MANAGEMENT OF SOFTWARE SYSTEMS

Module 2

Rijin IK

Assistant Professor
Department of Computer Science and Engineering
Vimal Jyothi Engineering College
Chemperi

February 11, 2023

Outline

Functional and non-functional requirements

Requirements

- The descriptions of the services that a system should provide and the constraints on its operation.

Requirements Engineering (RE)

- The process of finding out, analyzing, documenting and checking the services and constraints of a system.
- The first stage of the software engineering process.

Functional and non-functional requirements

Concept of User and System Requirements

- **User requirements**
  - User requirements are statements, in a natural language plus diagrams, of what services the system is expected to provide to system users and the constraints under which it must operate.
  - High-level abstract requirements.
- **System requirements**
  - Detailed description of what the system should do.
    - Detailed descriptions of the software system's functions, services, and operational constraints.
  - The system requirements document (sometimes called a functional specification) should define exactly what is to be implemented.

Functional and non-functional requirements

User requirements definition

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

System requirements specification

1.1 On the last working day of each month, a summary of the drugs prescribed, their cost and the prescribing clinics shall be generated.
1.2 The system shall generate the report for printing after 17.30 on the last working day of the month.
1.3 A report shall be created for each clinic and shall list the individual drug names, the total number of prescriptions, the number of doses prescribed and the total cost of the prescribed drugs.
1.4 If drugs are available in different dose units (e.g. 10mg, 20mg, etc.) separate reports shall be created for each dose unit.
1.5 Access to drug cost reports shall be restricted to authorized users as listed on a management access control list.

Figure 1: User and system requirements

Functional and non-functional requirements

Functional Requirements	**Non-functional Requirements**
• Statements of services the system should provide • How the system should react to particular inputs • How the system should behave in particular situations.	Constraints on the services or functions offered by the system.
Explicitly state what the system should not do.	Include timing constraints, constraints on the development process, and constraints imposed by standards.

Functional and non-functional requirements

Functional Requirements

- Describe what the system should do.
- Describe functionality or system services
- These requirements depends on the type of software being developed, the expected users of the software, and the general approach taken by the organization when writing requirements.
- When expressed as user requirements, it should be written in natural language so that system users and managers can understand them.
- Functional system requirements expand the user requirements and are written for system developers.
- The user and system requirements should be clear, unambiguous, easy to understand, complete, and consistent.

Functional and non-functional requirements

Examples for functional requirements for the Mentcare system:

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

Functional and non-functional requirements

Non- functional Requirements

- Non-Functional Requirements defines the constraints or characteristics on the system.
- Non-Functional Requirements deal with issues like scalability, maintainability, performance, portability, security, reliability, and many more.
- Failing to meet a non-functional requirement can mean that the whole system is unusable.
- The implementation of these requirements may be spread throughout the system, for two reasons
  1. May affect the overall architecture of a system rather than the individual components.
  2. May generate several, related functional requirements that define new system services that are required if the non-functional requirement is to be implemented.

Functional and non-functional requirements

Types of nonfunctional requirement

1. **Product requirements**
   - These requirements specify or constrain the runtime behavior of the software.
   - Examples include how fast the system must execute, how much memory it requires, etc
2. **Organizational requirements**
   - These requirements are broad system requirements derived from policies and procedures in the customer's and developer's organizations.
   - e.g. Process standards used, implementation requirements, etc.
3. **External requirements**
   - This broad heading covers all requirements that are derived from factors external to the system and its development process.
   - e.g. interoperability requirements, legislative requirements, etc

Functional and non-functional requirements

Types of nonfunctional requirement

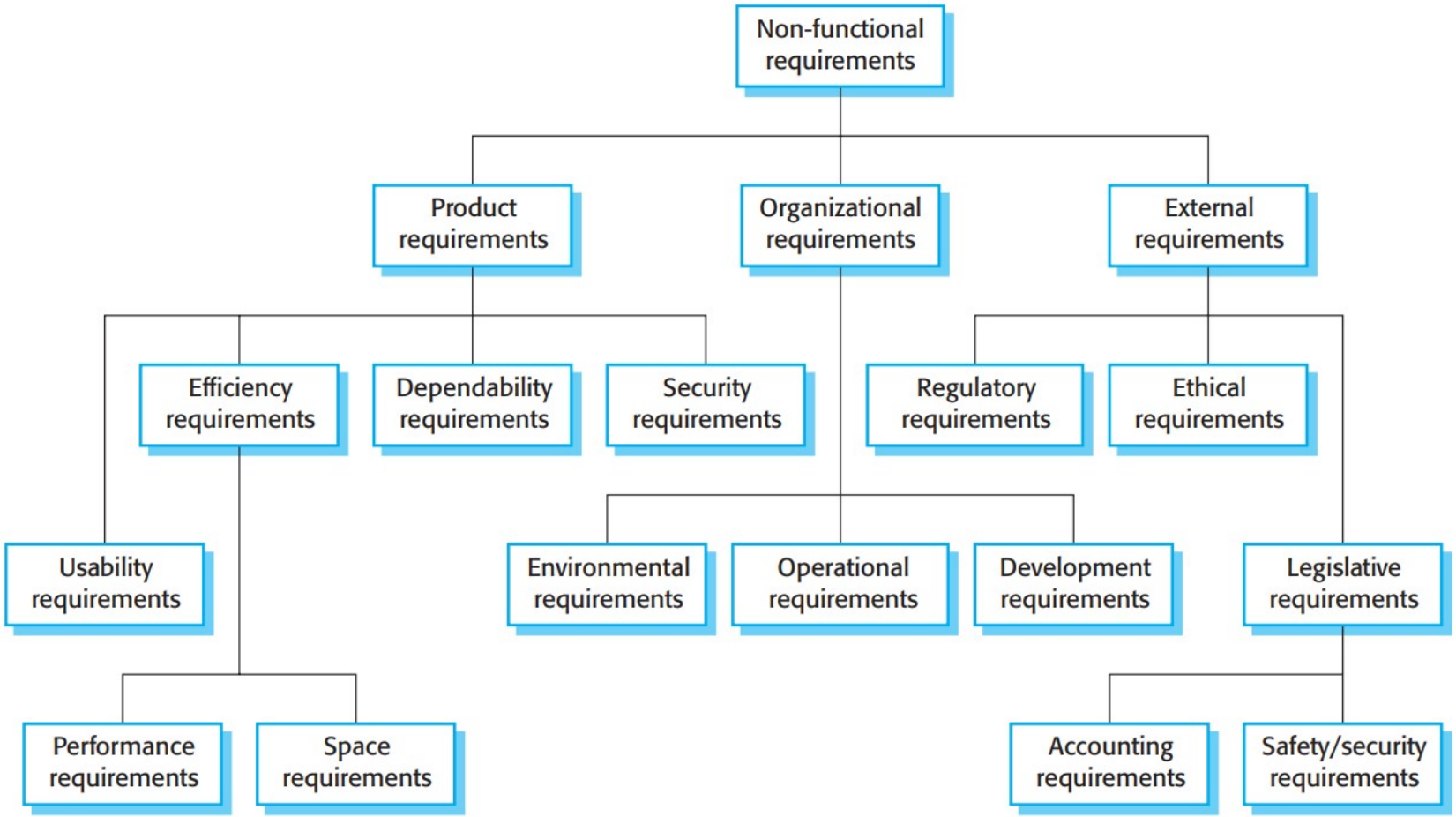


Figure 2: Types of non-functional requirements

Functional and non-functional requirements

Examples of possible non-functional requirements for the Mentcare system

PRODUCT REQUIREMENT
The Mentcare system shall be available to all clinics during normal working hours (Mon–Fri, 08:30–17:30). Downtime within normal working hours shall not exceed 5 seconds in any one day.

ORGANIZATIONAL REQUIREMENT
Users of the Mentcare system shall identify themselves using their health authority identity card.

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

Functional and non-functional requirements

Metrics for specifying non-functional requirements

Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Megabytes/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

Requirements engineering processes

- The process of finding out, analyzing, documenting and checking the services and constraints of a system.

RE involves three key activities:

1. Discovering requirements by interacting with stakeholders (elicitation and analysis)
2. Converting these requirements into a standard form (specification)
3. Checking that the requirements actually define the system that the customer wants (validation)

The output of the RE process is a system requirements document.

Requirements engineering processes

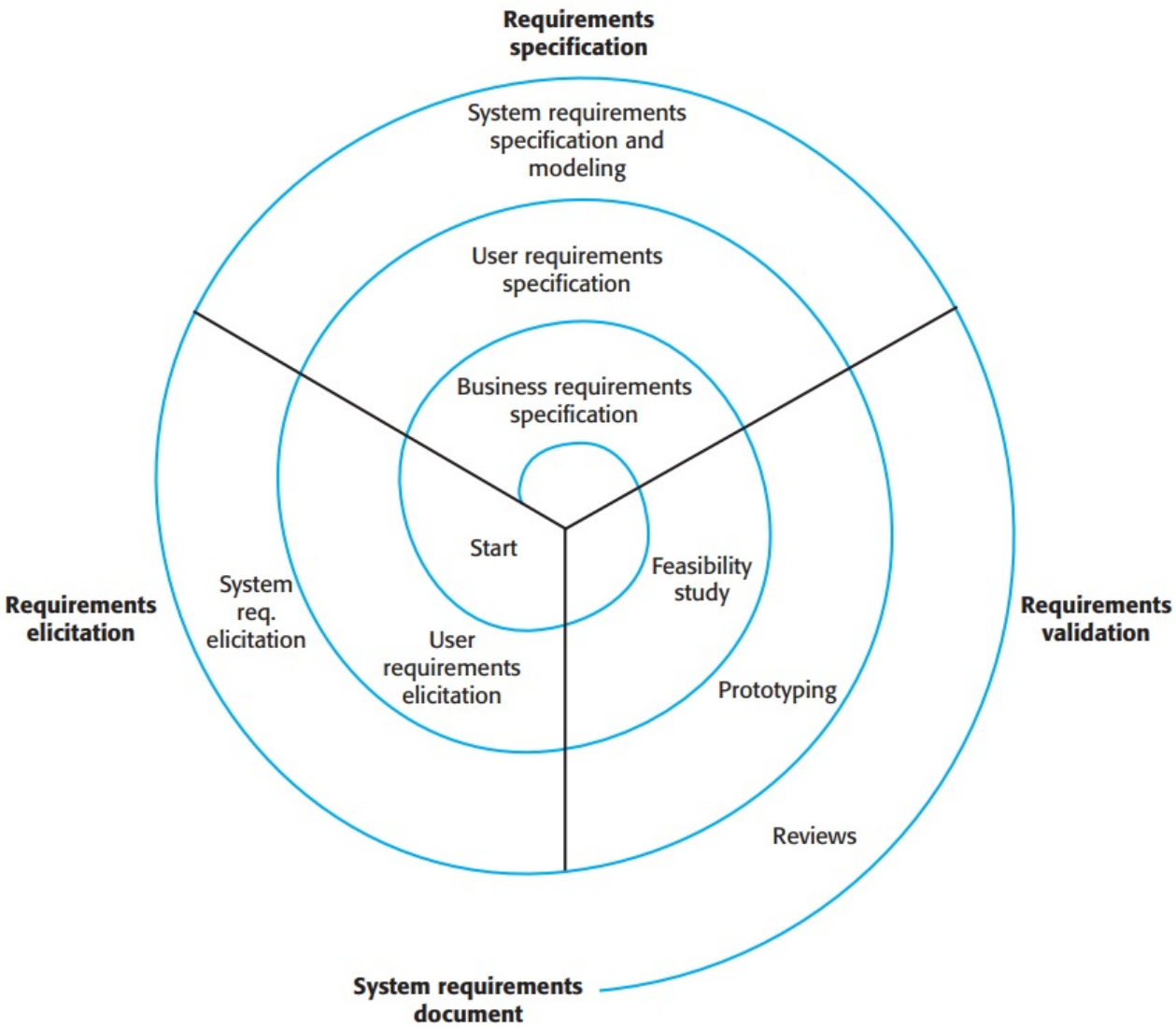


Figure 3: A spiral view of the requirements engineering process

Requirements elicitation

- The aims of the requirements elicitation process are to understand the work that stakeholders do and how they might use a new system to help support that work.
- During requirements elicitation, software engineers work with stakeholders to find out about the application domain, work activities, the services and system features that stakeholders want, the required performance of the system, hardware constraints, and so on.

Requirements elicitation

Difficulties/challenges in Eliciting and understanding requirements

- Stakeholders often don't know what they want from a computer system except in the most general terms.
- Stakeholders in a system naturally express requirements in their own terms and with implicit knowledge of their own work.
- Different stakeholders, with diverse requirements, may express their requirements in different ways.
- Political factors may influence the requirements of a system.
- The economic and business environment in which the analysis takes place is dynamic. New requirements may emerge from new stakeholders who were not originally consulted.

