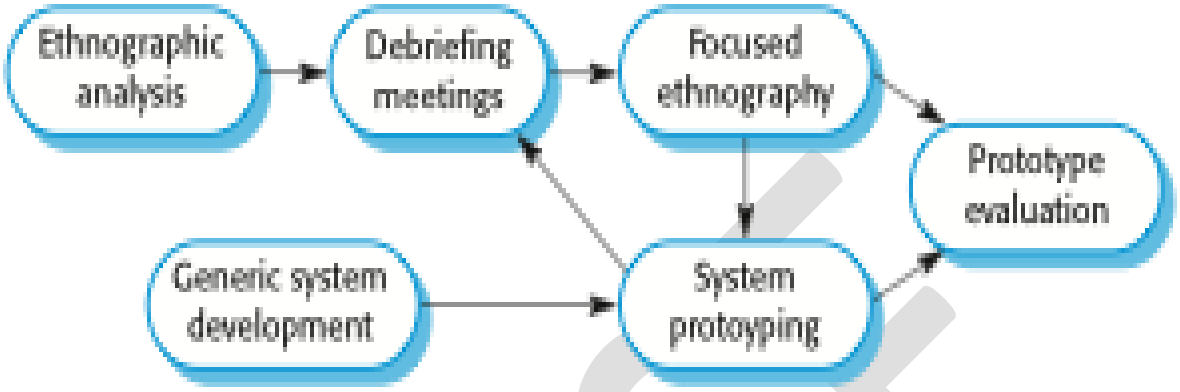
- A description of the starting situation;
- A description of the normal flow of events;
- A description of what can go wrong;
- Information about other concurrent activities;
- A description of the state when the scenario finishes**.**

- ✓ **Ethnography**
  - A social scientist spends a considerable time observing and analysing how people actually work.People do not have to explain or articulate their work.
  - Social and organisational factors of importance may be observed.
  - Ethnographic studies have shown that work is usually richer and more complex than suggested by simple system models.
  - Requirements that are derived from cooperation and awareness of other people's activities.
  - Awareness of what other people are doing leads to changes in the ways in which we do things.
  - Ethnography is effective for understanding existing processes but cannot identify new features that should be added to a system.

Focused ethnography

- Developed in a project studying the air traffic control process.
- Combines ethnography with prototyping
- Prototype development results in unanswered questions which focus the ethnographic analysis.
- The problem with ethnography is that it studies existing practices which may have some historical basis which is no longer relevant**.**

Ethnography and prototyping for requirements analysis



2. **Requirements specification**
   - The process of writing the user and system requirements in a requirements document.
   - User requirements have to be understandable by end-users and customers who do not have a technical background.
   - System requirements are more detailed requirements and may include more technical information.
   - The requirements may be part of a contract for the system development
   - **It is therefore important that these are as complete as possible.**
     Ways of writing a system requirements specification

Notation	Description
Natural language	The requirements are written using numbered sentences in natural language. Each sentence should express one requirement.
Structured natural language	The requirements are written in natural language on a standard form or template. Each field provides information about an aspect of the requirement.
Design description languages	This approach uses a language like a programming language, but with more abstract features to specify the requirements by defining an operational model of the system. This approach is now rarely used although it can be useful for interface specifications.
Graphical notations	Graphical models, supplemented by text annotations, are used to define the functional requirements for the system; UML use case and sequence diagrams are commonly used.
Mathematical specifications	These notations are based on mathematical concepts such as finite-state machines or sets. Although these unambiguous specifications can reduce the ambiguity in a requirements document, most customers don't understand a formal specification. They cannot check that it represents what they want and are reluctant to accept it as a system contract

- ✓ **Requirements and design**

In principle, requirements should state what the system should do and the design should describe how it does this.

In practice, requirements and design are inseparable

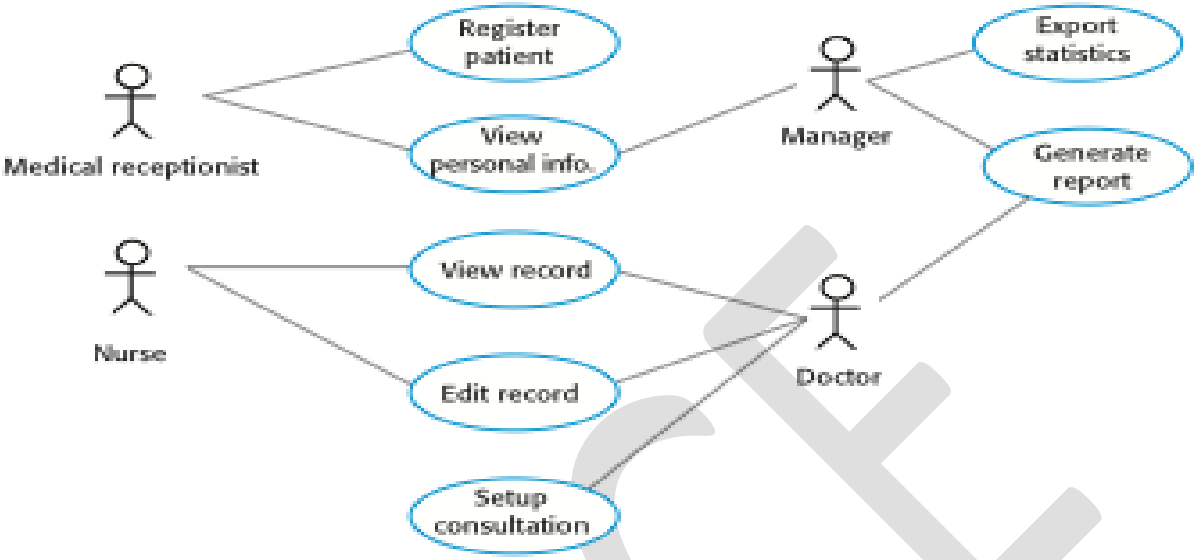
- A system architecture may be designed to structure the requirements;
- The system may inter-operate with other systems that generate design requirements;
- The use of a specific architecture to satisfy non-functional requirements may be a domain requirement.
- This may be the consequence of a regulatory requirement.

- ✓ **Natural language specification**
  - Requirements are written as natural language sentences supplemented by diagrams and tables.
  - Used for writing requirements because it is expressive, intuitive and universal. This means that the requirements can be understood by users and customers**.**
- ✓ **Guidelines for writing requirements**
  - Invent a standard format and use it for all requirements.
  - Use language in a consistent way. Use shall for mandatory requirements, should for desirable requirements.
  - Use text highlighting to identify key parts of the requirement.
  - Avoid the use of computer jargon.
  - Include an explanation (rationale) of why a requirement is necessary.
- ✓ **Problems with natural language**
  - Lack of clarity
    Precision is difficult without making the document difficult to read.
  - Requirements confusion
    Functional and non-functional requirements tend to be mixed-up.
  - Requirements amalgamation
    Several different requirements may be expressed together**.**
- ✓ **Structured specifications**
  - An approach to writing requirements where the freedom of the requirements writer is limited and requirements are written in a standard way.
  - This works well for some types of requirements e.g. requirements for embedded control system but is sometimes too rigid for writing business system requirements.
- ✓ Form-based specifications
  - Definition of the function or entity.
  - Description of inputs and where they come from.
  - Description of outputs and where they go to.
  - Information about the information needed for the computation and other entities used.
  - Description of the action to be taken.
  - Pre and post conditions (if appropriate).
  - The side effects (if any) of the function.
- ✓ **Tabular specification**
- Used to supplement natural language.
- Particularly useful when you have to define a number of possible alternative courses of action.

✓ **Use cases**

- Use-cases are a scenario based technique in the UML which identify the actors in an interaction and which describe the interaction itself.
- A set of use cases should describe all possible interactions with the system.

Use cases for the MHC-PMS



✓ **Developing Use cases**

- A use case tells a stylized story about how an end user (playing one of a number of possible roles) interacts with the system under a specific c set of circumstances.
- The story may be narrative text, an outline of tasks or interactions, a template-based description, or a diagrammatic representation.
- A use case depicts the software or system from the end user's point of view.
- The first step in writing a use case is to define the set of "actors" that will be involved in the story.
- Actors are the different people (or devices) that use the system or product within the context of the function and behavior that is to be described.
- Actors represent the roles that people (or devices) play as the system operates.
- An actor is anything that communicates with the system or product and that is external to the system itself.
- Primary actors → interact to achieve required system function and derive the intended benefit from the system. They work directly and frequently with the software.
- Secondary actors → support the system so that primary actors can do their work.
- Once actors have been identified, use cases can be developed

3. Requirements validation

- Concerned with demonstrating that the requirements define the system that the customer really wants.
- Requirements error costs are high so validation is very important.
- Fixing a requirements error after delivery may cost up to 100 times the cost of fixing an implementation error.

✓ **Requirements checking**

- **Validity. Does the system provide the functions which best support the customer's needs?**
- **Consistency. Are there any requirements conflicts?**
- **Completeness. Are all functions required by the customer included?**

- **Realism. Can the requirements be implemented given available budget and technology**
- **Verifiability. Can the requirements be checked?**

✓ **Requirements validation techniques**

- **Requirements reviews**
  Systematic manual analysis of the requirements:requirements are analysed systematically by a team of reviewers who check for errors and inconsistencies.
- **Prototyping**
  Using an executable model of the system to check requirements.
- **Test-case generation**
  Developing tests for requirements to check testability.

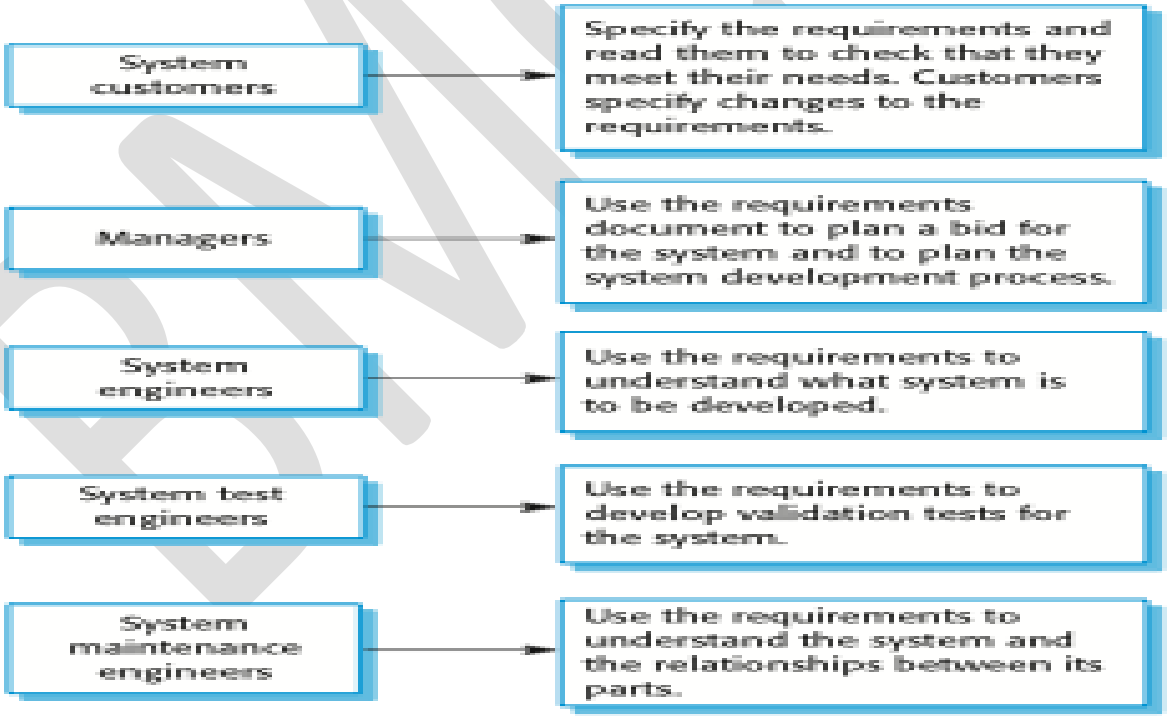
✓ **Software Requirements Document**

- The software requirements document is the official statement of what is required of the system developers.
- Should include both a definition of user requirements and a specification of the system requirements.
- It is NOT a design document. As far as possible, it should set of WHAT the system should do rather than HOW it should do it.