The requirements elicitation and analysis process

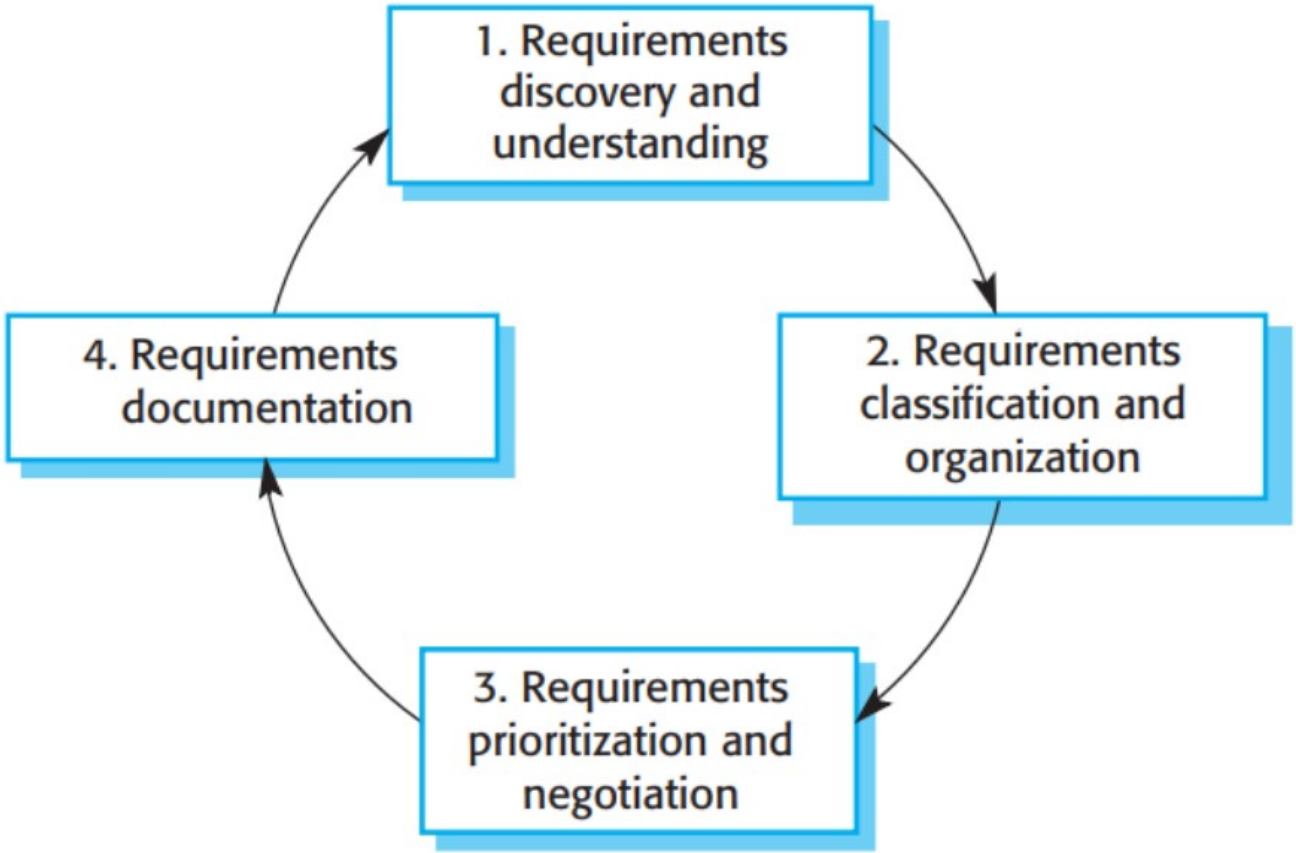


Figure 4: The requirements elicitation and analysis process

The requirements elicitation and analysis process

Requirements discovery and understanding

- This is the process of interacting with stakeholders of the system to discover their requirements. Domain requirements from stakeholders and documentation are also discovered during this activity.

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

- This activity is concerned with prioritizing requirements and finding and resolving requirements conflicts through negotiation(when multiple stakeholders are involved). Usually, stakeholders have to meet to resolve differences and agree on compromise requirements.

Requirements documentation

- The requirements are documented and input into the next round of the spiral. An early draft of the software requirements documents may be produced at this stage.

Requirements elicitation

Requirements elicitation techniques

There are two fundamental approaches to requirements elicitation:

1. **Interviewing**, where you talk to people about what they do.
2. **Observation or ethnography**, where you watch people doing their job to see what artifacts they use, how they use them, and so on.

Requirements elicitation techniques

Interviewing

Interviews may be of two types:

- **Closed interviews**, where the stakeholder answers a predefined set of questions.
- **Open interviews**, in which there is no predefined agenda. The requirements engineering team explores a range of issues with system stakeholders and hence develops a better understanding of their needs.

To be an effective interviewer, you should bear two things in mind

- You should be open-minded, avoid preconceived ideas about the requirements, and willing to listen to stakeholders.
- You should prompt the interviewee to get discussions going by using a springboard question or a requirements proposal, or by working together on a prototype system.

Requirements elicitation techniques

Ethnography

- The day-to-day work is observed, and notes are made of the actual tasks in which participants are involved.
- The value of ethnography is that it helps discover implicit system requirements that reflect the actual ways that people work, rather than the formal processes defined by the organization.

Ethnography is particularly effective for discovering two types of requirements:

1. Requirements derived from the way in which people actually work, rather than the way in which business process definitions say they ought to work. In practice, people never follow formal processes.
2. Requirements derived from cooperation and awareness of other people's activities.

Requirements elicitation techniques

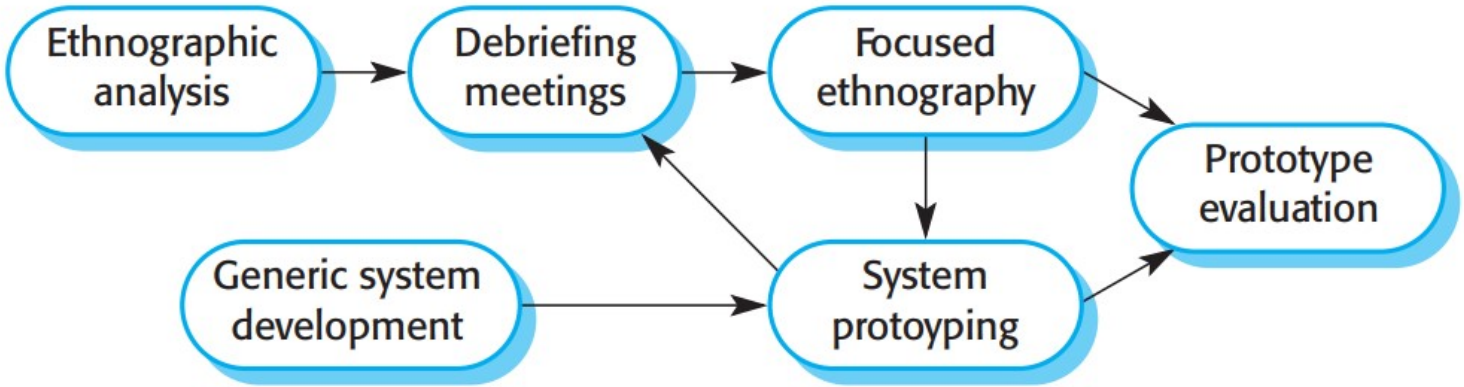


Figure 5: Ethnography and prototyping for requirements analysis

Requirements validation

Requirements validation

- Requirements validation is the process of checking that requirements define the system that the customer really wants.
- Requirements validation is critically important because errors in a requirements document can lead to extensive rework costs when these problems are discovered during development or after the system is in service.

Requirements validation

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

1. **Validity checks** check that the requirements reflect the real needs of system users.
2. **Consistency checks** requirements in the document should not conflict.
3. **Completeness checks** requirements document should include requirements that define all functions and the constraints intended by the system user.
4. **Realism checks** checked to ensure that they can be implemented within the proposed budget for the system.
5. **Verifiability** system requirements should always be written so that they are verifiable.

Requirements validation

Requirements validation techniques

1. **Requirements reviews** The requirements are analyzed systematically by a team of reviewers who check for errors and inconsistencies.
2. **Prototyping:** This involves developing an executable model of a system and using this with end-users and customers to see if it meets their needs and expectations. Stakeholders experiment with the system and feed back requirements changes to the development team.
3. **Test-case generation:** Requirements should be testable. If a test is difficult or impossible to design, this usually means that the requirements will be difficult to implement and should be reconsidered. Developing tests from the user requirements before any code is written is an integral part of test-driven development.

Requirements change

Requirements change

1. Requirements management planning
2. Requirements change management

Requirements change

- The requirements for large software systems are always changing. One reason for the frequent changes is that these systems are often developed to address "wicked" problems—problems that cannot be completely defined.

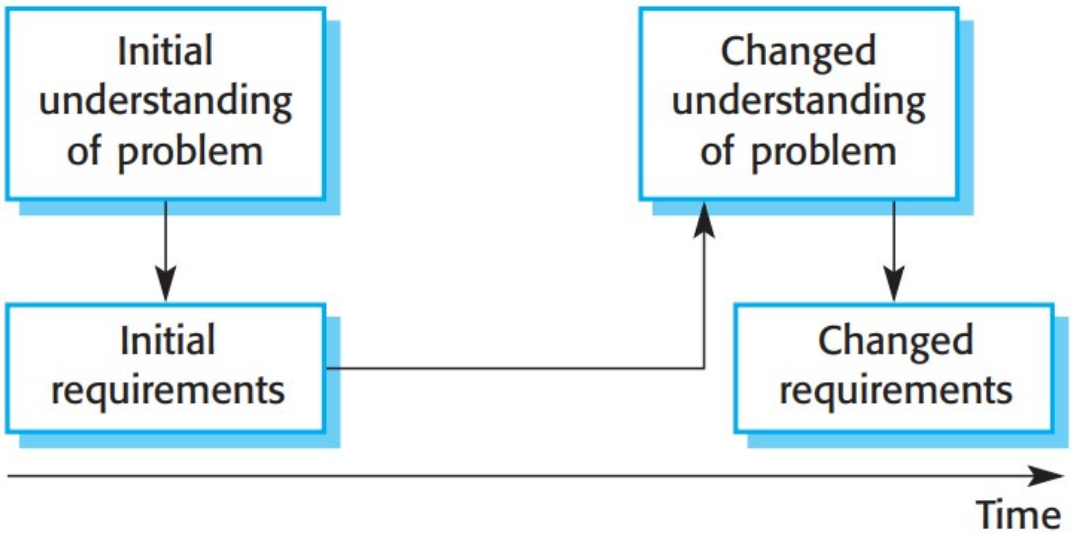


Figure 6: Requirements evolution

Requirements change

Reason

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

- The business and technical environment of the system always changes after installation. New hardware may be introduced and existing hardware updated.
- The people who pay for a system and the users of that system are rarely the same people.
  - System customers impose requirements because of organizational and budgetary constraints. These may conflict with end-user requirements, and, after delivery, new features may have to be added for user support if the system is to meet its goals.
- Large systems usually have a diverse stakeholder community, with stakeholders having different requirements. Their priorities may be conflicting or contradictory.

Requirements change

Requirements management planning

- Requirements management planning is concerned with establishing how a set of evolving requirements will be managed.
- During the planning stage, you have to decide on a number of issues:
- **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.
  - **Traceability policies:** These policies define the relationships between each requirement and between the requirements and the system design that should be recorded.
  - **Tool support:** Requirements management involves the processing of large amounts of information about the requirements. Tools that may be used range from specialist requirements management systems to shared spreadsheets and simple database systems.

Requirements change

Requirements change management

Deciding if a requirements change should be accepted

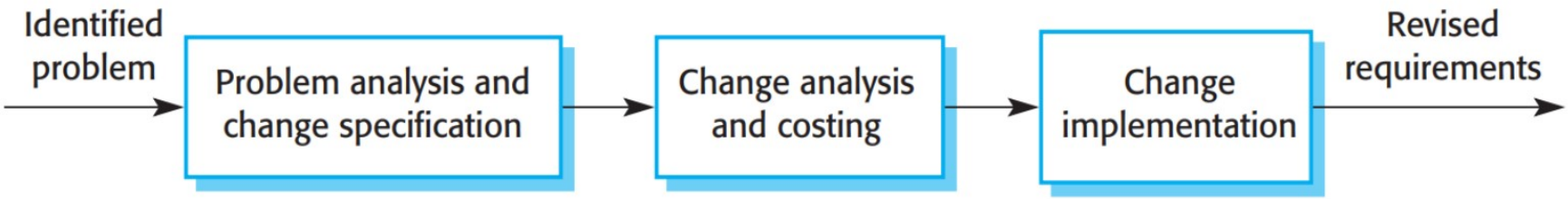


Figure 7: Requirements change management

Requirements change management

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

Traceability Matrix

- Traceability matrix is a table type document that is used in the development of software application to trace requirements.
- In this document the test cases are mapped to the corresponding requirement
  - To ensure that every requirement is covered in the form of a test case
  - To find any gap between requirement and test case
- 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)**.

Traceability Matrix

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

Figure 8: Example of RTM template

Traceability Matrix

Goals of Traceability Matrix:

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

Types of Traceability Test Matrix

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

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

Traceability Matrix

Forward Traceability

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

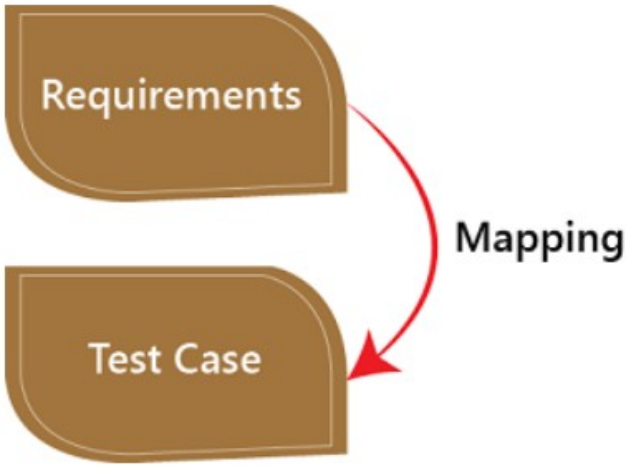


Figure 9: Forward Traceability

Traceability Matrix

Backward or reverse traceability

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

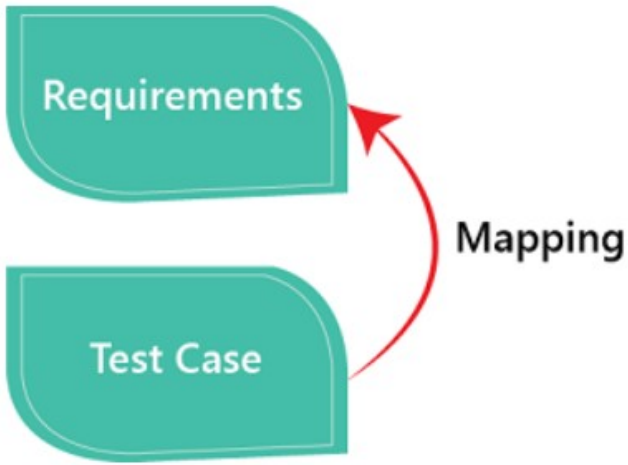


Figure 10: Backward or reverse traceability

Traceability Matrix

Bi-directional traceability

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

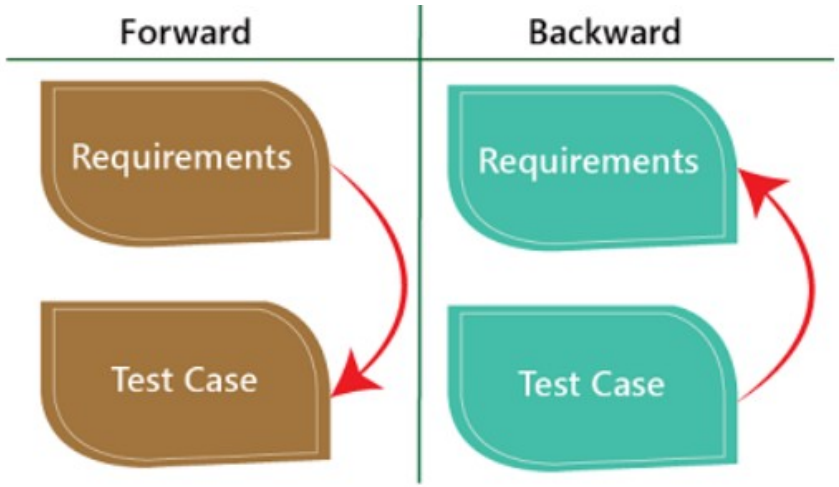


Figure 11: Bi-directional traceability

Traceability Matrix

Advantage 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.
- In this, 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.

Developing use cases

Use case

- Use case is , how an end user (playing one of a number of possible roles) interacts with the system under a specific set of circumstances.
- It may be,
  - A narrative text
  - An outline of tasks or interactions.
  - A template-based description.
  - A diagrammatic representation.
- Regardless of its form, a use case depicts the software or system from the end user's point of view.

DEVELOPING USE CASES

- The first step in writing a use case is to define the set of **"actors"** that will be involved in the story.
- **Actor**
  - The users that interact with a system.
  - An actor can be a person, an organization, or an outside system that interacts with your application or system.
  - They must be external objects that produce or consume data.
- An actor and an end user are not necessarily the same thing.
- A typical user may play a number of different roles when using a system, whereas an actor represents a class of external entities (often, but not always, people) that play just one role in the context of the use case.

Developing use cases

Example

- Consider a machine operator (a user) who interacts with the control computer for a manufacturing cell that contains a number of robots and numerically controlled machines.
- After careful review of requirements, the software for the control computer requires four different modes (roles) for interaction:
  - programming mode
  - test mode
  - monitoring mode
  - troubleshooting mode.
- Therefore, four actors can be defined: programmer, tester, monitor, and troubleshooter.

Developing use cases

Types of actor

- Because requirements elicitation is an evolutionary activity, not all actors are identified during the first iteration.
- It is possible to identify **primary actors** during the first iteration and **secondary actors** as more is learned about the system.

Primary actors

- 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

- Secondary actors support the system so that primary actors can do their work.

Once actors have been identified, use cases can be developed

Developing use cases

A number of questions should be answered by a use case

1. Who is the primary actor, the secondary actor(s)?
2. What are the actor's goals?
3. What preconditions should exist before the story begins?
4. What main tasks or functions are performed by the actor?
5. What exceptions might be considered as the story is described?
6. What variations in the actor's interaction are possible?
7. What system information will the actor acquire, produce, or change?
8. Will the actor have to inform the system about changes in the external environment?
9. What information does the actor desire from the system?
10. Does the actor wish to be informed about unexpected changes?

Developing use cases

Example:basic SafeHome requirements define 4 actors:

1. homeowner (a user),
2. setup manager (likely the same person as homeowner, but playing a different role),
3. sensors (devices attached to the system), and
4. monitoring and response subsystem (the central station that monitors the SafeHome home security function).

For the purposes of this example, we consider only the homeowner actor. The homeowner actor interacts with the home security function in a number of different ways using either the alarm control panel or a PC.

The homeowner:

- enters a password to allow all other interactions,
- inquires about the status of a security zone,
- inquires about the status of a sensor,
- presses the panic button in an emergency, and
- activates/deactivates the security system.

Developing use cases

Considering the situation in which the homeowner uses the control panel, the basic use case for system activation follows:

1. The homeowner observes the SafeHome control panel to determine if the system is ready for input.
   - If the system is not ready, a not ready message is displayed on the LCD display,
   - and the homeowner must physically close windows or doors so that the not ready message disappears. [A not ready message implies that a sensor is open; i.e., that a door or window is open.]
2. The homeowner uses the keypad to key in a four-digit password.
   - The password is compared with the valid password stored in the system.
   - If the password is incorrect, the control panel will beep once and reset itself for additional input.
   - If the password is correct, the control panel awaits further action.

Developing use cases

3 The homeowner selects and keys in stay or away to activate the system.
   - Stay activates only perimeter sensors (inside motion detecting sensors are deactivated). Away activates all sensors.

4 When activation occurs, a red alarm light can be observed by the homeowner.

Developing use cases

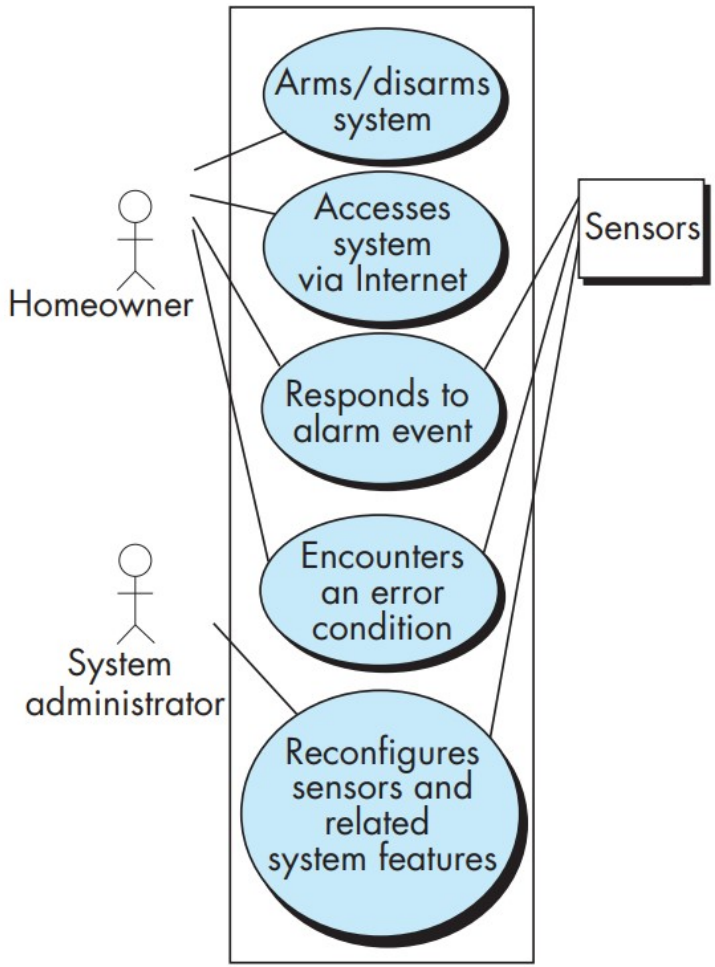


Figure 12: use case diagram for SafeHome home security function

Developing use cases

Template for detailed descriptions of use cases:

Use case:	*InitiateMonitoring*
Primary actor:	Homeowner.
Goal in context:	To set the system to monitor sensors when the homeowner leaves the house or remains inside.
Preconditions:	System has been programmed for a password and to recognize various sensors.
Trigger:	The homeowner decides to "set" the system, that is, to turn on the alarm functions.

Developing use cases

Scenario :

1. Homeowner: observes control panel
2. Homeowner: enters password
3. Homeowner: selects “stay” or “away”
4. Homeowner: observes read alarm light to indicate that SafeHome has been armed

Exceptions:

1. Control panel is not ready: homeowner checks all sensors to determine which are open; closes them.
2. Password is incorrect (control panel beeps once): homeowner reenters correct password.
3. Password not recognized: monitoring and response subsystem must be contacted to reprogram password.
4. Stay is selected: control panel beeps twice and a stay light is lit; perimeter sensors are activated.
5. Away is selected: control panel beeps three times and an away light is lit; all sensors are activated.

Developing use cases

Priority:	Essential, must be implemented
When available:	First increment
Frequency of use:	Many times per day
Channel to actor:	Via control panel interface
Secondary actors:	Support technician, sensors

Channels to secondary actors:

Support technician: phone line

Sensors: hardwired and radio frequency interfaces

Developing use cases

Open issues:

- Should there be a way to activate the system without the use of a password or with an abbreviated password?
- Should the control panel display additional text messages?
- How much time does the homeowner have to enter the password from the time the first key is pressed?
- Is there a way to deactivate the system before it actually activates?

Software Requirements Specification Template

Chapter	Description
Preface	This defines 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 describes 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 defines 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 presents a high-level overview of the anticipated system architecture, showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.
System requirements specification	This describes 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 chapter includes 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 describes 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 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.

Figure 13: The structure of a requirements document

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

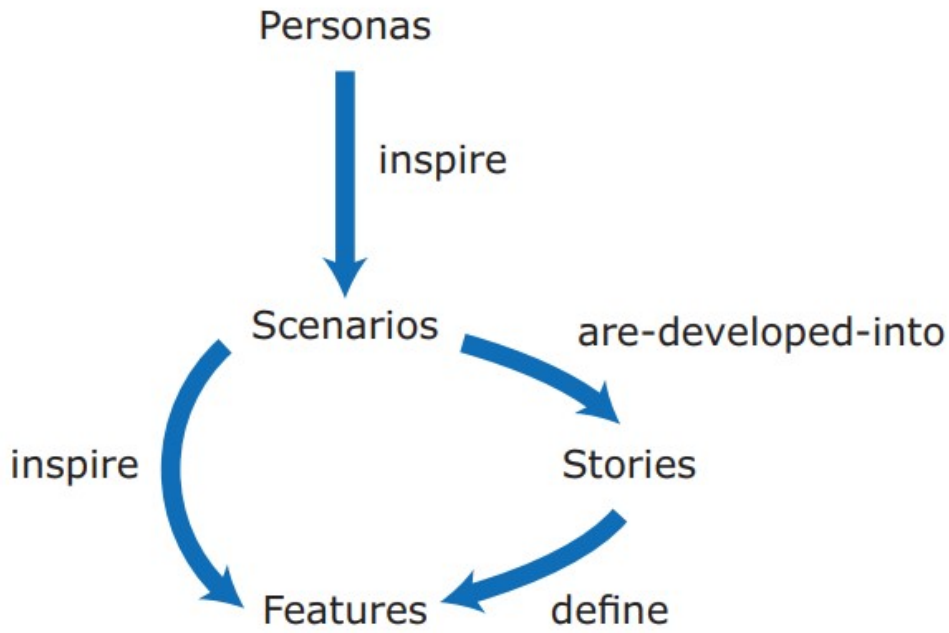


Figure 14: From personas to features

Personas

Personas

- Personas are about "**imagined users," character portraits of types of user that you think might adopt your product.**
  - **Example:** if your product is aimed at *managing appointments for dentists*, you might create a *dentist persona, a receptionist persona, and a patient persona*.
- Personas of different types of users help you imagine what these users may want to do with your software and how they might use it.
- They also help you envisage difficulties that users might have in understanding and using product features.