The 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 nonfunctional 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.

Chapter	Description
System requirements specification	This should describe the functional and nonfunctional requirements in more detail. If necessary, further detail may also be added to the nonfunctional requirements. Interfaces to other systems may be defined.
System models	This might include graphical system models showing the relationships between the system components and 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 so on.



Users of a requirements document

INFO

Software Requirements Specification Template

A *software requirements specification* (SRS) is a work product that is created when a detailed description of all aspects of the software to be built must be specified before the project is to commence. It is important to note that a formal SRS is not always written. In fact, there are many instances in which effort expended on an SRS might be better spent in other software engineering activities. However, when software is to be developed by a third party, when a lack of specification would create severe business issues, or when a system is extremely complex or business critical, an SRS may be justified.

Karl Wiegers [Wie03] of Process Impact Inc. has developed a worthwhile template (available at **www.processimpact.com/process_assets/srs_template.doc**) that can serve as a guideline for those who must create a complete SRS. A topic outline follows:

Table of Contents

Revision History

1. **Introduction**
   - 1.1 Purpose
   - 1.2 Document Conventions
   - 1.3 Intended Audience and Reading Suggestions
   - 1.4 Project Scope
   - 1.5 References
2. **Overall Description**
   - 2.1 Product Perspective
   - 2.2 Product Features
   - 2.3 User Classes and Characteristics
   - 2.4 Operating Environment
   - 2.5 Design and Implementation Constraints
   - 2.6 User Documentation
   - 2.7 Assumptions and Dependencies
3. **System Features**
   - 3.1 System Feature 1
   - 3.2 System Feature 2 (and so on)
4. **External Interface Requirements**
   - 4.1 User Interfaces
   - 4.2 Hardware Interfaces
   - 4.3 Software Interfaces
   - 4.4 Communications Interfaces
5. **Other Nonfunctional Requirements**
   - 5.1 Performance Requirements
   - 5.2 Safety Requirements
   - 5.3 Security Requirements
   - 5.4 Software Quality Attributes
6. **Other Requirements**

Appendix A: Glossary
Appendix B: Analysis Models
Appendix C: Issues List

A detailed description of each SRS topic can be obtained by downloading the SRS template at the URL noted in this sidebar.

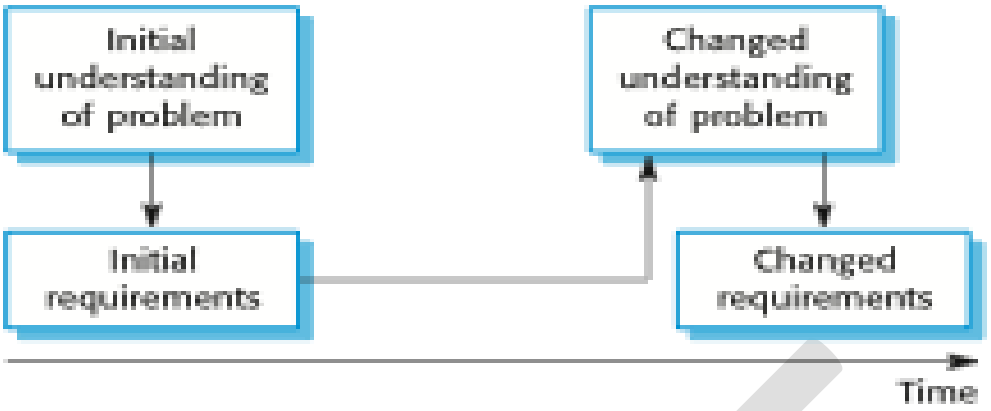
Requirements Management

- Requirements management is the process of managing changing requirements during the requirements engineering process and system development.
- New requirements emerge as a system is being developed and after it has gone into use.
- 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 need to establish a formal process for making change proposals and linking these to system requirements.

✓ **Changing requirements**

- The business and technical environment of the system always changes after installation.
- The people who pay for a system and the users of that system are rarely the same people.
- Large systems usually have a diverse user community, with many users having different requirements and priorities that may be conflicting or contradictory.

✓ **Requirements evolution**



Requirements management planning

Establishes the level of requirements management detail that is required.

Requirements management decisions:

- Requirements identification: Each requirement must be uniquely identified so that it can be cross-referenced with other requirements.
- A change management process: This is the set of activities that assess the impact and cost of changes. I discuss this process in more detail in the following section.
- Traceability policies: These policies define the relationships between each requirement and between the requirements and the system design that should be recorded**.**
- Tool support: Tools that may be used range from specialist requirements management systems to spreadsheets and simple database systems.

Requirements change management

Deciding if a requirements change should be accepted

Problem analysis and change specification

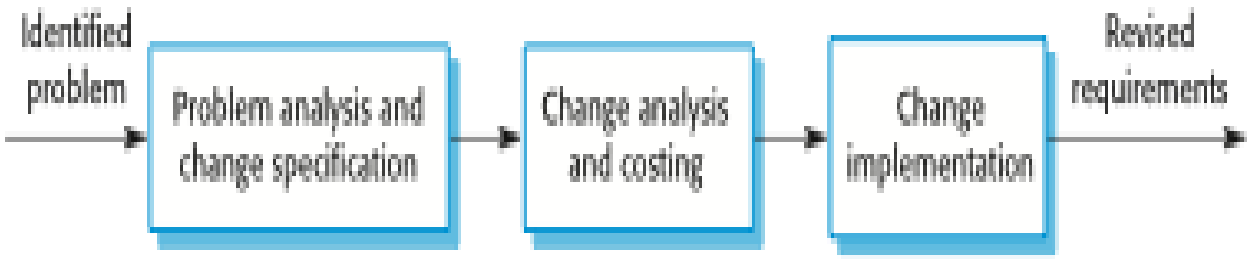
- During this stage, the problem or the change proposal is analyzed to check that it is valid. This analysis is fed back to the change requestor who may respond with a more specific requirements change proposal, or decide to withdraw the request.

Change analysis and costing

- The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements. Once this analysis is completed, a decision is made whether or not to proceed with the requirements change.

Change implementation

- The requirements document and, where necessary, the system design and implementation, are modified. Ideally, the document should be organized so that changes can be easily implemented



Traceability Matrix

- Is an Engg team that refers to documented links between Software Engg work products (Eg Requirements and test cases)
- Traceability matrix allows a requirement engineer to represent the relationship between requirements and other work products.
- Rows of the matrix are labelled using requirement names and columns can be labelled with the name of Software Engg work product.
- A matrix cell is marked to indicate the presence of link between the two.
- 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 ensure that every requirement is covered in the form of a Test case so that we don't miss out any testing.
- Map all the requirements and corresponding test cases to ensure that we have written all the test cases for each condition.
- This matrix can support a variety of Engg development activities.
- They can provide continuity for developers as a project moves from one project phase to another.
- It can be used to ensure the Engg work products have taken all requirements into account.
- As the no: of req and the number of work products grows.it become increasingly difficult to keep the traceability up to date.

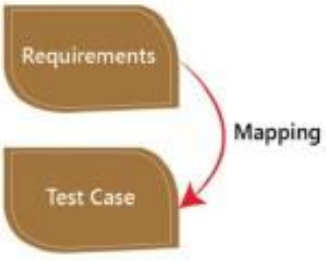
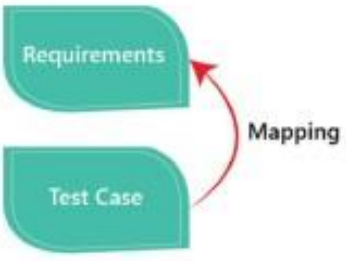
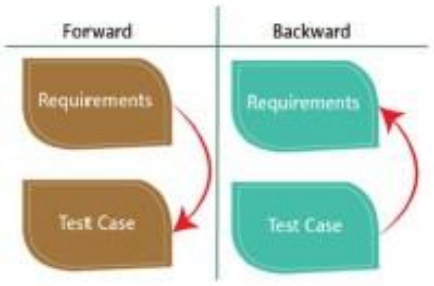
	A	B	C	D	E
1	RTM Template				
2	**Requirement number**	**Module name**	**High level requirement**	**Low level requirement**	**Test case name**
3	2	Loan	**2.1 Personal loan**	2.1.1--> personal loan for private employee	beta-2.0-personal loan
4				2.1.2--> personal loan for government employee	
5				2.1.3--> 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					

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

1. Forward traceability

2. Backward or reverse traceability

3. Bi-directional traceability

Forward Traceability	Backward Traceability	Bi-directional Traceability
• Used to ensure that every business's needs or requirements are executed correctly in the application and also tested rigorously. • Requirements are mapped into the forward direction to the test cases.	• 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. • Requirements are mapped into the backward direction to the test cases.	• A combination of forwarding and backward traceability matrix. • Used to make sure that all the business needs are executed in the test cases. • Also evaluates the modification in the requirement which is occurring due to the bugs in the application.
		

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.

✓ **Advantages of RTM:**

• With the help of the RTM document, we can display the complete test execution and bugs status based on requirements.

• It is used to show the missing requirements or conflicts in documents.

• We can ensure the complete test coverage, which means all the modules are tested.

• It will also consider the efforts of the testing teamwork towards reworking or reconsidering on the test cases.