Personas

Persona should include the following :

- Description about the the users' backgrounds
- Description about why the users might want to use your product
- Description about their education and technical skills.

Personas

Persona descriptions

- A persona should '**paint a picture' of a type of product user**.
- They should be relatively **short and easy-to-read**.
- Should describe their **background** and **why they might want to use your product**.
- These help you assess whether or not a software feature is likely to be useful, understandable and usable by typical product users.

Persona descriptions

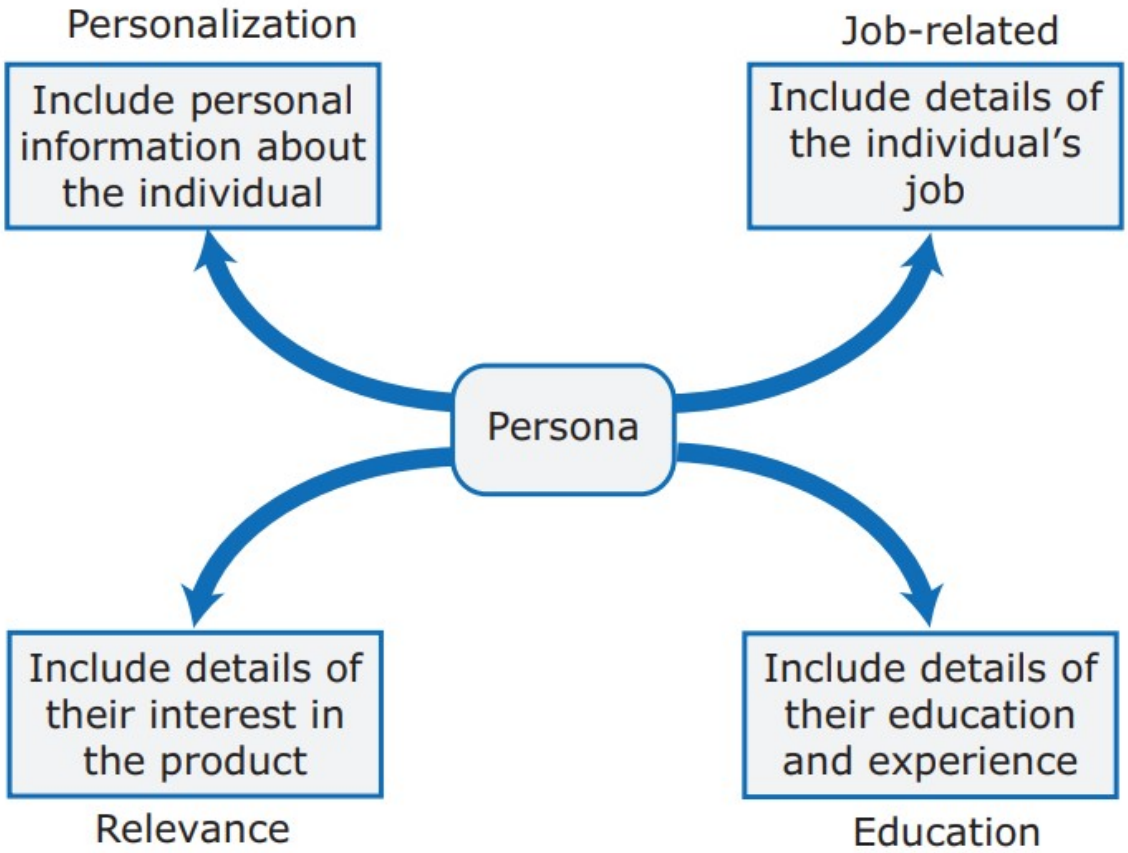


Figure 15: Persona descriptions

Persona descriptions

Jack, a primary school teacher

Jack, age 32, is a primary school (elementary school) teacher in Ullapool, a large coastal village in the Scottish Highlands. He teaches children from ages 9 to 12. He was born in a fishing community north of Ullapool, where his father runs a marine fuels supply business and his mother is a community nurse. He has a degree in English from Glasgow University and retrained as a teacher after several years working as a web content author for a large leisure group.

Jack's experience as a web developer means that he is confident in all aspects of digital technology. He passionately believes that the effective use of digital technologies, blended with face-to-face teaching, can enhance the learning experience for children. He is particularly interested in using the iLearn system for project-based teaching, where students work together across subject areas on a challenging topic.

Figure 16: Persona descriptions

Persona descriptions

Emma, a history teacher

Emma, age 41, is a history teacher in a secondary school (high school) in Edinburgh. She teaches students from ages 12 to 18. She was born in Cardiff in Wales, where both her father and her mother were teachers. After completing a degree in history from Newcastle University, she moved to Edinburgh to be with her partner and trained as a teacher. She has two children, aged 6 and 8, who both attend the local primary school. She likes to get home as early as she can to see her children, so often does lesson preparation, administration, and marking from home.

Emma uses social media and the usual productivity applications to prepare her lessons, but is not particularly interested in digital technologies. She hates the virtual learning environment that is currently used in her school and avoids using it if she can. She believes that face-to-face teaching is most effective. She might use the iLearn system for administration and access to historical films and documents. However, she is not interested in a blended digital/face-to-face approach to teaching.

Figure 17: A persona for a history teacher

Persona descriptions

Elena, a school IT technician

Elena, age 28, is a senior IT technician in a large secondary school (high school) in Glasgow with over 2000 students. Originally from Poland, she has a diploma in electronics from Potsdam University. She moved to Scotland in 2011 after being unemployed for a year after graduation. She has a Scottish partner, no children, and hopes to develop her career in Scotland. She was originally appointed as a junior technician but was promoted, in 2014, to a senior post responsible for all the school computers.

Although not involved directly in teaching, Elena is often called on to help in computer science classes. She is a competent Python programmer and is a "power user" of digital technologies. She has a long-term career goal of becoming a technical expert in digital learning technologies and being involved in their development. She wants to become an expert in the iLearn system and sees it as an experimental platform for supporting new uses for digital learning.

Figure 18: A persona for an IT technician

Scenarios

Scenarios

- A scenario is a **narrative** that **describes how a user, or a group of users, might use your system.**
- There is no need to include everything in a scenario – the scenario isn't a system specification.
- It is simply a **description** of a situation **where a user is using your product's features to do something that they want to do**.
- Scenario descriptions may vary in length **from two to three paragraphs up to a page of text.**

Scenarios

Scenarios Cont..

- An **imagined or projected sequence of events**,
- 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.

Scenarios

Fishing in Ullapool

Jack is a primary school teacher in Ullapool, teaching P6 pupils. He has decided that a class project should be focused around the fishing industry in the area, looking at the history, development, and economic impact of fishing.

As part of this, students are asked to gather and share reminiscences from relatives, use newspaper archives, and collect old photographs related to fishing and fishing communities in the area. Pupils use an iLearn wiki to gather together fishing stories and SCRAN (a history archive site) to access newspaper archives and photographs. However, Jack also needs a photo-sharing site as he wants students to take and comment on each others' photos and to upload scans of old photographs that they may have in their families. He needs to be able to moderate posts with photos before they are shared, because pre-teen children can't understand copyright and privacy issues.

Jack sends an email to a primary school teachers' group to see if anyone can recommend an appropriate system. Two teachers reply and both suggest that he use KidsTakePics, a photo-sharing site that allows teachers to check and moderate content. As KidsTakePics is not integrated with the iLearn authentication service, he sets up a teacher and a class account with KidsTakePics.

He uses the the iLearn setup service to add KidsTakePics to the services seen by the students in his class so that, when they log in, they can immediately use the system to upload photos from their phones and class computers.

Figure 19: Jack's scenario: Using the iLearn system for class projects

Scenarios

Elements of scenario

- A brief statement of the overall objective.
  - In Jack's scenario, this is to support a class project on the fishing industry.
- References to the persona involved (Jack) so that you can get information about the capabilities and motivation of that user.
- Information about what is involved in doing the activity.
  - For example, in Jack's scenario, this involves gathering reminiscences from relatives, accessing newspaper archives, and so on.
- If appropriate, an explanation of problems that can't be readily addressed using the existing system.
  - Young children don't understand issues such as copyright and privacy, so photo sharing requires a site that a teacher can moderate to make sure that published images are legal and acceptable.
- A description of one way that the identified problem might be addressed.
  - This may not always be included especially if technical knowledge is needed to solve the problem. In Jack's scenario, the preferred approach is to use an external tool designed for school students.

Scenarios

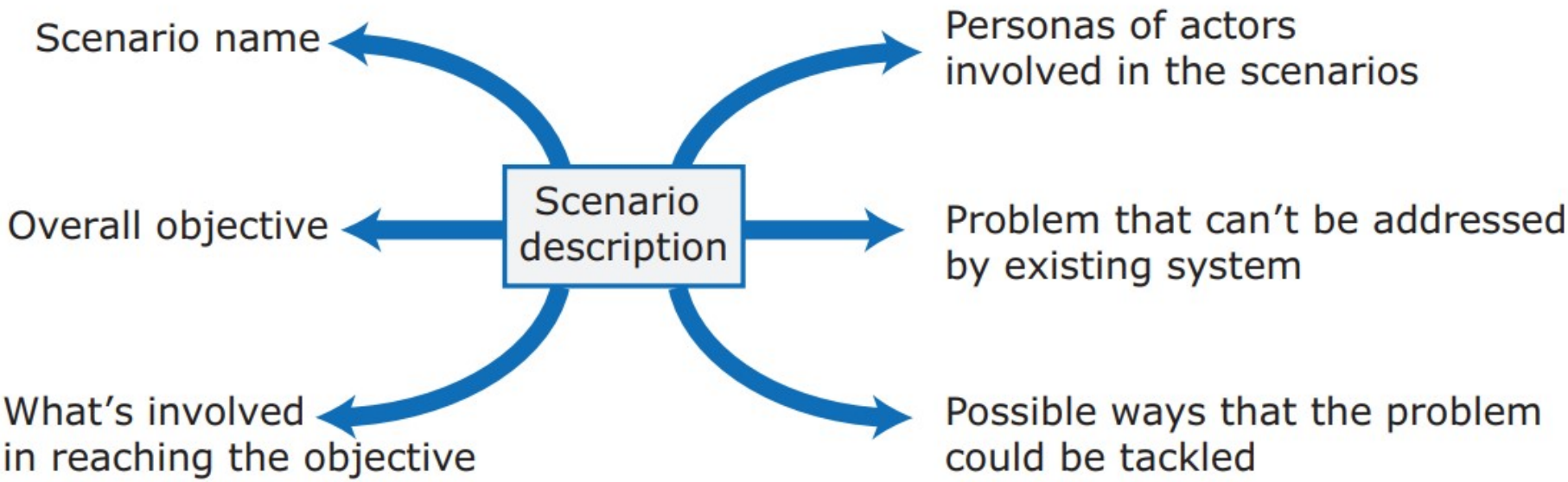


Figure 20: Elements of a scenario description

Scenarios

Writing scenarios

- 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.
- 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

Elena has been asked by David, the head of the art department in her school, to help set up an iLearn environment for his department. David wants an environment that includes tools for making and sharing art, access to external websites to study artworks, and "exhibition" facilities so that the students' work can be displayed.

Elena starts by talking to art teachers to discover the tools that they recommend and the art sites that they use for studies. She also discovers that the tools they use and the sites they access vary according to the age of their students. Consequently, different student groups should be presented with a toolset that is appropriate for their age and experience.

Once she has established what is required, Elena logs into the iLearn system as an administrator and starts configuring the art environment using the iLearn setup service. She creates sub-environments for three age groups plus a shared environment that includes tools and sites that may be used by all students.

She drags and drops tools that are available locally and the URLs of external websites into each of these environments. For each of the sub-environments, she assigns an art teacher as its administrator so that they can't refine the tool and website selection that has been set up. She publishes the environments in "review mode" and makes them available to the teachers in the art department.

After discussing the environments with the teachers, Elena shows them how to refine and extend the environments. Once they have agreed that the art environment is useful, it is released to all students in the school.

Figure 21: Elena's scenario: Configuring the iLearn system

User stories

User stories

- A user story is a well-formed, **short and simple description of a software requirement** from the **perspective of an end-user**, written in an **informal and natural language**.
- 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.

User stories

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

- This story reflects what has become the standard format of a user story:
  - **As a** <role>, I <want I need> **to** <do something>
    - As a teacher, I want to tell all members of my group when new information is available
- A variant of this standard format adds a justification for the action:
  - **As a** <role> I <want I need> **to** <do something> **so that** <reason>
    - As a teacher, I need to be able to report who is attending a class trip so that the school maintains the required health and safety records.

User stories

User stories cont..

- When you define user stories from a scenario, you provide more information to developers to help them design the product's features.
- If you are writing stories to be part of a product backlog, you should **avoid negative stories.**
- Scenarios and stories are helpful in both choosing and designing system features.
- Scenarios and user stories can be thought of as "tools for thinking" about a system rather than a system specification. They don't have to be complete or consistent, and there are no rules about how many of each you need.

User stories

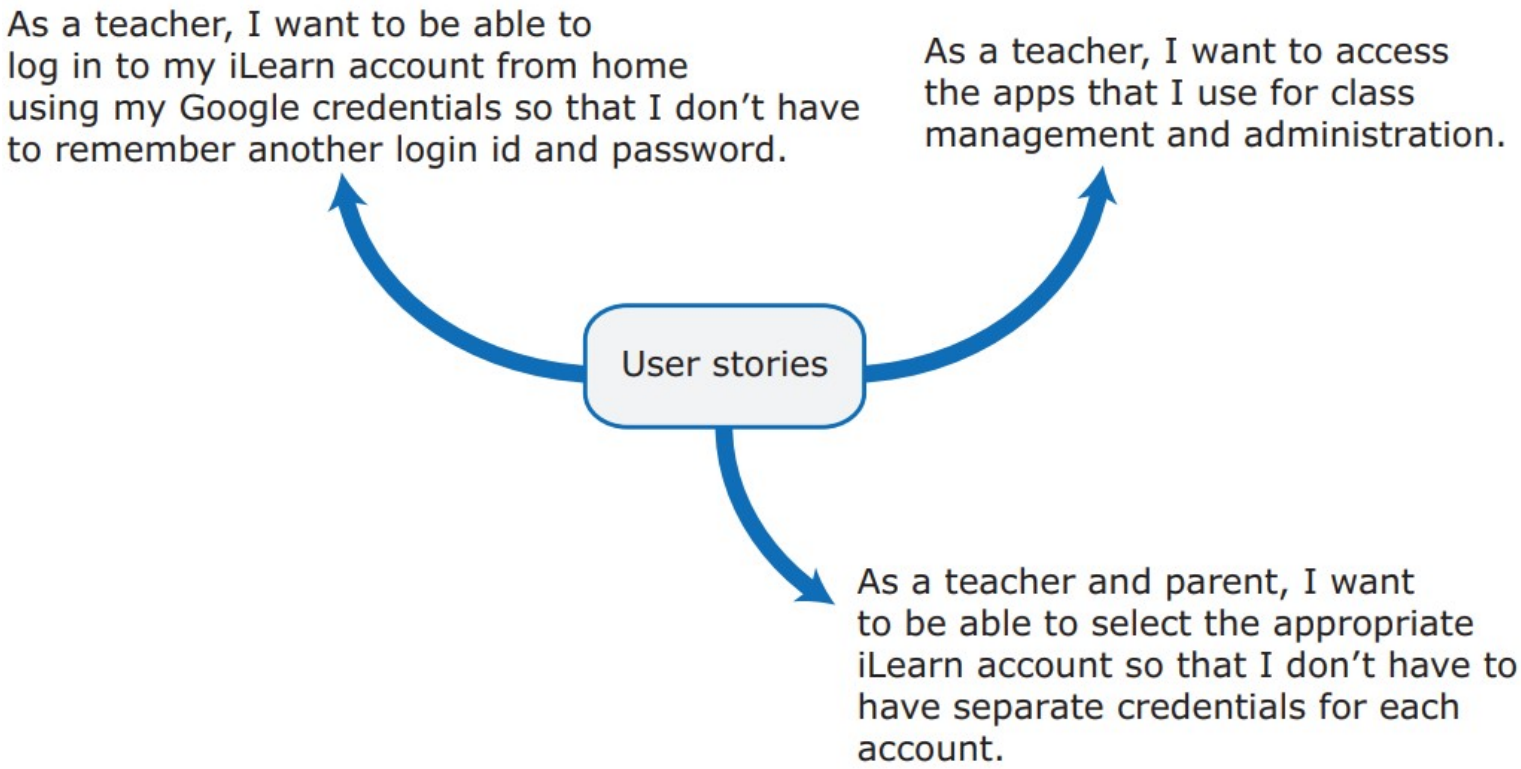


Figure 22: User stories from Emma's scenario

User stories

User stories in planning

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

Feature identification

A "feature" can be defined as: "A discrete piece of functionality desired by stakeholders"

Feature identification

- A feature is a way of allowing users to access and use your product's functionality so that the feature list defines the overall functionality of the system.
- Identify the product features that are independent, coherent and relevant:
  - **Independence:** A feature should not depend on how other system features are implemented and should not be affected by the order of activation of other features.
  - **Coherence Features:**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:**Features should reflect the way users normally carry out some task. They should not offer obscure functionality that is rarely required.

Feature identification

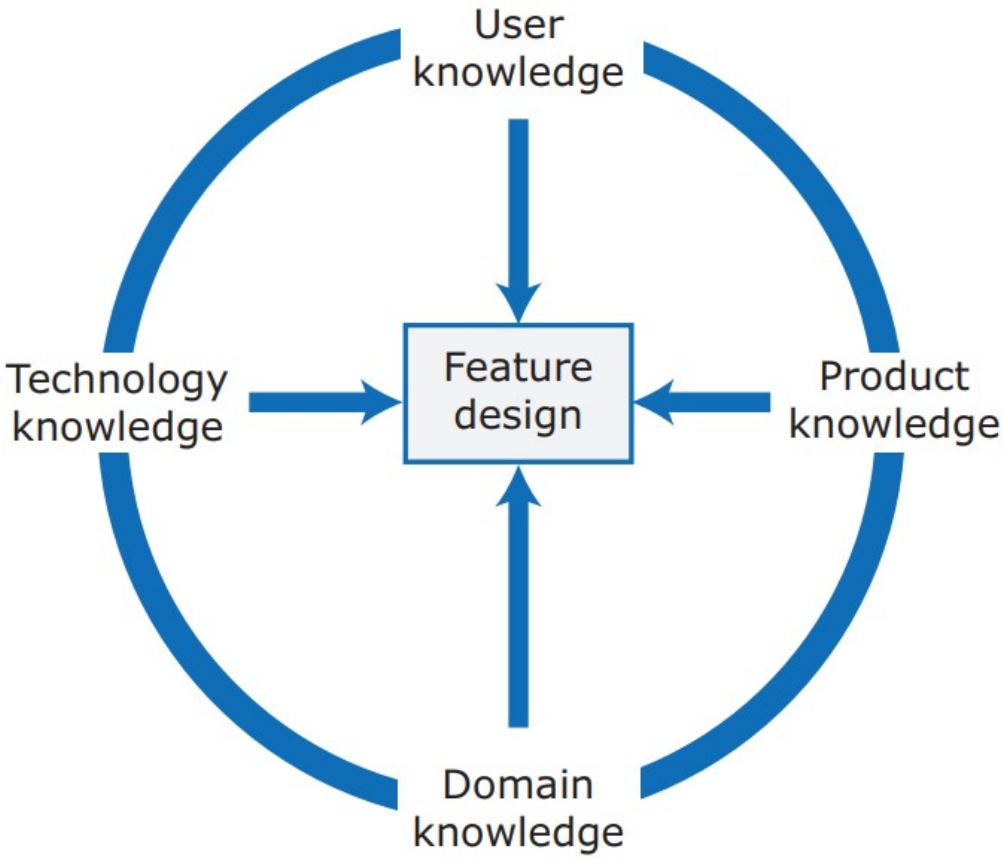


Figure 23: Feature design

Feature identification

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 24: Knowledge required for feature design

Factors in feature set design

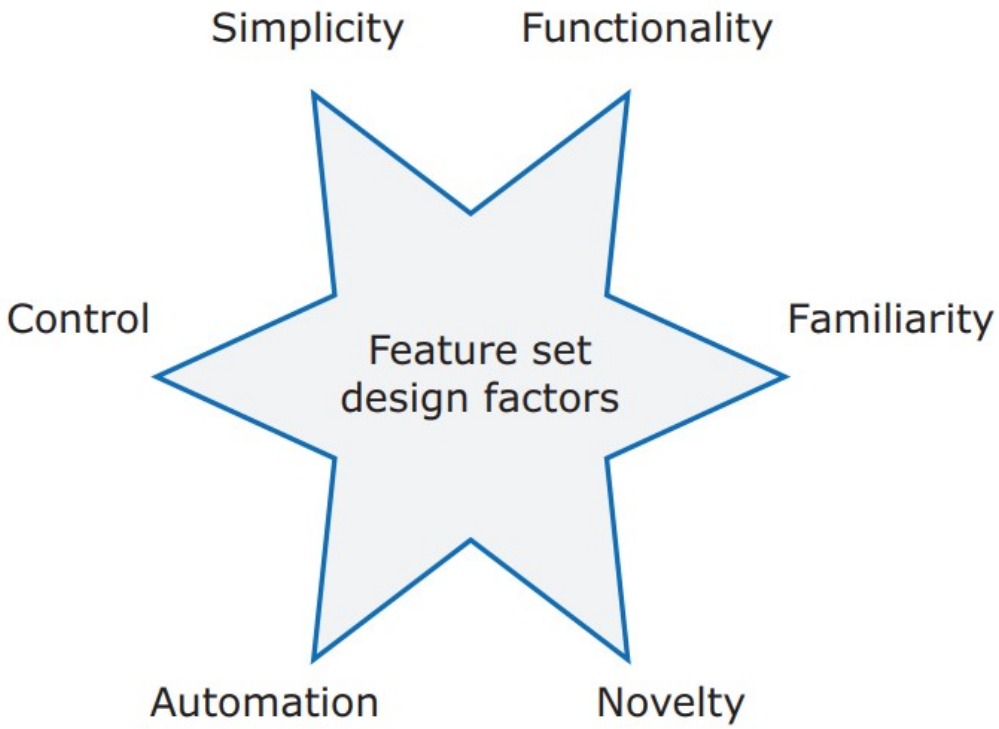


Figure 25: Factors in feature set design

Feature identification

Factors in feature set design

- **Simplicity and functionality**
  - Find a balance between providing a **simple, easy-to-use** system and **including enough functionality to attract users** with a variety of needs.
- **Familiarity and novelty**
  - Users prefer that new software should support the **familiar everyday tasks that are part of their work or life**. Find a **balance between familiar features and new features** that convince users that your product can do more than its competitors.
- **Automation and control**
  - Some users like automation, where the software does things for them. Others prefer to have control.

Feature identification

Feature creep

- Feature creep, more commonly known as scope creep, refers to when you **add excessive features to a product** that make it too **complicated or difficult** to use.
- Too many features make products hard to use and understand
- There are 3 reasons why feature creep occurs:
  - Product managers are **reluctant to say 'no'** when users ask for specific features.
  - Developers **try to match features in competing products**.
  - The product includes **features to support** both **inexperienced and experienced users**.

Feature identification

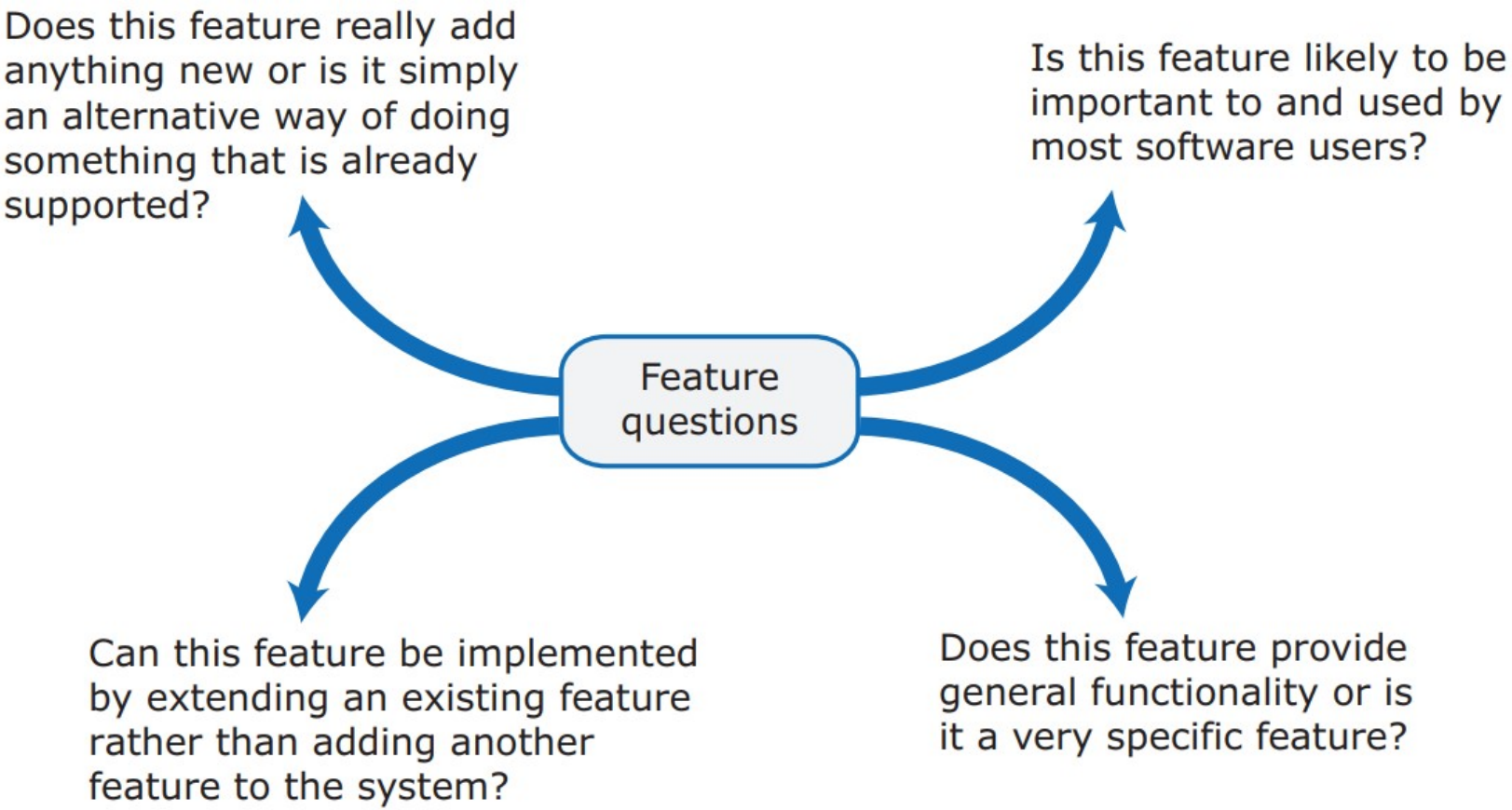


Figure 26: Avoiding feature creep

Feature identification

The feature list

- The output of the feature identification process should be a list of features that you use for designing and implementing your product.
- There is no need to go into a lot of detail about the features at this stage. You add detail when you are implementing the feature.
- You can describe features using a standard input-action-output template by using structured narrative descriptions or by a set of user stories.