Personas, Scenarios and Stories ,Feature Identification

From personas to features

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

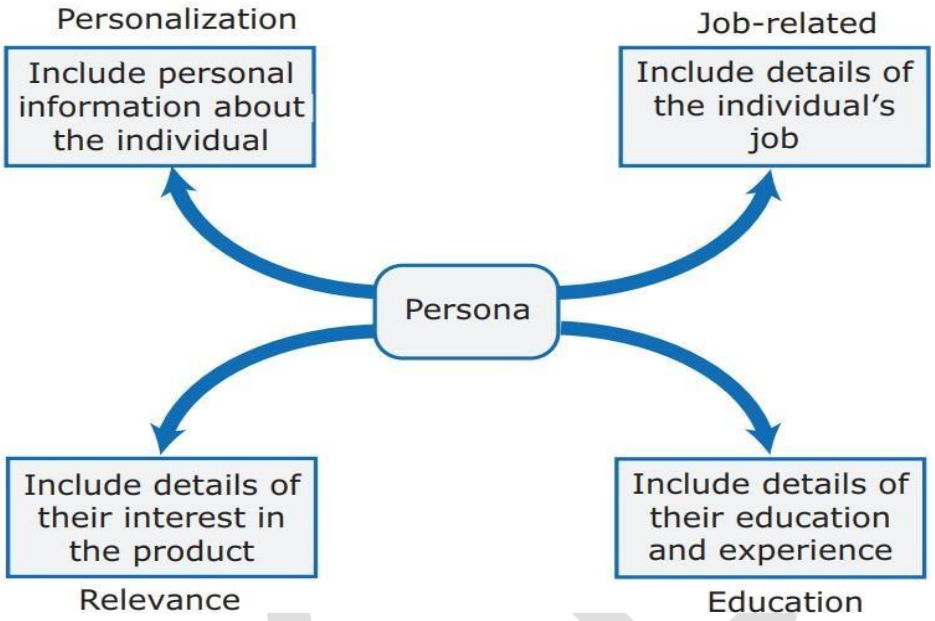
PERSONAS

- Personas are about "imagined users," character portraits of types of user that you think might adopt your product.
- Ex: if your product is aimed at managing appointments fordentists, you might create a dentist persona, a receptionist persona, and a patient persona.
- Personas of different types of users help to 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.
- There is no standard way to represent personas

Persona should include the following:

- ✓ Description about the users' backgrounds
- ✓ Description about why the users might want to use your product
- ✓ Description about their education and technical skills.
- Personas should be relatively short and easy to read.
- Personas are a tool that allows team members to "step into the users' shoes." Instead of thinking about what they would do in a particular situation, they can imagine how a persona would behave and react.
- They can help you check your ideas to ensure that you are not including product features that aren't really needed.
- They help you to avoid making unwarranted assumptions, based on your own knowledge, and designing an overcomplicated or irrelevant product.
- Personas, scenarios and user stories lead to features that might be implemented in a software product.

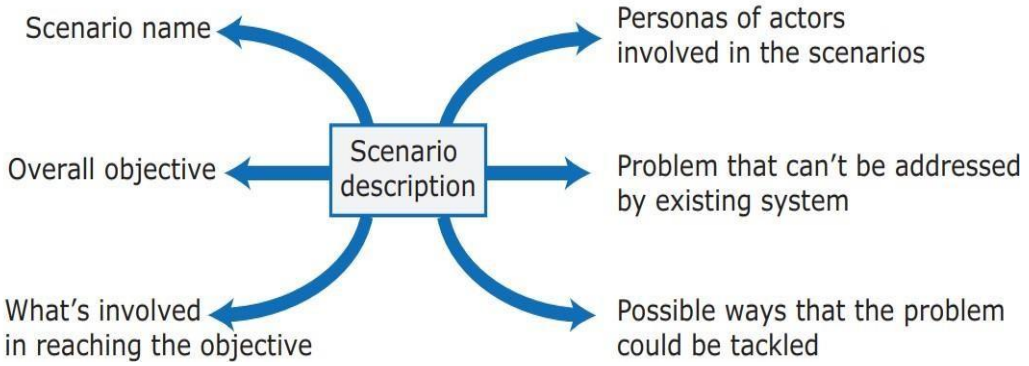
Figure 3.4 Persona descriptions



SCENARIOS

- A scenario is a narration that describes a situation in which a user is using your product's features to do something that they want to do.
- Scenarios are used in the design of requirements and system features, in system testing, and in user interface design
- It should briefly explain the user's problem and present an imagined way thatthe problem might be solved.
- Scenarios are high-level stories of system use.
- They should describe a sequence of interactions with the system but shouldnot include details of these interactions.
- They are the basis for both use cases, which are extensively used in object- oriented methods, and user stories, which are used in agile methods.

Figure 3.5 Elements of a scenario description



- Narrative, high-level scenarios, are primarily a means of facilitating communication and stimulating design creativity.
- They 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.

Structured scenarios should include different fields such as:

- ✓ what the user sees at the beginning of a scenario,
- ✓ a description of the normal flow of events,
- ✓ a description of what might go wrong, and so on.

At the early stages of product design, the scenarios be narrative rather than structured.

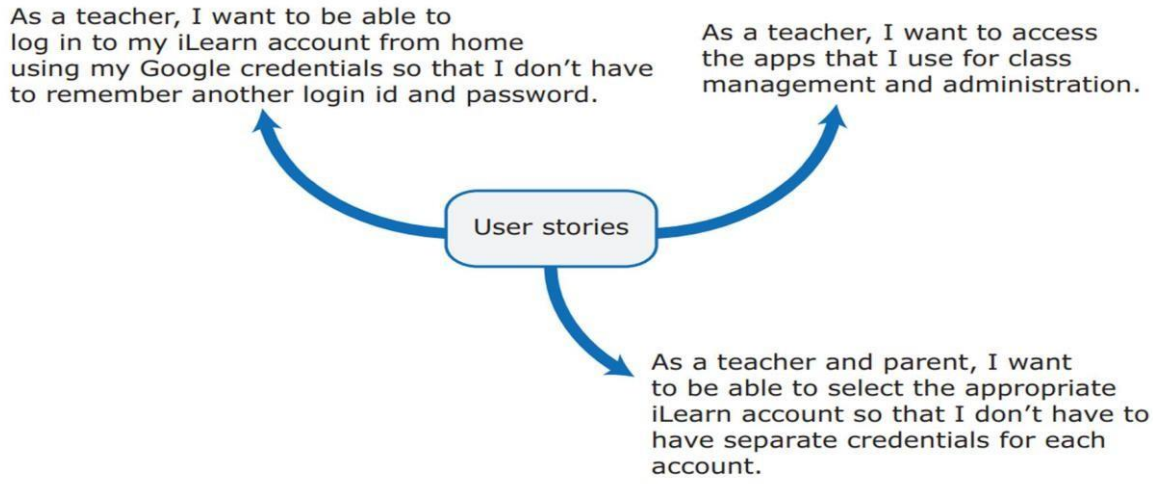
Writing scenarios

- ✓ Start with the personas that you have created.
- ✓ Try to imagine several scenarios for each persona.
- ✓ Not necessary to include every details you think users might do with your product.
- ✓ Scenarios should always be written from the user's perspective and should be based on identified personas or real users.
- ✓ Scenario writing is not a systematic process and different teams approach it in different ways.
- ✓ Writing scenarios always gives you ideas for the features that you can include in the system.

User Stories

- ✓ These 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.
- ✓ User stories are not intended for planning but for helping with feature identification.
- ✓ Aim to develop stories that are helpful in one of 2 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.

Figure 3.6 User stories from Emma's scenario



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 ofthe system.
- ✓ Feature is a fragment of functionality that implements some user or system need. We can access features through user interface of a product.
- ✓ Feature is something that the user needs or wants.

- **Identify the product features that are independent, coherent and relevant:**
- **Independence** ⟶ A feature should not depend on how other system features implemented and should not be affected by the order of activation of other features.
- **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.
- **Relevance System features** ⟶ should reflect the way users normally carry out some task. They should not offer obscure functionality that is rarely required.

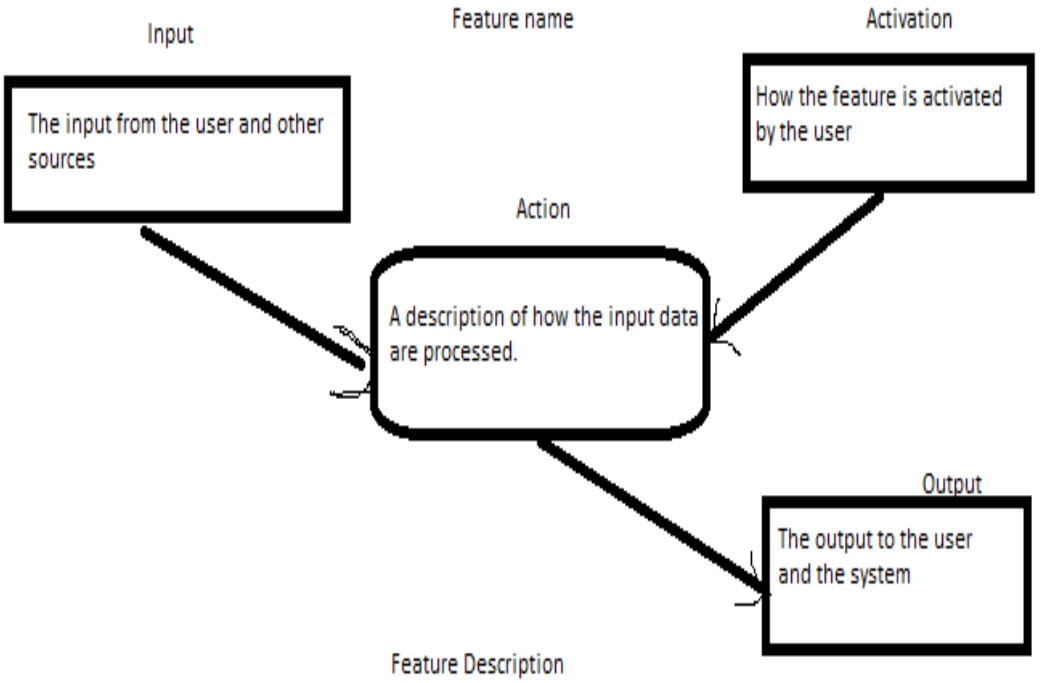


Figure 3.8 Feature design

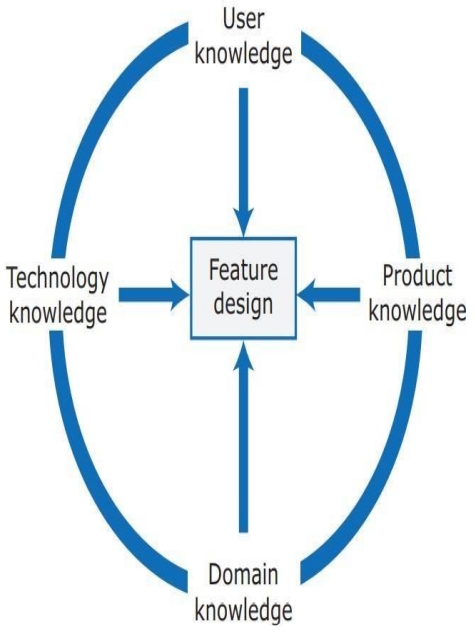
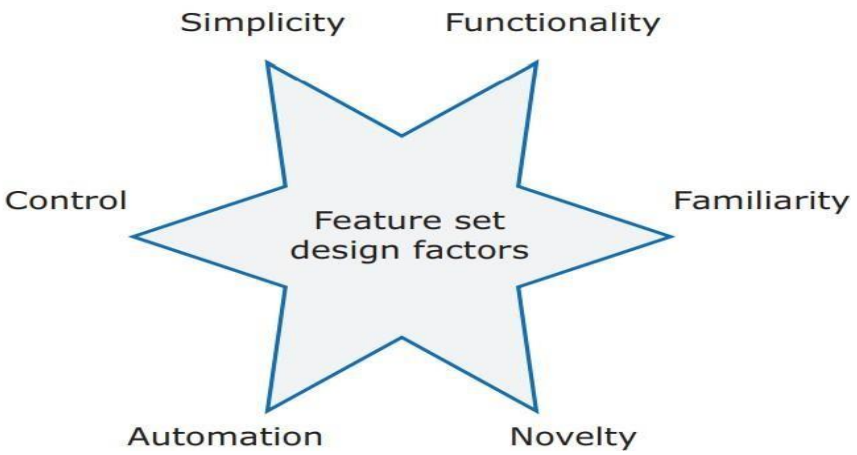


Table 3.8 Knowledge required for feature design

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.

Figure 3.9 Factors in feature set design

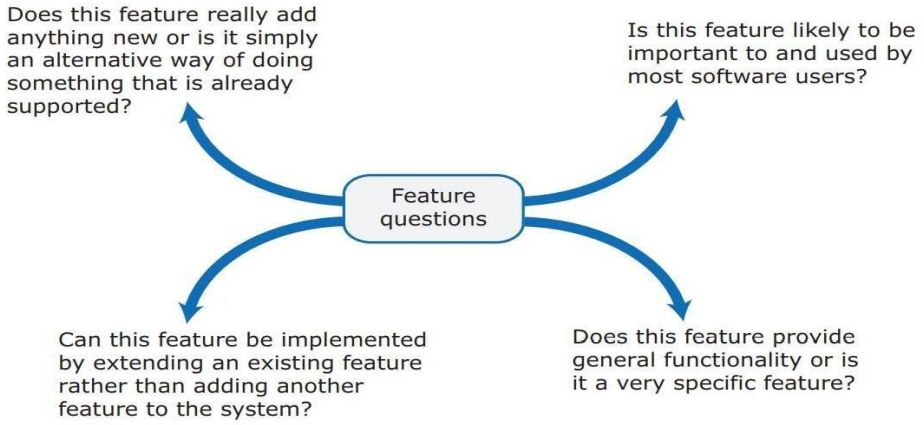


- One problem that product developers should be aware of and try to avoid is "**feature creep.**"
- Feature creep the number of features in a product creeps potential uses of the product are envisaged.
- It adds to the complexity of a product, which means that you are likely to introduce bugs and security vulnerabilities into the software.
- It also usually makes the user interface more complex.

Feature creep happens for 3 reasons:

- Product managers and marketing executives discuss the functionality they need with a range of different product users. Different users have slightly different needs or may do the same thing but in slightly different ways.
- Competitive products are introduced with slightly different functionality to your product. There is marketing pressure to include comparable functionality so that market share is not lost to these competitors. Thiscan lead to "feature wars," where competing products become more and more bloated as they replicate the features of their competitors.
- The product tries to support both experienced and inexperienced users. Easy ways of implementing common actions are added for inexperienced users and the more complex features to accomplish the same thing are retained because experienced users prefer to work that way.
- Toavoid feature creep, the product manager and the development team should review all feature proposals and compare new proposals to features that have already been accepted for implementation.

Figure 3.10 Avoiding feature creep



Feature identification should be a team activity, and as features are identified, the team should discuss them and generate ideas about related features.

- Collaborative writing
- Blogs and web pages

Feature List

The output of the feature identification process should be a list of features that you use for designing and implementing your product.

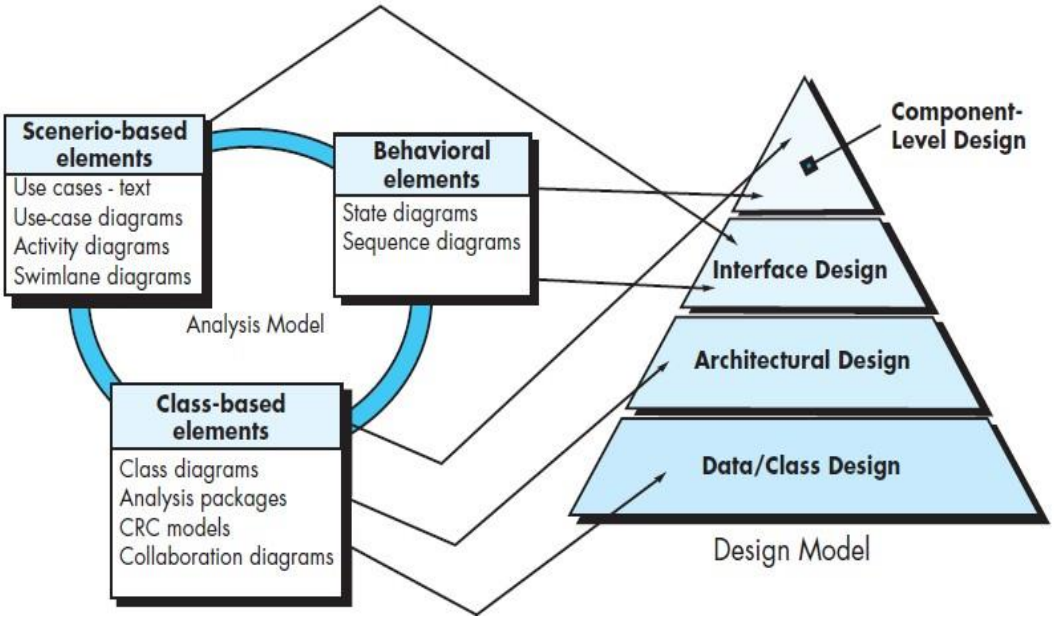
- Add detail when you are implementing the feature.
- You can describe a feature from one or more user stories.
- Scenarios and user stories should always be your starting point for identifying product features.

Design concepts - Design within the context of software engineering, Design Process, Design concepts, Design Model.

Software design encompasses the set of principles, concepts, and practices that lead to the development of a high-quality system or product. It is the place where creativity rules—where stakeholder requirements, business needs, and technical considerations all come together in the formulation of a product or system. Design creates a representation or model of the software, the design model provides detail about software architecture, data structures, interfaces, and components that are necessary to implement the system.

DESIGN WITHIN THE CONTEXT OF SOFTWARE ENGINEERING

Software design sits at the technical kernel of software engineering. Beginning once software requirements have been analyzed and modeled, software design is the last translating the requirements model into the design model .



Each of the elements of the requirements model provides information that is necessary to create the four design models required for a complete specification of design. The flow of information during software design is illustrated. The requirements model, manifested by scenario-based, class-based, and behavioral elements, feed the design task.

The data/class design transforms class models into design class realizations and the requisite data structures required to implement the software. The objects and provide the basis for the data design activity.

The architectural design defines the relationship between major structural elements of the software, the architectural styles and patterns. The architectural design representation—the framework of a computer-based system—is derived from the requirements model.

The interface design describes how the software communicates with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior. Therefore, usage scenarios and behavioral models provide much of the information required for interface design.

The **component-level design** transforms structural elements of the software architecture into a procedural description of software components. Information obtained from the class-based models and behavioral models serve as the basis for component design.

The importance of software design can be stated with a single word— *quality*. Design is the only way that you can accurately translate stakeholder's requirements into a finished software product or system. Software design serves as the foundation for all the software engineering and software support activities that follow.

THE DESIGN PROCESS

Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software. Initially, the blueprint depicts a holistic view of software.

Quality Guidelines.

Consider the following 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 (these are discussed later in this chapter), 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.

Assessing Design Quality—the Technical Review

During design, quality is assessed by conducting a series of technical reviews (TRs). A technical review is a meeting conducted by members of the software team. Usually two, three, or four people participate depending on the scope of the design information to be reviewed.

Quality Attributes. The FURPS quality attributes represent a target for all software design:

- ***Functionality i***s 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 ability to organize and control elements of the software configuration),the ease with which a system can be installed, and the ease with which problems can be localized.

Common characteristics:
(1) A mechanism for the translation of the requirements model into a design representation,
(2) A notation for representing functional components and their interfaces,
(3) Heuristics for refinement and partitioning
(4) Guidelines for quality assessment.

DESIGN CONCEPTS

Abstraction

- 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. The name of a procedural abstraction implies these functions, but specific details are suppressed.

  An example of a procedural abstraction would be the word *open* for a door. *Open* implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.).
- A ***data abstraction***is a named collection of data that describes a data object. In the context of the procedural abstraction *open,* we can define a data abstraction called **door.** Like any data object, the data abstraction for **door** would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions).

Architecture

Software architecture alludes to "the overall structure of the software and the ways in which that structure provides conceptual integrity for a system". 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. A set of architectural patterns enables a software engineer to reuse design-level concepts.

Shaw and Garlan describe a 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, filters) 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.
- ***Families of related systems*** **"**draw upon repeatable patterns that are commonly encountered in the design of families of similar systems."

Given the specification of these properties, the 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.
- **F*unctional models*** can be used to represent the functional hierarchy of a system.

Patterns

- *"A* pattern is a named nugget of insight which conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns".
- 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
  (1) Whether the pattern is applicable to the current work,
  (2) Whether the pattern can be reused (hence, saving design time), and
  (3) Whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

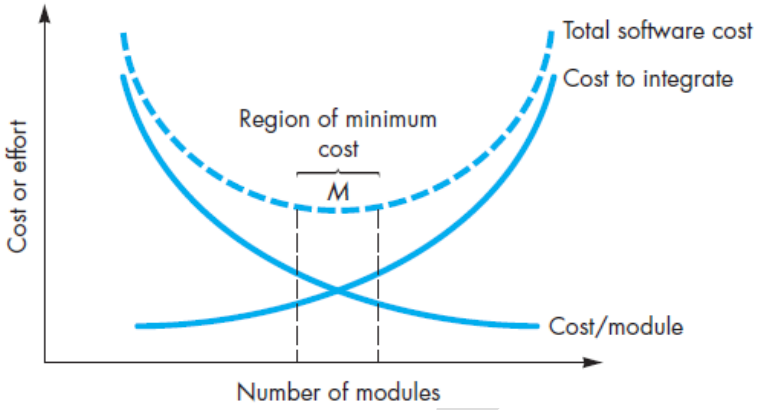
Separation of Concerns

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.

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.
- "Modularity is the single attribute of software that allows a program to be intellectually manageable".

- Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible. In the Figure, the effort (cost) to develop an individual software module does decrease as the total number of modules increases.



- Given the same set of requirements, more modules means smaller individual size. However, 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, but we do not have the necessary sophistication to predict *M* with assurance.

Information Hiding

- The principle of *information hiding* suggests that modules be "characterized by design decisions that (each) hides from all others."
- Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function.

Functional Independence

- Functional independence is achieved by developing modules with "single minded" function and an "aversion" to excessive interaction with other modules.
- Functional independence is a key to good design, and design is the key to software quality.
- 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.
- A **cohesive** module performs a single task, requiring little interaction with other components in other parts of a program
- **Coupling** is an indication of interconnection among modules in a software structure.
- Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.
- High cohesion and low coupling make the module to be effectively design.

Refinement

- ***Stepwise refinement*** is a top-down design strategy.
- An application is developed by successively refining levels of procedural detail.

- A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached.
- Refinement is actually **a process of *elaboration***. You begin with a statement of function (or description of information) that is defined at a high level of abstraction.
- **Abstraction and refinement are complementary concepts.**
- Abstraction enables you to specify procedure and data internally but suppress the need for "outsiders" to have knowledge of low-level details.
- Refinement helps you to reveal low-level details as design progresses.
- Both concepts allow you to create a complete design model as the design evolves.

Aspects

An aspect is implemented as a separate module (component) rather than as software fragments that are "scattered" or "tangled" throughout many components.

Refactoring

- R*efactoring* is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior.
- "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.