Feature identification

Feature derivation

- Features can be identified **directly from the product vision or from scenarios**.
- You can **highlight phrases in narrative description** to identify features to be included in the software.
  - You should think about the features needed to support user actions, identified by active verbs, such as use and choose.

Design concepts

Design concepts

- Design with context of software engineering
- The Design process
- Design Concepts
- The Design Model

Design concepts

Design

- Software design is a process to **transform user requirements into some suitable form**, which **helps the programmer in software coding and implementation**.
- The design model **provides** detail about **software architecture, data structures, interfaces, and components** that are **necessary to implement the system.**
- **The goal of design**
  To produce a model or representation that exhibits
  - **Firmness:** A program should not have any bugs that inhibit its function.
  - **Commodity:** A useful or valuable thing
  - **Delight:** The experience of using the program should be a pleasurable one.

Design concepts

Design Within the Context of Software Engineering

- Software design is the last software engineering action within the modeling activity and sets the stage for construction (code generation and testing).

Design Within the Context of Software Engineering

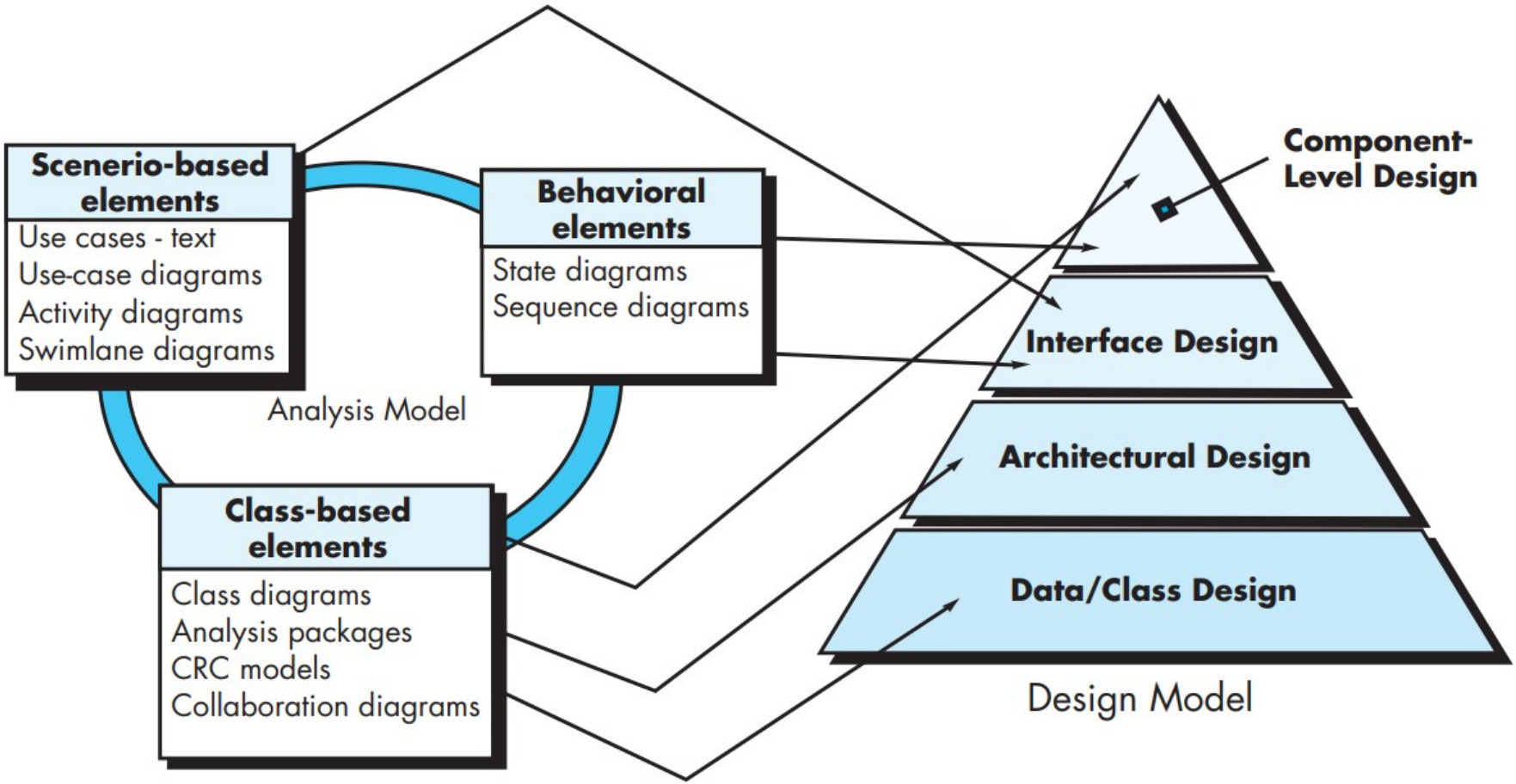


Figure 27: Translating the requirements model into the design model

Design Within the Context of Software Engineering

The requiremets model haning the following basic elements,which feed the design task.

- Scenario-based
- Class-based and
- behavioral elements,

The design process will produces 4 different design model

- **Data/Class design** – transforms analysis classes into implementation classes and data structures
- **Architectural design** – defines relationships among the major software structural elements
- **Interface design** – defines how software elements, hardware elements, and end-users communicate
- **Component-level design** – transforms structural elements into procedural descriptions of software components

THE DESIGN PROCESS

- Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software.

Software Quality Guidelines and Attributes

- Three characteristics used for evaluation of quality
  1. The design should **implement all of the explicit requirements contained in the requirements model**, and it must accommodate all of the implicit requirements desired by stakeholders.
  2. The design should be **readable and understandable** for those who generate code and test the software.
  3. The design **should provide a complete picture of the software**, addressing the data, functional, and behavioral domains from an implementation perspective.

Quality Guidelines:

1. A design is generated using the recognizable architectural styles and compose a good design characteristic of components and it is implemented in evolutionary manner for testing.
2. A design should be modular; i.e., the software should be logically partitioned into elements or subsystems.
3. A design should contain distinct representations of data, architecture, interfaces, and components.

Quality Guidelines cont..

4. A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
5. A design should lead to components that exhibit independent functional characteristics.
6. A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
7. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
8. A design should be represented using a notation that effectively communicates its meaning.

Quality Attributes:

1. **Functionality:** 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.
2. **Usability:** assessed by considering human factors , overall aesthetics, consistency, and documentation.
3. **Reliability:** 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.
4. **Performance:** measured using processing speed, response time, resource consumption, throughput, and efficiency.
5. **Supportability:** combines extensibility, adaptability, and serviceability.

Design concepts

An overview of fundamental software design concepts:

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

Design concepts

Abstraction

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

Architecture

- The complete structure of the software is known as software architecture.
- It consists of components,connectiors, and the relationship between them
- 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."

Architecture Cont..

- 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.
  - **Functional models:** used to represent the functional hierarchy of a system.

Design concepts

Patterns:

- A design structure that solves a perticular design problem within a specific context
- It provides a description that enables a designer to determine whether the pattern is applicable,
- whether the pattern can be reused, and
- whether the pattern can serve as a guide for developing for similar pattern

Design concepts

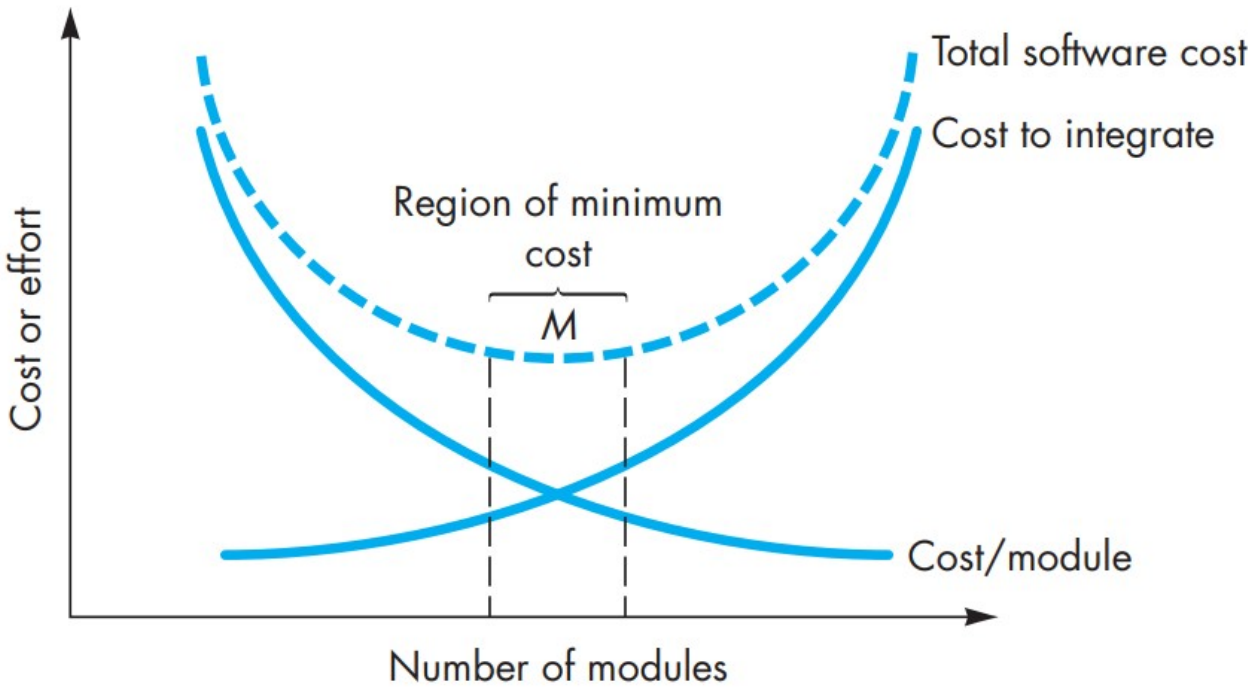
Separation of Concerns:

- 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.

Design concepts

Modularity:

- 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



Design concepts

Information Hiding :

- The designing of modules so that information (algorithms and data) contained within a module is inaccessible to other modules that have no need for such information.
- This enforces access constraints to both procedural(ie. implementation)details and local data structure

Design concepts

Functional Independence :

- The functional independence is the concept of separation and related to the concept of modularity, abstraction and information hiding.
- Independence is assessed using 2 qualitative criteria:
  - **Cohesion:** an indication of the relative functional strength of a module.
    - A cohesive module performs a single task and it requires a small interaction with the other components in other parts of the program.
  - **Coupling:** an indication of the relative interdependence among modules.
    - Coupling is an indication of interconnection between modules in a structure of software.

Design concepts

Refinement :

- Stepwise refinement is a top-down design strategy.
- It 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.
- You then elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.
- Abstraction and refinement are complementary concepts.

Design concepts

Aspects :

- An aspect is a representation of a crosscutting concern.
- A crosscutting concern is some characteristic of the system that applies across many different requirements
- Consider 2 requirements, A and B. Requirement A crosscuts requirement B "if a software decomposition [refinement] has been chosen in which B cannot be satisfied without taking A into account".