Object-Oriented Design Concepts

The object-oriented (OO) paradigm is widely used in modern software engineering. OO design concepts such as classes and objects, inheritance, messages, and polymorphism

Design Classes

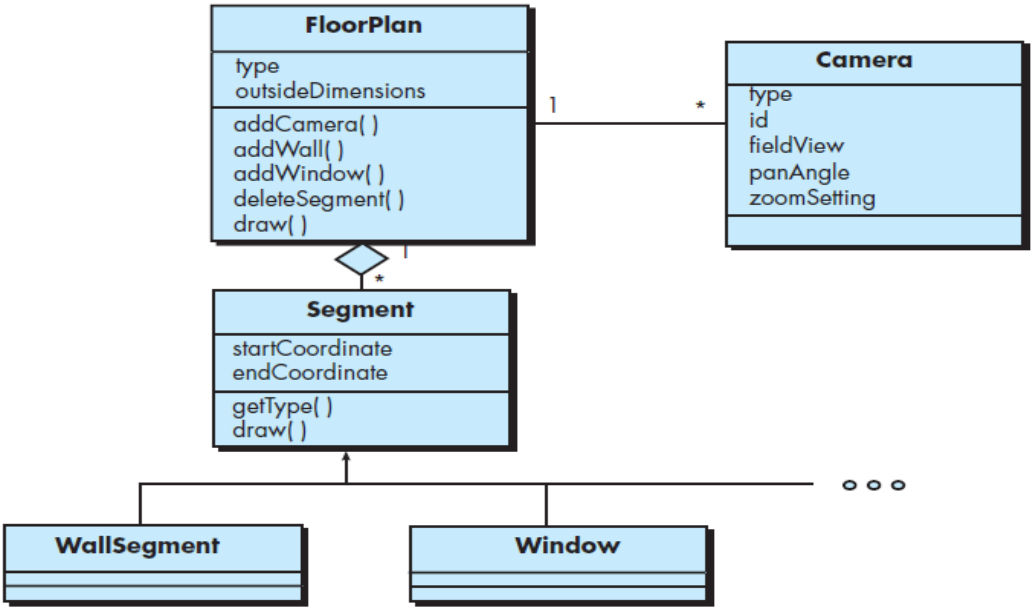
The analysis model defines a set of analysis classes . Five different types of design classes, each representing a different layer of the design architecture, can be developed

- ***User interface classes*** define all abstractions that are necessary for human-computer interaction (HCI) and often implement the HCI in the context of a metaphor.
- ***Business domain classes*** identify the attributes and services (methods) that are required to implement some element of the business domain that was defined by one or more analysis classes.
- ***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.

High cohesion. A cohesive design class has a small, focused set of responsibilities and single-mindedly applies attributes and methods to implement those responsibilities.

Low coupling. Within the design model, it is necessary for design classes to collaborate with one another. However, collaboration should be kept to an acceptable minimum. If a design model is highly coupled (all design classes collaborate with all other design classes), the system is difficult to implement, to test, and to maintain over time. In general, design classes within a subsystem

should have only limited knowledge of other classes. This restriction, called the *Law of Demeter* suggests that a method should only send messages to methods in neighboring classes.



Design class for Floor Plan and composite aggregation for the class

Dependency Inversion

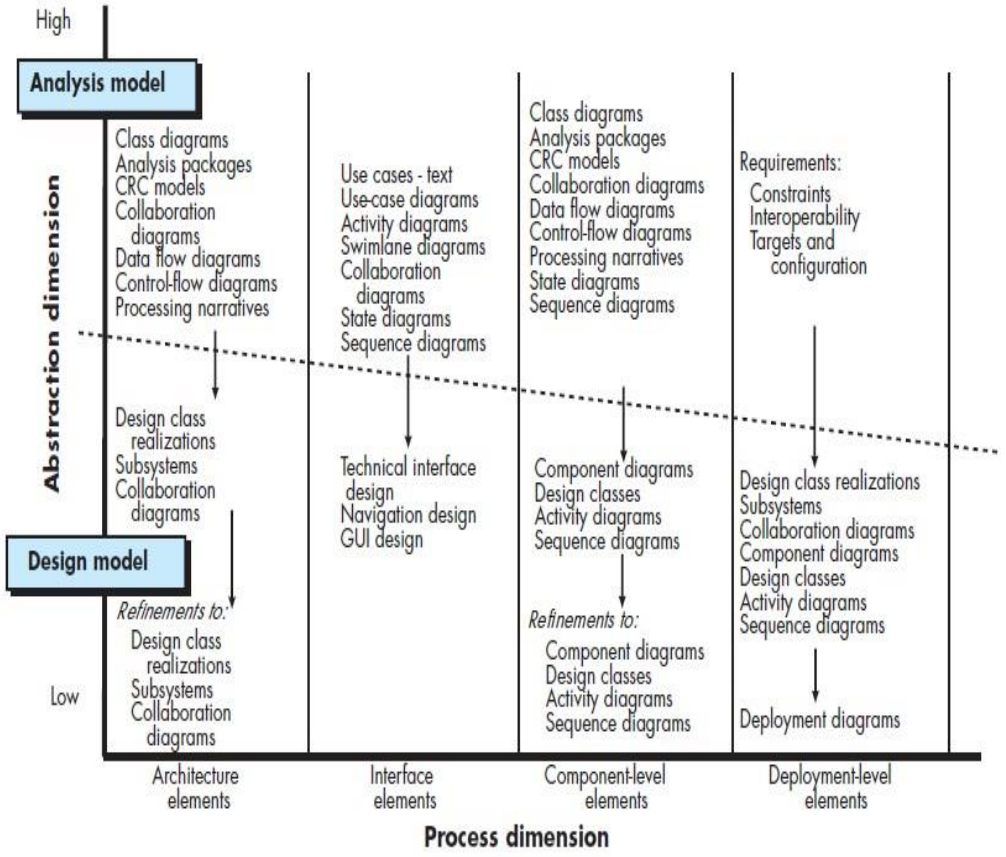
Dependency inversion principle which states: *High-level modules(classes) should not depend [directly] upon low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.*

Design for Test

There is an ongoing debate about whether software design or test case design should come first. Test-driven development (TDD) write tests before implementing any other code. They take to heart Tom Peters' credo, "Test fast, fail fast, and adjust fast." Testing guides their design as they implement in short, rapid-fi re "write test code—fail the test—write enough code to pass—then pass the test" cycles.

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. The dashed line indicates the boundary between the analysis and design models.
- However, that model elements indicated along the horizontal axis are not always developed in a sequential fashion. In most cases preliminary architectural design sets the stage and is followed by interface design and component-level design, which often occur in parallel. The deployment model is usually delayed until the design has been fully developed.



1 . Data Design Elements

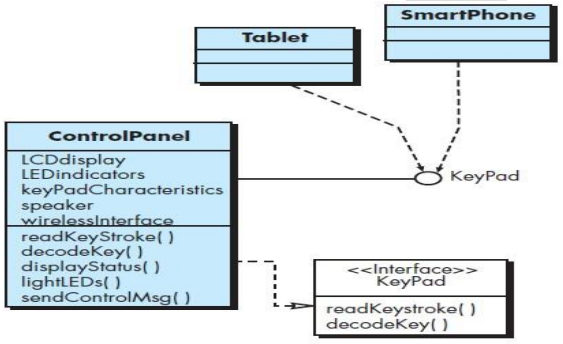
- Data design (sometimes referred to as *data architecting*) creates a model of data and/or information that is represented at a high level of abstraction (the customer/user's view of data). This data model is then refined into progressively more implementation-specific representations that can be processed 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.

2. Architectural Design Elements

The *architectural design* for software is the equivalent to the floor plan of a house. The floor plan depicts the overall layout of the rooms; their size, shape, and relationship to one another; and the doors and windows that allow movement into and out of the rooms. The floor plan gives us an overall view of the house. Architectural design elements give us an overall view of the software.
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.

3. Interface Design Elements

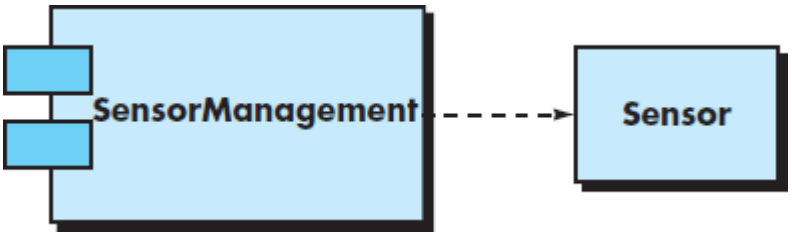
- The interface design for software is analogous to a set of detailed drawings (and specifications) for the doors, windows, and external utilities of a house. In essence, the detailed drawings (and specifications) for the doors, windows, and external utilities tell us how things and information flow into and out of the house and within the rooms that are part of the floor plan.
- The interface design elements for software depict information flows into and out of a system and how it is communicated among the components defined as part of the architecture.
- 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.



Interface representation For ControlPanel

4. Component-Level Design Elements

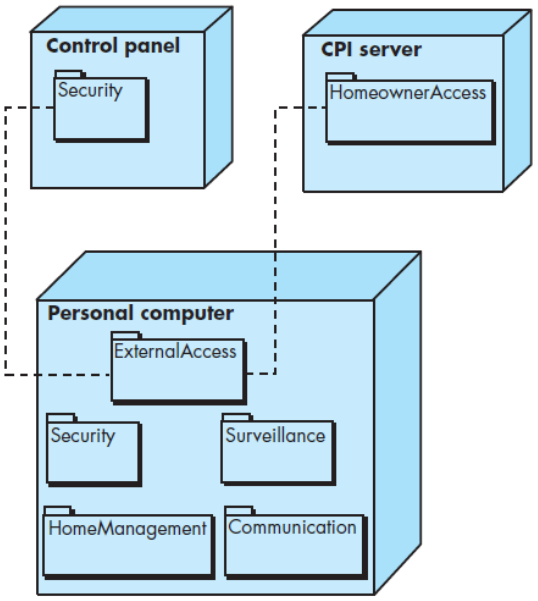
- The component-level design for software is the equivalent to a set of detailed drawings (and specifications) for each room in a house. These drawings depict wiring and plumbing within each room, the location of electrical receptacles and wall switches, faucets, sinks, showers, tubs, drains, cabinets, and closets, and every other detail associated with a room.
- 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).



A UML component diagram

5. 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.
- For example, the elements of the *SafeHome* product are configured to operate within three primary computing environments—a homebasedPC, the *SafeHome* control panel, and a server housed at CPI Corp. (providing Internet-based access to the system). In addition, limited functionality may be provided with mobile platforms.
- During design, a UML deployment diagram is developed and then refined as shown in Figure. In the figure, three computing environments are shown (in actuality, there would be more including sensors, cameras, and functionality delivered by mobile platforms). The subsystems (functionality) housed within each computing element are indicated. For example, the personal computer houses subsystems that implement security, surveillance, home management, and communications features.
- In addition, an external access subsystem has been designed to manage all attempts to access the *SafeHome* system from an external source. Each subsystem would be elaborated to indicate the components that it implements.
- The diagram shown in Figure is in *descriptor form.* This means that the deployment diagram shows the computing environment but does not explicitly indicate configuration details. For example, the "personal computer" is not further identified. It could be a Mac, a Windows-based PC, a Linux-box or a mobile platform with its associated operating system. These details are provided when the deployment diagram is revisited in *instance form* during the latter stages of design or as construction begins. Each instance of the deployment (a specific named hardware configuration) is identified.



A UML deployment diagram

ARCHITECTURAL DESIGN

Architectural Design - Software Architecture, Architectural Styles, Architectural considerations, 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.
An architecture model encompassing data architecture and program structure is created during architectural design.

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.

Identify 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"

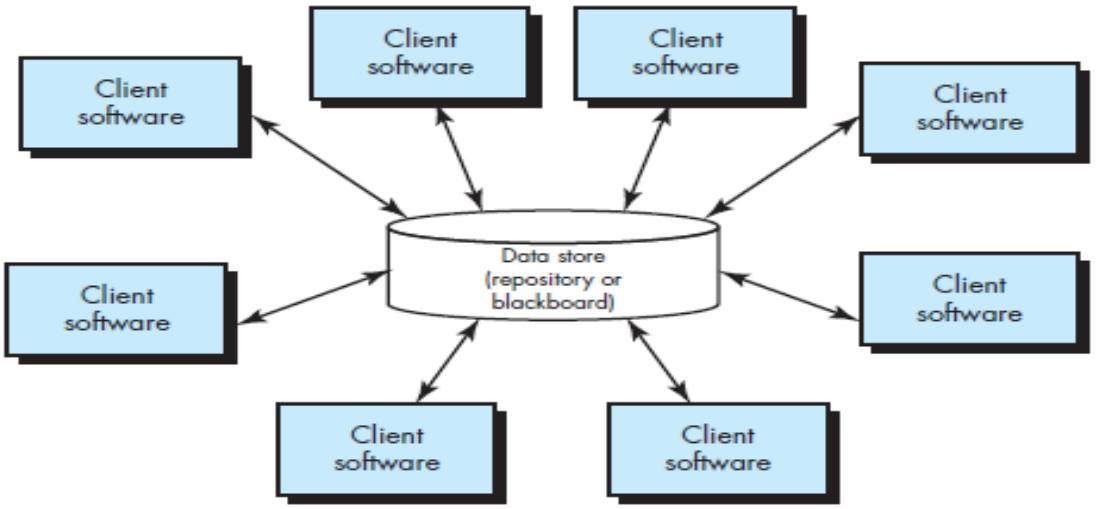
ARCHITECTURAL STYLES

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. In the case where an existing architecture is to be reengineered, the imposition of an architectural style will result in fundamental changes to the structure of the software including a reassignment of the functionality of components.

Different Architectural Styles

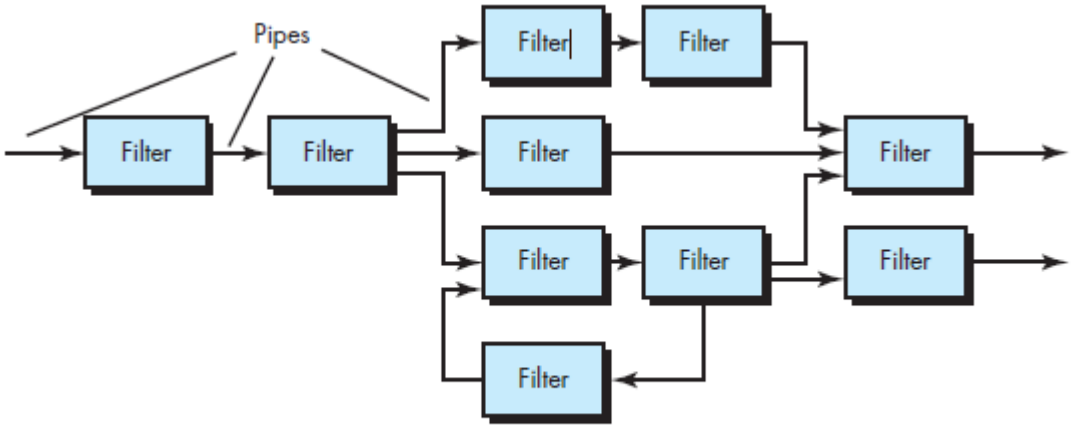
- **Data-Centered Architecture:** A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. Figure illustrates a typical data-centered style. Client software accesses 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). In addition, 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-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. However, the filter does not require knowledge of the workings of its neighboring filters.

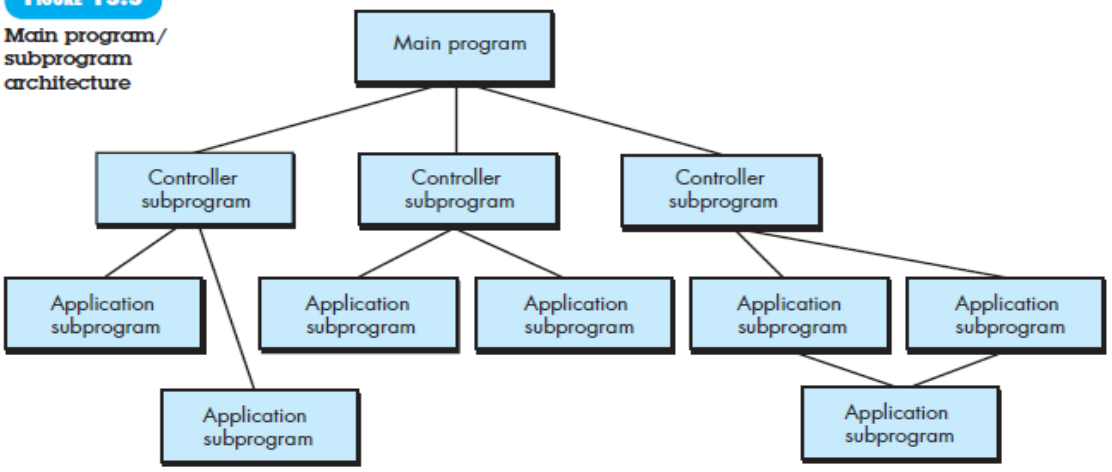
If the data flow degenerates into a single line of transforms, it is termed ***batch sequential***. This structure accepts a batch of data and then applies a series of sequential components (filters) to transform it.



Pipes and filters

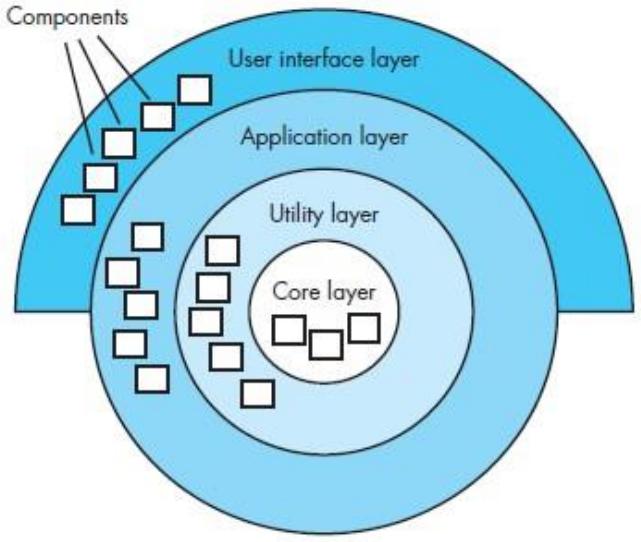
- **Main program/subprogram architecture**

FIGURE 13.3
Main program/ subprogram architecture



- **Call and Return Architectures:** This architectural style enables to achieve a program structure that is relatively easy to modify and scale. A number of sub styles 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. Figure above illustrates an architecture of this type.
  - ***Remote procedure call architectures***. The components of a main program/ subprogram architecture are distributed across multiple computers on a network.

FIGURE 13.4
Layered architecture



- **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:**

  - The basic structure of a layered architecture is illustrated. 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.
  - Once requirements engineering uncovers the characteristics and constraints of the system to be built, the architectural style and/or combination of patterns that best fits those characteristics and constraints can be chosen. In many cases, more than one pattern might be appropriate and alternative architectural styles can be designed and evaluated. For example, a layered style (appropriate for most systems) can be combined with a data-centered architecture in many database applications.

ARCHITECTURAL CONSIDERATIONS

Buschmann and Henny suggest several architectural considerations that can provide software engineers with guidance as architecture decisions are made:

- **Economy** —many software architectures suffer from unnecessary complexity driven by the inclusion of unnecessary features or nonfunctional requirements (e.g., reusability when it serves no purpose). 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. As an example of architectural symmetry, consider a *customer account* object whose life cycle is modeled directly by a software architecture that requires both *open ()* and *close()* methods. Architectural symmetry can be both structural and behavioral.
- **Emergence** —Emergent, self-organized behavior and control are often the key to creating scalable, efficient, and economic software architectures. For example, many real-time software applications are event driven. The sequence and duration of the events that define the system's behavior is an emergent quality. It is very difficult to plan for every possible sequence of events. Instead the system architect should create a flexible system that accommodates this emergent behavior.

ARCHITECTURAL DESIGN

As architectural design begins, context must be established. To accomplish this, the external entities (e.g., other systems, devices, and 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 to 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.

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. The generic structure of the architectural context diagram is illustrated.

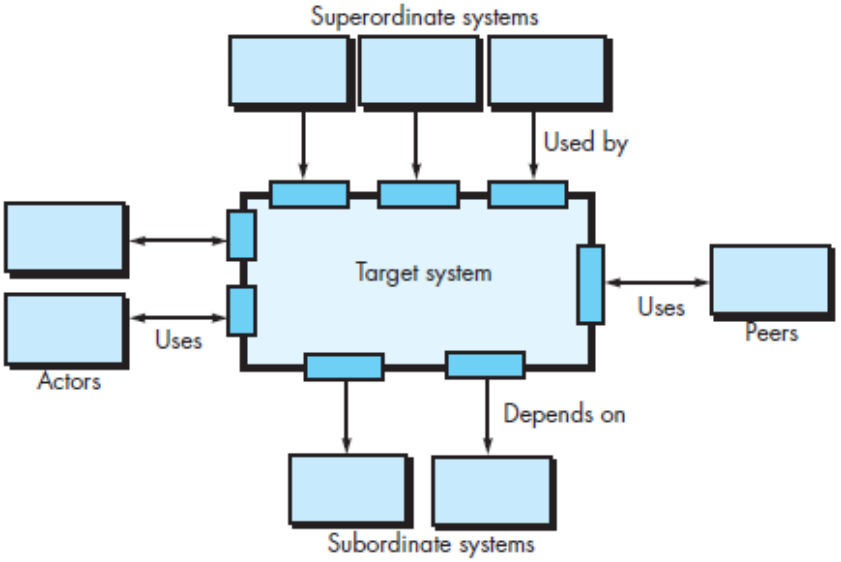


FIGURE 13.5 Architectural context diagram
Source: Adapted from [Bos00].

Referring to the figure, 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).

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. In general, 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. Continuing the discussion of the *Safe Home* security function, you 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.