Design concepts

Refactoring :

- A reorganization technique that simplifies the design (or code) of a component without changing its function or behavior.
- Removes redundancy, unused design elements, inefficient or unnecessary algorithms,poorly constructed or inappropriate data structures, or any other design failures

Design concepts

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, among others

Design concepts

Design Classes:

- A design class is a description of a set of objects that share the same responsibilities, relationships, operations, attributes, and semantics.
- 5 different types of design classes:
  - **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 :**The class identifies the attributes that are required to implement some elements of the business domain.
  - **Process classes:** implement lower-level business abstractions required to fully manage the business domain classes.
  - **Persistent classes:** represent data stores (e.g., a database) that will persist beyond the execution of the software.
  - **System classes:** implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

Design concepts

Design Classes cont..:

- 4 characteristics of a well-formed design class:
  - Complete and sufficient
  - Primitiveness
  - High cohesion **(keeping parts of a code base that are related to each other in a single place.)**
  - Low coupling **(separating unrelated parts of the code base as much as possible.)**

Design concepts

Dependency Inversion:

- The dependency inversion principle 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:

- whether software design or test case design should come first.
- Advocates of test-driven development (TDD) write tests before implementing any other code.
- "Test fast, fail fast, adjust fast."

THE DESIGN MODEL

- The design model can be viewed in two different dimensions.
  1. **Process dimension** indicates the evolution of the design model as design tasks are executed as part of the software process.
  2. **Abstraction dimension** represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refi ned iteratively.

THE DESIGN MODEL

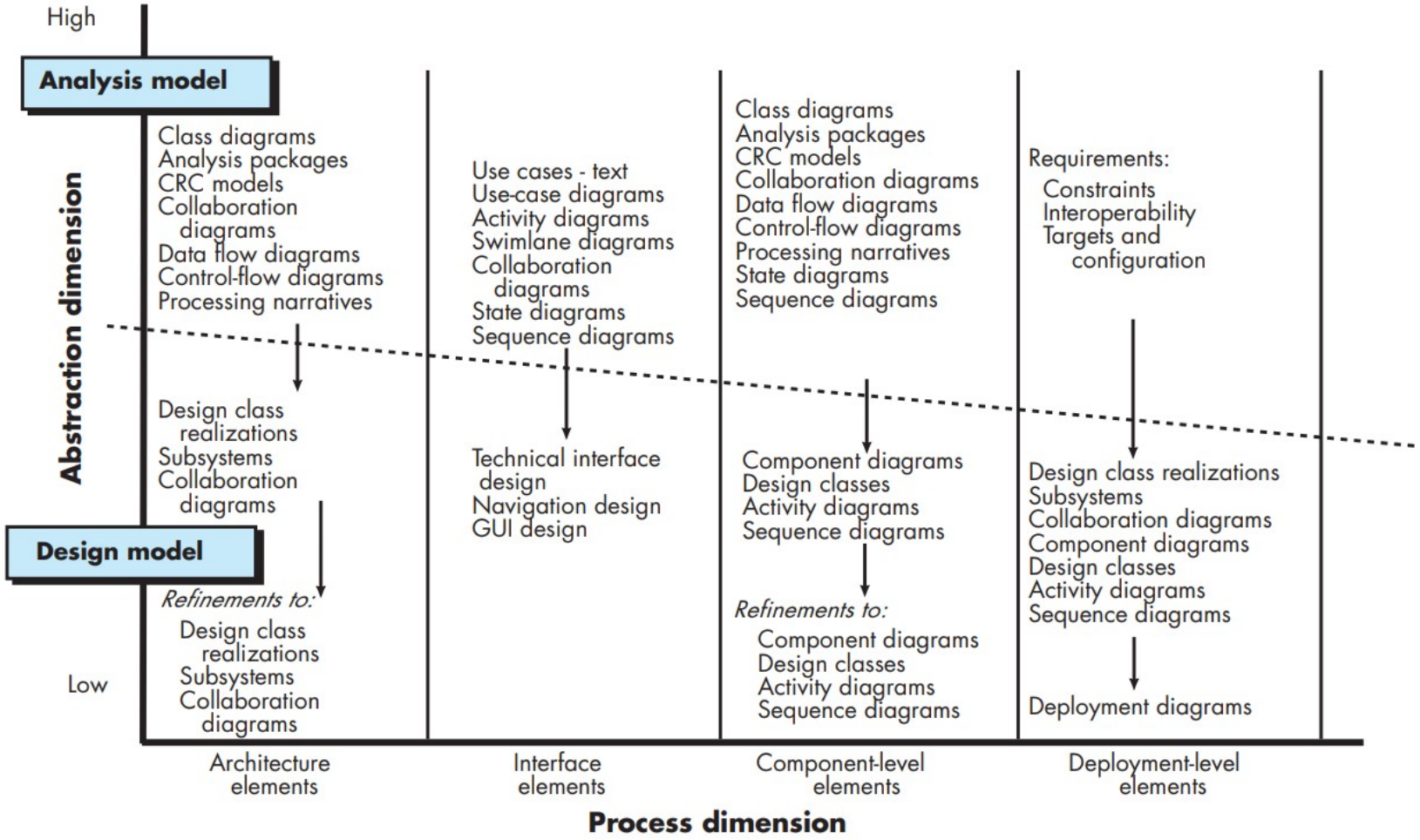


Figure 29: Dimensions of the design model

THE DESIGN MODEL

The design model has 4 major elements:

- Data Design Elements,
- Architectural Design Elements,
- Interface Design Elements, and
- Component-Level Design Elements.

THE DESIGN MODEL

The design model has following layered elements:

- Data Design Elements,
  - Creates a model of data and objects that is represented at a high level of abstraction
- Architectural Design Elements,
  - Depicts the overall layout of the software
- Interface Design Elements,
  - Tells how information flows into and out of the system and how it is communicated among the components defined as part of the architecture
  - includes the user interface, external interfaces, and internal interfaces
- Component-Level Design Elements.
  - Describes the internal details of each software component by way of data structure definitions,algorithms, and interface specifications.
- Development-Level Design Elements:
  - Deployment-level design elements indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software.

ARCHITECTURAL DESIGN

- Software Architecture,
- Architectural Styles,
- Architectural considerations,
- Architectural Design

Software Architecture,

- What is Architecture
- Why Is Architecture Important?
- Architectural Descriptions
- Architectural Decisions

What is Architecture

- The software architecture of a program or computing system is the **structure or structures of the system**, which comprise **software components**, the **externally visible properties** of those components, and the **relationships among them**
- The architecture is not the operational software. Rather, it is a representation that enables you to
  - analyze the effectiveness of the design in meeting its stated requirements,
  - consider architectural alternatives at a stage when making design changes isstill relatively easy, and
  - reduce the risks associated with the construction of the software.

Why Is Architecture Important?

- Three Key Reasons
  - The representation of software architecture **allows the communication between all stakeholder and the developer.**
  - The architecture **helps with early-stage decision-making** that impact on all software engineering work and it is the ultimate success of the system.
  - This model helps for **making further changes and adjustments**

Architectural Descriptions

- **Different stakeholders** will see an architecture from **different viewpoints** that are driven by different sets of concerns.
- An architectural description is actually **a set of work products** that **reflect different views of the system**.

Architectural Decisions:

- Each view developed as part of an architectural description **addresses a specific stakeholder concern**.
- **To develop each view** the system architect **considers a variety of alternatives** and **ultimately decides** on the specific **architectural features that best meet the concern**.

ARCHITECTURAL STYLES

- The main **aim** of architectural style is **to build a structure for all components of the system**.
- The software that is built for computer-based systems can exhibit one of the different architectural styles.
- The design categories of **architectural styles includes**:
  1. a **set of components**
  2. a **set of connectors** that enables" communication and coordination
  3. **constraints** that define **how components** can be **integrated** to form the system
  4. **Semantic models** to understand the overall properties of a system

ARCHITECTURAL STYLES

- Taxonomy of architectural styles
  - Data-centered architectures.
  - Data-flow architectures.
  - Call and return architectures.
  - Object-oriented architectures.
  - Layered architectures

Data-centered architectures.

- **A data store** (e.g., a file or database) resides **at the center** of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store.
- Data-centered architectures promote integrability. that is, existing components can be changed and new client components added to the architecture without concern about other client (because the client components operate independently)

Data-centered architectures.

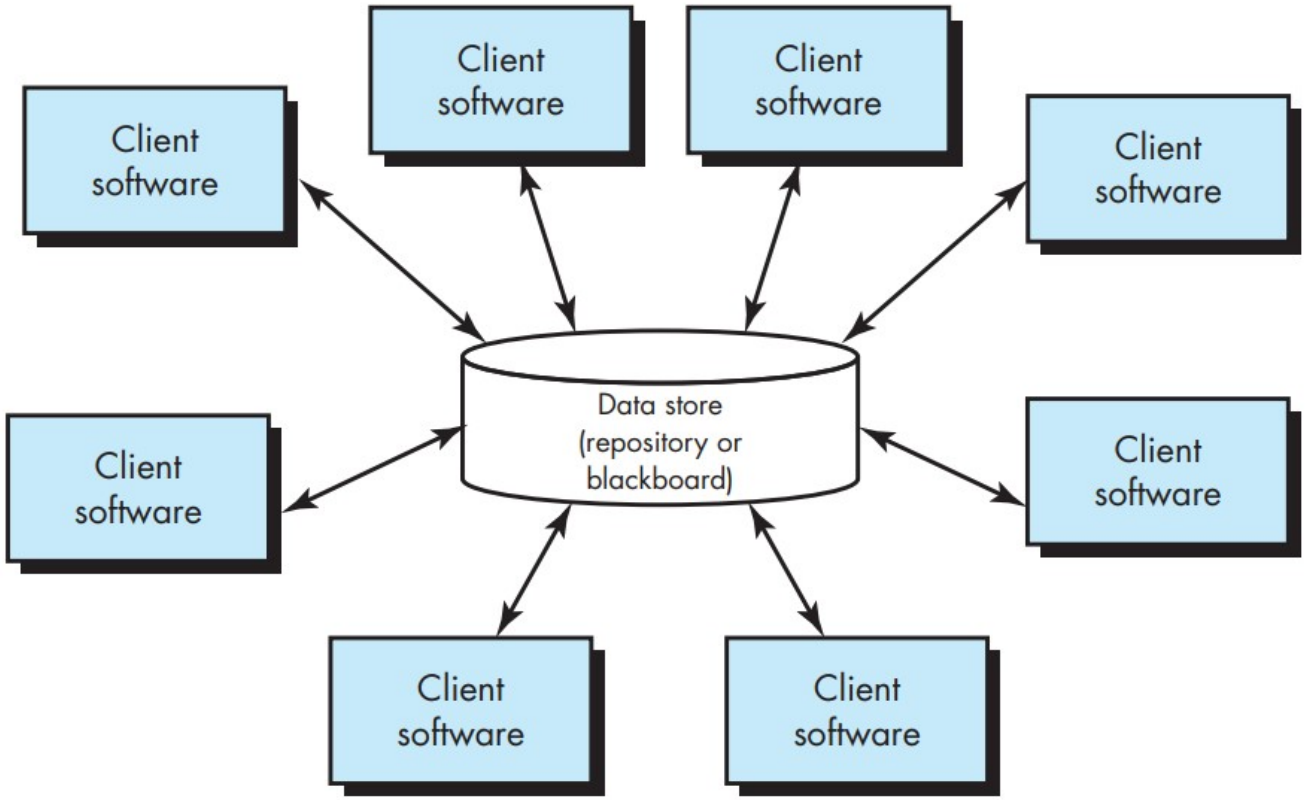


Figure 30: Data-centered architecture

Data-flow architectures.

- **Shows the flow of input data**, its **computational components** and **output data**
- Structure is **also called pipe and Filter**
- **Pipe** provides path for **flow of data**
- **Filters manipulate data** and work independent of its neighboring filter
- If data flow degenerates into a single line of transform, it is termed as batch sequential.

Data-flow architectures.