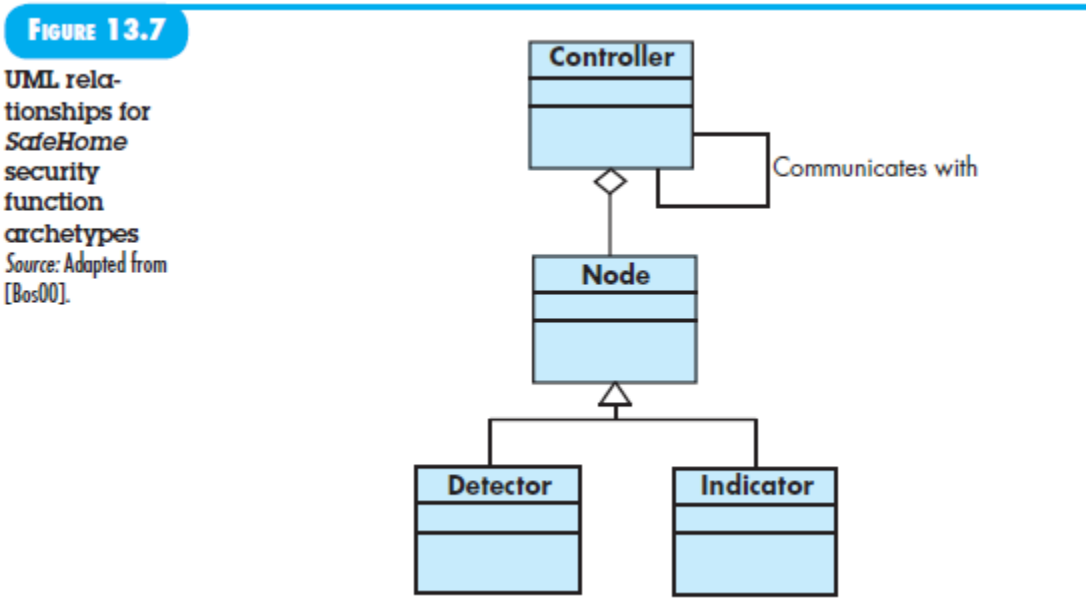


FIGURE 13.7 UML relationships for *SafeHome* security function archetypes
Source: Adapted from [Bos00].

Each of these archetypes is depicted using UML notation as shown in Figure .**Detector** might be refined into a class hierarchy of sensors.

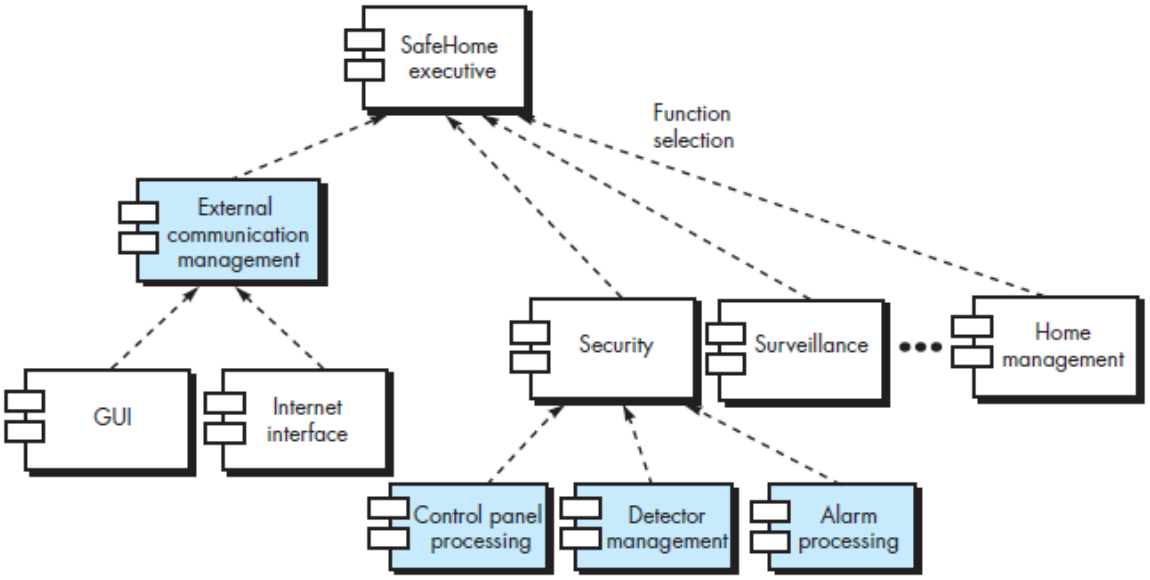
Refining the Architecture into Components

- As the software architecture is refined into components, the structure of the system begins to emerge. These analysis classes represent entities within the application (business) domain that must be addressed within the software architecture. Hence, the application domain is one source for the derivation and refinement of components. Another source is the infrastructure domain.
- The architecture must accommodate many infrastructure components that enable application components but have no business connection to the application domain.

- For example, memory management components, communication components, database components, and task management components are often integrated into the software architecture.
- The interfaces depicted in the architecture context diagram imply one or more specialized components that process the data that flows across the interface. In some cases (e.g., a graphical user interface), a complete subsystem architecture with many components must be designed.
- Continuing the *Safe Home S*ecurity function example, you might 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.

The overall architectural structure (represented as a UML component diagram) is illustrated in Figure. Transactions are acquired by *external communication management* as they move in from components that process the *SafeHome* GUI and the Internet interface. This information is managed by a *SafeHome* executive component that selects the appropriate product function (in this case security). The *control panel processing* component interacts with the homeowner to arm/disarm the security function. The *detector management* component polls sensors to detect an alarm condition, and the *alarm processing* component produces output when an alarm is detected.

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



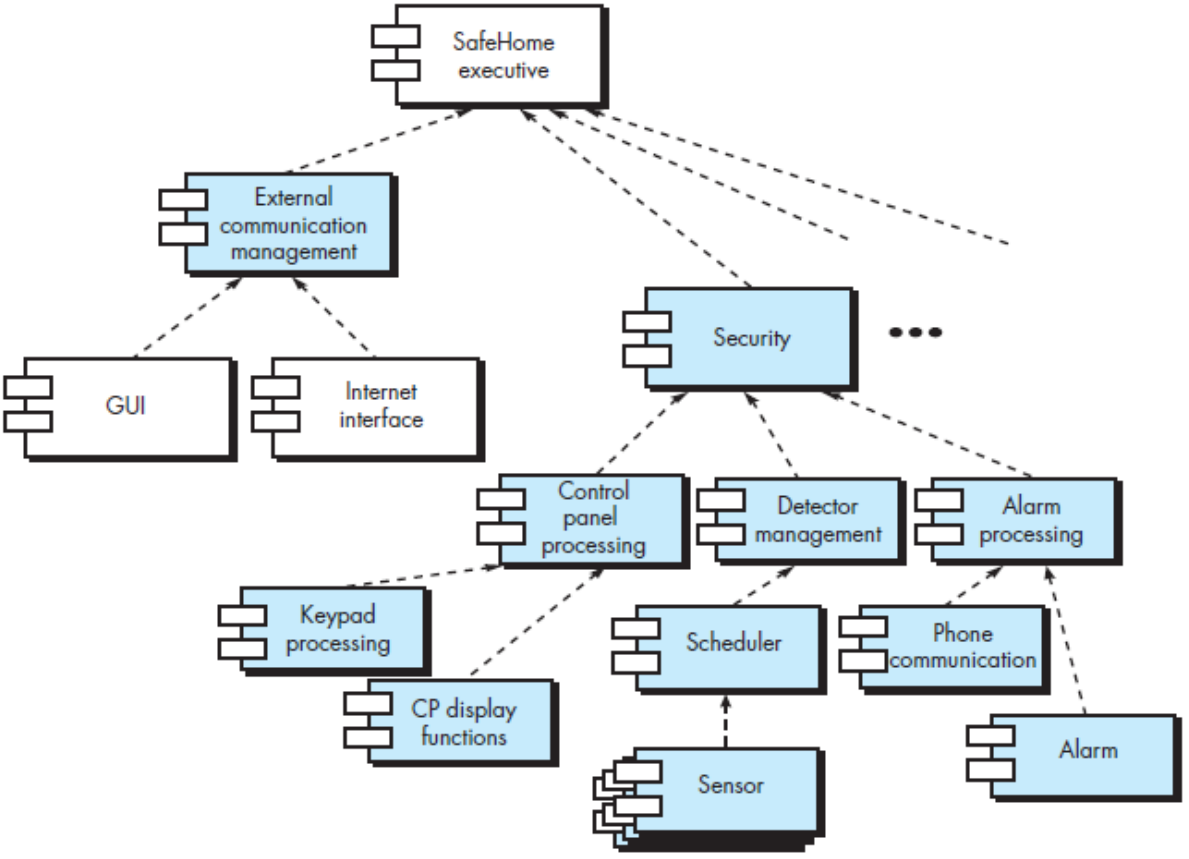
Describing Instantiations of the System

- The architectural design that has been modeled to this point is still relatively high level. The context of the system has been represented; archetypes that indicate the important abstractions within the problem domain have been defined, the overall structure of the system is apparent, and the major software components have been identified. However, further refinement is still necessary.

- To accomplish this, an actual instantiation of the architecture is developed. By this we mean that the architecture is applied to a specific problem with the intent of demonstrating that the structure and components are appropriate.

Figure illustrates an instantiation of the *SafeHome* architecture for the security system. Components shown in Figure above are elaborated to show additional detail. For example, the *detector management* component interacts with a *scheduler* infrastructure component that implements polling of each *sensor* object used by the security system. Similar elaboration is performed for each of the components represented in Figure

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. The architectural components (Web pages) of a WebApp are designed to allow control to be passed to other system components, allowing very flexible navigation structures. The physical location of media and other content resources also influences the architectural choices made by software engineers.

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.
- A number of 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., smart phones, 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

Component

A component is a modular building block for computer software. The OMG Unified Modeling Language Specification defines a component as "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, and people) that exist outside the boundaries of the software.

The design for each component, represented in graphical, tabular, or text-based notation, is the primary work product produced during component-level design.

Three important views of what a component is and how it is used as design modeling proceeds.

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. To accomplish this, we begin with the analysis model and elaborate analysis classes(for components that relate to the problem domain) and infrastructure classes (for components (for components that provide support services for the problem domain).

The Traditional View

A traditional component 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.

A Process-Related View

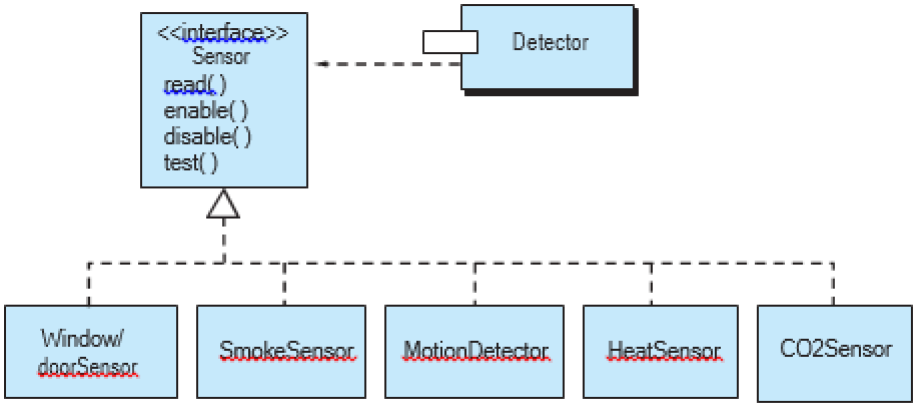
Over the past three decades, the software engineering community has emphasized the need to build systems that make use of existing software components or design patterns. A catalog of proven design or code-level components is made available to you as design work proceeds. As the software architecture is developed, we 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.

DESIGN CLASS BASED COMPONENTS

Basic design principles

The Open-Closed Principle (OCP). "A module [component] should be open for extension but closed for modification"

- Should 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, you create abstractions that serve as a buffer between the functionality that is likely to be extended and the design class itself.
- 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.



The Liskov Substitution Principle (LSP).

- "*Subclasses should be substitutable for their base classes".*
- This design principle suggests that a component that uses a base class should continue to function properly if a class derived from the base class is passed to the component instead
- In the context, a "contract" is a *pre-condition* that must be true before the component uses a base class and a *post-condition* that should be true after the component uses a base class. When you create derived classes, be sure they conform to the pre- and post-conditions.

Dependency Inversion Principle (DIP).

- "*Depend on abstractions. Do not dependon concretions".*
- Abstractions are the place where a design can be extended without great complication. The more a component depends on other concrete components (rather than on abstractions such as an interface), the more difficult it will be to extend.

The Interface Segregation Principle (ISP).

- *"Many client-specific interfaces are better than one general purpose interface".*
- There are many instances in which multiple client components use the operations provided by a server class.
- Should create a specialized interface to serve each major category of clients. Only those operations that are relevant to a particular category of clients should be specified in the interface for that client. If multiple clients require the same operations, it should be specified in each of specialized interfaces.

The Release Reuse Equivalency Principle (REP).

- *"The granule of reuse is the granule of release"*
- When classes or components are designed for reuse, an implicit contract is established between the developer of the reusable entity and the people who will use it.
- The developer commits to establish a release control system that supports and maintains older versions of the entity while theusers slowly upgrade to the most current version.

The Common Closure Principle (CCP).

- *"Classes that change together belong together."*
- Classes should be packaged cohesively.
- That is, when classes are packaged as part of a design, they should address the same functional or behavioural area.
- When some characteristic of that area must change, it is likely that only those classes within the package will require modification. This leads tomore effective change control and release management

The Common Reuse Principle (CRP).

- *"Classes that aren't reused together should not be grouped together"*
- When one or more classes with a package changes, the release number of the package changes.
- All other classes or packages that rely on the package that has been changed must now update to the most recent release of the package and be tested to ensure that the new release operated without incident.
- If classes are not grouped cohesively, it is possible that a class with no relationship to other classes within a package is changed. This will precipitate unnecessary integration and testing.
- For this reason, only classes that are reused together should be included within a package.

Component-Level Design Guidelines

These guidelinesapply to components, their interfaces, and the dependencies and inheritance characteristics that have an impact on the resultant design.

Suggests the following guidelines:

Components. Naming conventions should be established for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model. Architectural component names should be drawn from the problem domain and should have meaning to all stakeholders who view the architectural model. For example, the class

We can choose to use stereotypes to help identify the nature of components at the detailed design level. For example, <<infrastructure>> might be used to identify an infrastructure component, <<database>> could be used to identify a database that services one or more design classes or the entire system; <<table>> can be used to identify a table within a database.

Interfaces. Interfaces provide important information about communication and collaboration (as well as helping us to achieve the OPC).

Dependencies and Inheritance. For improved readability, it is a good idea to model dependencies from left to right and inheritance from bottom (derived classes) to top (base classes).

Cohesion

Implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself.

Functional. Exhibited primarily by operations, this level of cohesion occurs when a module performs one and only one computation and then returns aresult.

Layer. Exhibited by packages, components, and classes, this type of cohesion occurs when a higher layer accesses the services of a lower layer, but lower layers do not access higher layers.

Communicational. All operations that access the same data are defined within one class. In general, such classes focus solely on the data in question, accessing and storing it.

Classes and components that exhibit **functional, layer, and communicational cohesion are relatively easy to implement, test, and maintain.**

Coupling

- As the amount of communication and collaboration increases (i.e., as the degree of "connectedness" between classes increases), the complexity of the system also increases. And as complexity increases, the difficulty of implementing, testing, and maintaining software grows.
- ***Coupling*** is a qualitative measure of the degree to which classes are connected to one another. As classes (and components) become more interdependent, coupling increases. An important objective in component-level design is tokeep coupling as low as is possible.
- ***Content coupling*** occurs when one component "surreptitiously modifies data that is internal to another component" .This violates information hiding—a basic designconcept.
- ***Control coupling*** occurs when operation *A()* invokes operation *B()* and passes a control flag to *B*. The control flag then "directs" logical flow within *B*. The problem with this form of coupling is that an unrelated change in *B* can result in the necessity to change the meaning of the control flag that *A* passes. If this is overlooked, an error will result.
- ***External coupling*** occurs when a component communicates or collaborates with infrastructure components (e.g., operating system functions, database capability, tele-communication functions). Although this type of coupling is necessary, it should be limited to a small number of components or classes within a system.

- Software must communicate internally and externally. Therefore, coupling is a fact of life. However, the designer should work to reduce coupling whenever possible

CONDUCTING COMPONENT LEVEL DESIGN

The following steps represent a typical task set for component-level design, when it is applied for an object-oriented system.

Step 1. Identify all design classes that correspond to the problem domain. Using the requirements and architectural model, each analysis class and architectural component is elaborated

Step 2. Identify all design classes that correspond to the infrastructure domain. These classes are not described in the requirements model and are often missing from the architecture model, but they must be described at this point. Classes and components in this category include GUI components (often available as reusable

components), operating system components, and object and data management components.

Step 3. Elaborate all design classes that are not acquired as reusable components. Elaboration requires that all interfaces, attributes, and operations necessary to implement the class be described in detail. Design heuristics (e.g., component cohesion and coupling) must be considered as this task is conducted.

Step 3a. Specify message details when classes or components collaborate. The requirements model makes use of a collaboration diagram to show how analysis classes collaborate with one another. Messages that are passed between objects within a system.

Step 3b. Identify appropriate interfaces for each component. Within the context of component-level design, a UML interface is "a group of externally visible (i.e., public) operations. The interface contains no internal structure, it has no attributes, no associations. ".

Step 3c. Elaborate attributes and define data types and data structures required to implement them. In general, data structures and types used to define attributes are defined within the context of the programming language that is to be used for implementation.

Step 3d. Describe processing flow within each operation in detail. This may be accomplished using a programming language-based pseudo code or with a UML activity diagram. Each software component is elaborated through a number of iterations that apply the stepwise refinement concept.

The first iteration defines each operation as part of the design class. In every case, the operation should be characterized in a way that ensures high cohesion; that is, the operation should perform a single targeted function or sub function. The next iteration does little more than expand the operation name.

Step 4. Describe persistent data sources (databases and files) and identify the classes required to manage them. Databases and files normally transcend the design description of an individual component. In most cases, these persistent data stores are initially specified as part of architectural design. However, as design elaboration proceeds, it is often useful to provide additional detail about the structure and organization of these persistent data sources.

Step 5. Develop and elaborate behavioural representations for a class or component. UML state diagrams were used as part of the requirements model to represent the externally observable behaviour of the system and the more localized behaviour of individual analysis classes. During component-level design, it is sometimes necessary to model the behaviour of a design class.

The dynamic behavior of an object (an instantiation of a design class as the program executes) is affected by events that are external to it and the current state (mode of behavior) of the object. To understand the dynamic behavior of an object, you should examine all use cases that are relevant to the design class throughout its life.

Step 6. Elaborate deployment diagrams to provide additional implementation detail. Deployment diagrams are used as part of architectural design and are represented in descriptor form. In this form, major system functions (often

represented as subsystems) are represented within the context of the computing environment that will house them.

During component-level design, deployment diagrams can be elaborated to represent the location of key packages of components. However, components generally are not represented individually within a component diagram. In some cases, deployment diagrams are elaborated into instance form at this time. This means that the specific hard- ware and operating system environment(s) that will be used is (are) specified and the location of component packages within this environment is indicated.

Step 7. Refactor every component-level design representation and always con- sider alternatives. Design is an iterative process. The first component-level model we create will not be as complete, consistent, or accurate as the *n*th iteration you apply to the model. It is essential to refactor as design work is conducted.

COMPONENT LEVEL DESIGN FOR WEB APPLICATIONS

WebApp component is (1) a well-defined cohesive function that manipulates content or provides computational or data processing for an end user or (2) a cohesive package of content and functionality that provides the end user with some required capability. Therefore, component-level design for WebApps often incorporates elements of content design and functional design.

Content Design at the Component Level

Content design at the component level focuses on content objects and the manner in which they may be packaged for presentation to a WebApp end user. The formality of content design at the component level should be tuned to the characteristics of the WebApp to be built. In many cases, content objects need not be organized as components and can be manipulated individually. However, as the size and complexity (of the WebApp, content objects, and their interrelation-ships) grows, it may be necessary to organize content in a way that allows easier reference and design manipulation. In addition, if content is highly dynamic (e.g., the content for an online auction site), it becomes important to establish a clear structural model that incorporates content components.

Functional Design at the Component Level

WebApp functionality is delivered as a series of components developed in parallel with the information architecture to ensure consistency. We beginby considering both the requirements model and the initial information architecture and then examining how functionality affects the user's interaction with the application, the information that is presented, and the user tasks that are conducted.

During architectural design, WebApp content and functionality are combined to create a functional architecture. A ***functional architecture*** is a representation of the functional domain of the WebApp and describes the key functional components in the WebApp and how these components interact with each other

Design Document Template

Contents

CASE STUDY

Ariane 5 launch accident

This case study describes the accident that occurred on the initial launch of the Ariane 5 rocket, a launcher developed by the European Space Agency. The rocket exploded shortly after take-off and the subsequent enquiry showed that this was due to a fault in the software in the inertial navigation system.

In June 1996, the then new Arianne 5 rocket was launched on its maiden flight. It carried a payload of scientific satellites. Ariane 5 was commercially very significant for the European Space Agency as it could carry a much heavier payload than the Ariane 4 series of launchers. Thirty seven seconds into the flight, software in the inertial navigation system, whose software was reused from Ariane 4, shut down causing incorrect signals to be sent to the engines. These swivelled in such a way that uncontrollable stresses were placed on the rocket and it started to break up. Ground controllers initiated self-destruct and the rocket and payload was destroyed.

A subsequent enquiry showed that the cause of the failure was that the software in the inertial reference system shut itself down because of an unhandled numeric exception (integer overflow). There was a backup software system but this was not diverse so it failed in the same way.

The Ariane 5 launcher failure

While developing the Ariane 5 space launcher, the designers decided to reuse the inertial reference software that had performed successfully in the Ariane 4 launcher. The inertial reference software maintains the stability of the rocket. The designers decided to reuse this without change (as you would do with components), although it included additional functionality that was not required in Ariane 5.

In the first launch of Ariane 5, the inertial navigation software failed, and the rocket could not be controlled. The rocket and its payload were destroyed. The cause of the problem was an unhandled exception when a conversion of a fixed-point number to an integer resulted in a numeric overflow. This caused the runtime system to shut down the inertial reference system, and launcher stability could not be maintained. The fault had never occurred in Ariane 4 because it had less powerful engines and the value that was converted could not be large enough for the conversion to overflow.

This illustrates an important problem with software reuse. Software may be based on assumptions about the context where the system will be used, and these assumptions may not be valid in a different situation.

More information about this failure is available at: http://software-engineering-book.com/case-studies/ariane5/