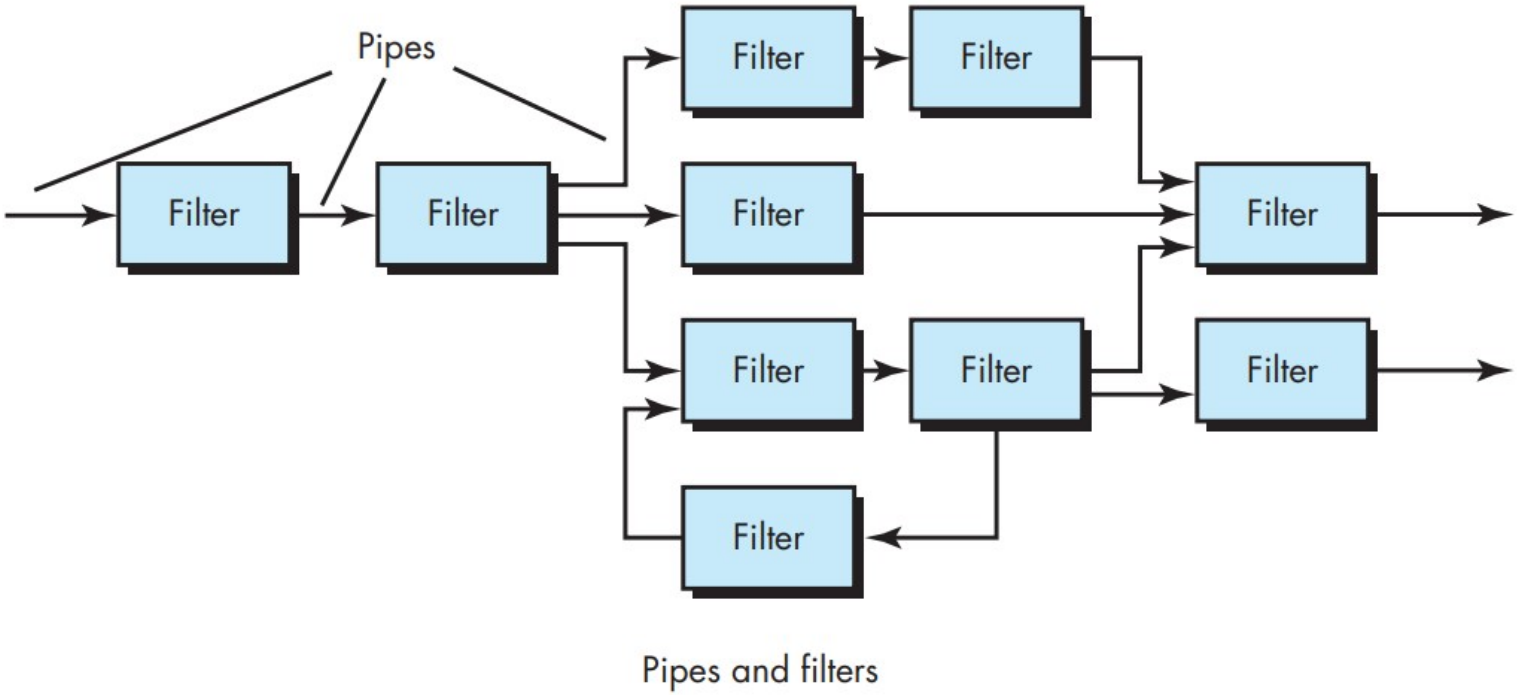


Figure 31: Data-flow architecture

Call and return architectures.

- This architectural style enables you to achieve a program structure that is relatively easy to modify and scale.
- A number of 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 that in turn may invoke still other components.
  - **Remote procedure call architectures. :** The components of a main program/subprogram architecture are distributed across multiple computers on a network.

Call and return architectures.

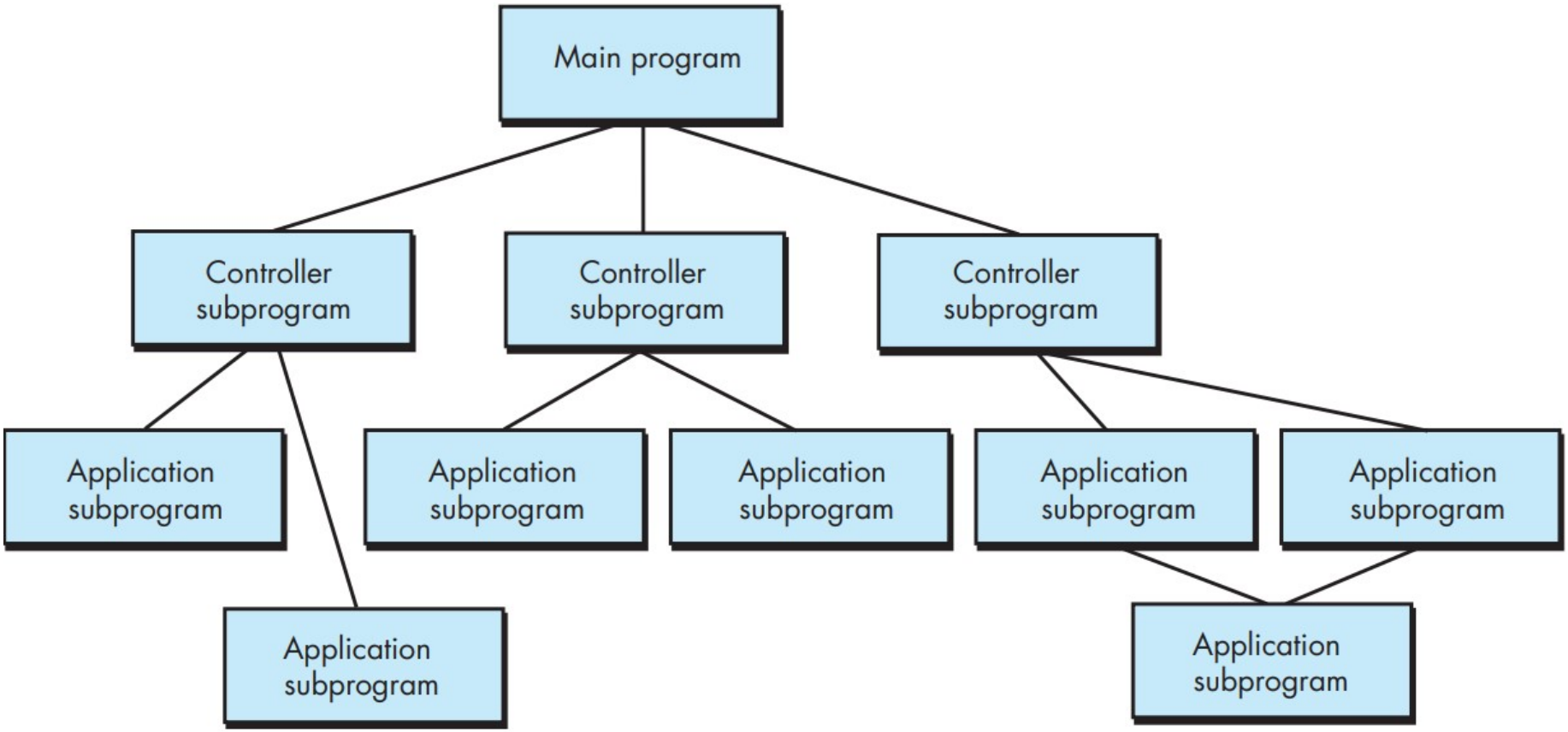


Figure 32: Main program/ subprogram architecture

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.

- A number of different layers are defined
- Inner Layer(interface with OS)
- Intermediate Layer Utility services and application function)
- Outer Layer (User interface)

Layered architectures.

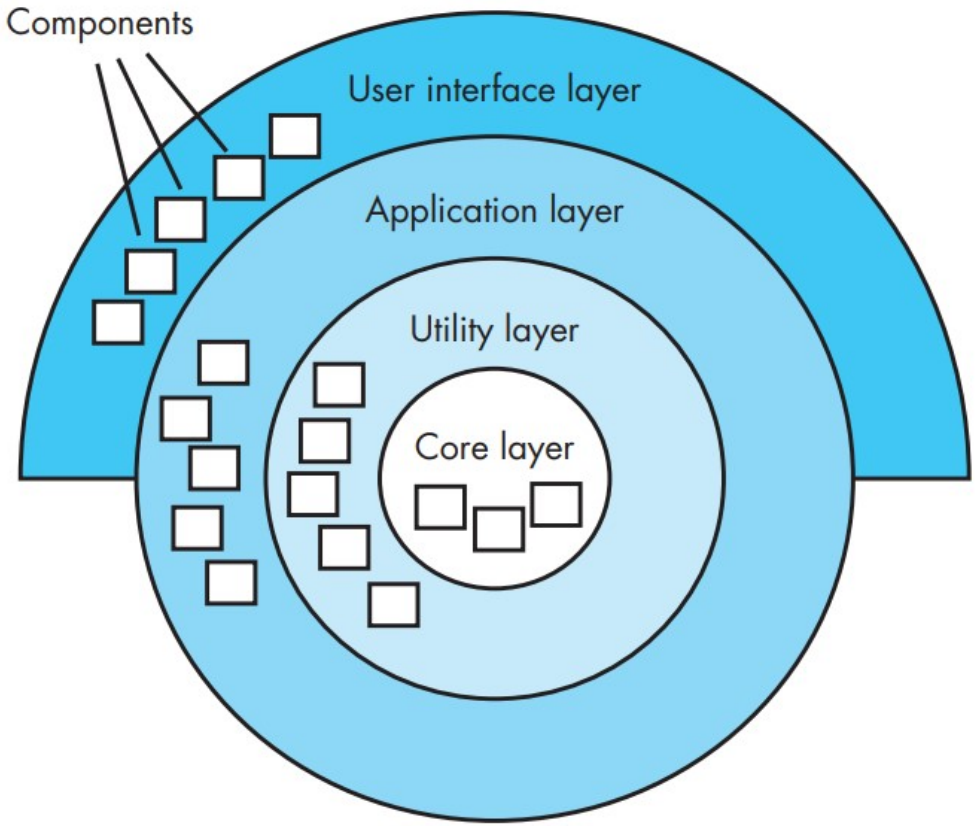


Figure 33: Layered architecture

ARCHITECTURAL CONSIDERATIONS

- Before proceeding to the final plan of constructing the structure, the firm have to mull about various aesthetic aspects to get the desired product. These aspects are known as Architectural Considerations.
- the are various architectural considerations made are as follows:
  - **Economy:**
  - **Visibility:**
  - **Symmetry:**
  - **Emergence:**
  - **Spacing:**

ARCHITECTURAL CONSIDERATIONS

- **Economy:** The architecture prepared must be economically feasible. It should not contain unnecessary structures, which may result in the increase of dead load and loss of finances in the construction.
- **Visibility:** The drawings and the design layouts prepared by the design engineer should be elucidated to the workers on site, in order to provide visibility of the final product. They should be made to ensure the ease of understanding of design.
- **Symmetry:** It is necessary to ensure that the consistency of the product, with the presented layout of the design as maintained by the design engineer.
- **Emergence:** The idea of the structure to be constructed must be very clear from the drawings prepared.
- **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.

ARCHITECTURAL DESIGN

- The design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction
- The three component of architectural design
  - **Architectural context diagram:** model how software interacts with external entities
  - **Archetypes:** are classes or patterns that represent an abstraction critical to the system
  - **Architectural components** are derived from the application domain, the infrastructure, and the interface

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.

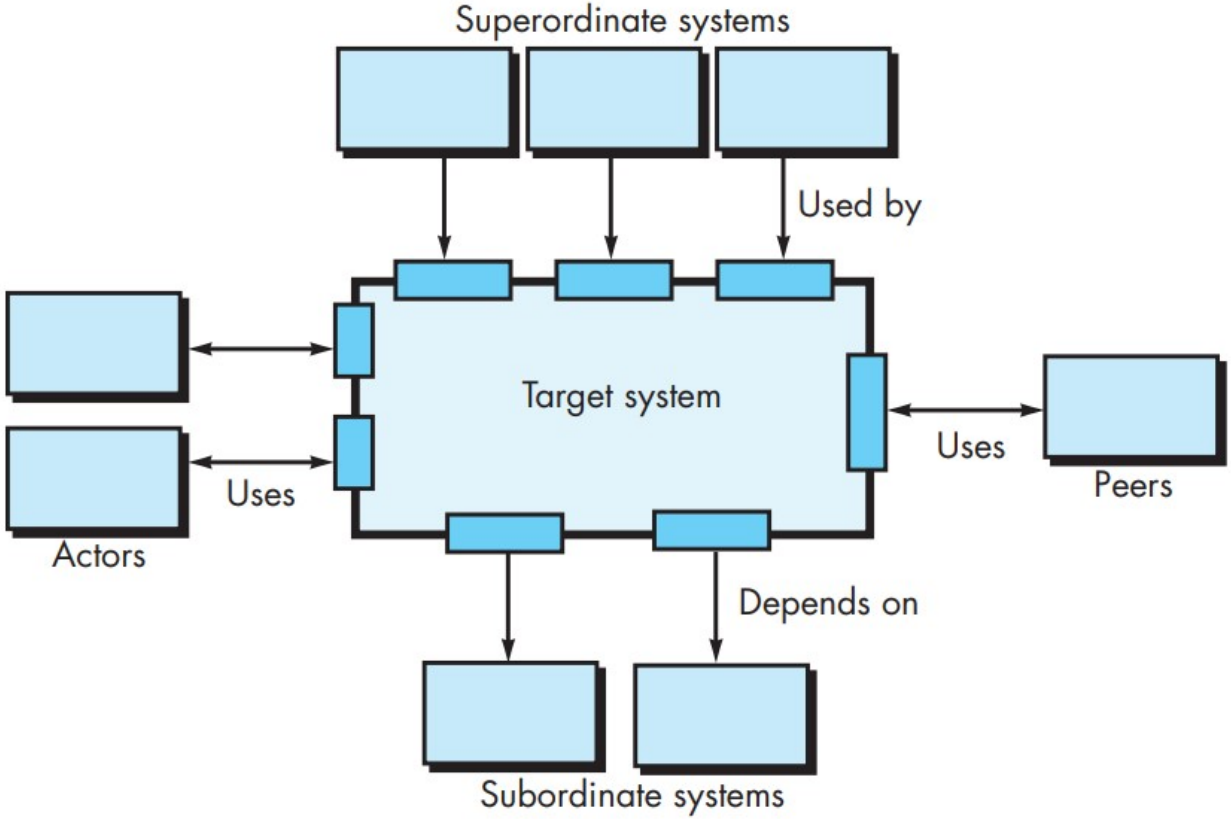


Figure 34: Architectural context diagram

Representing the System in Context

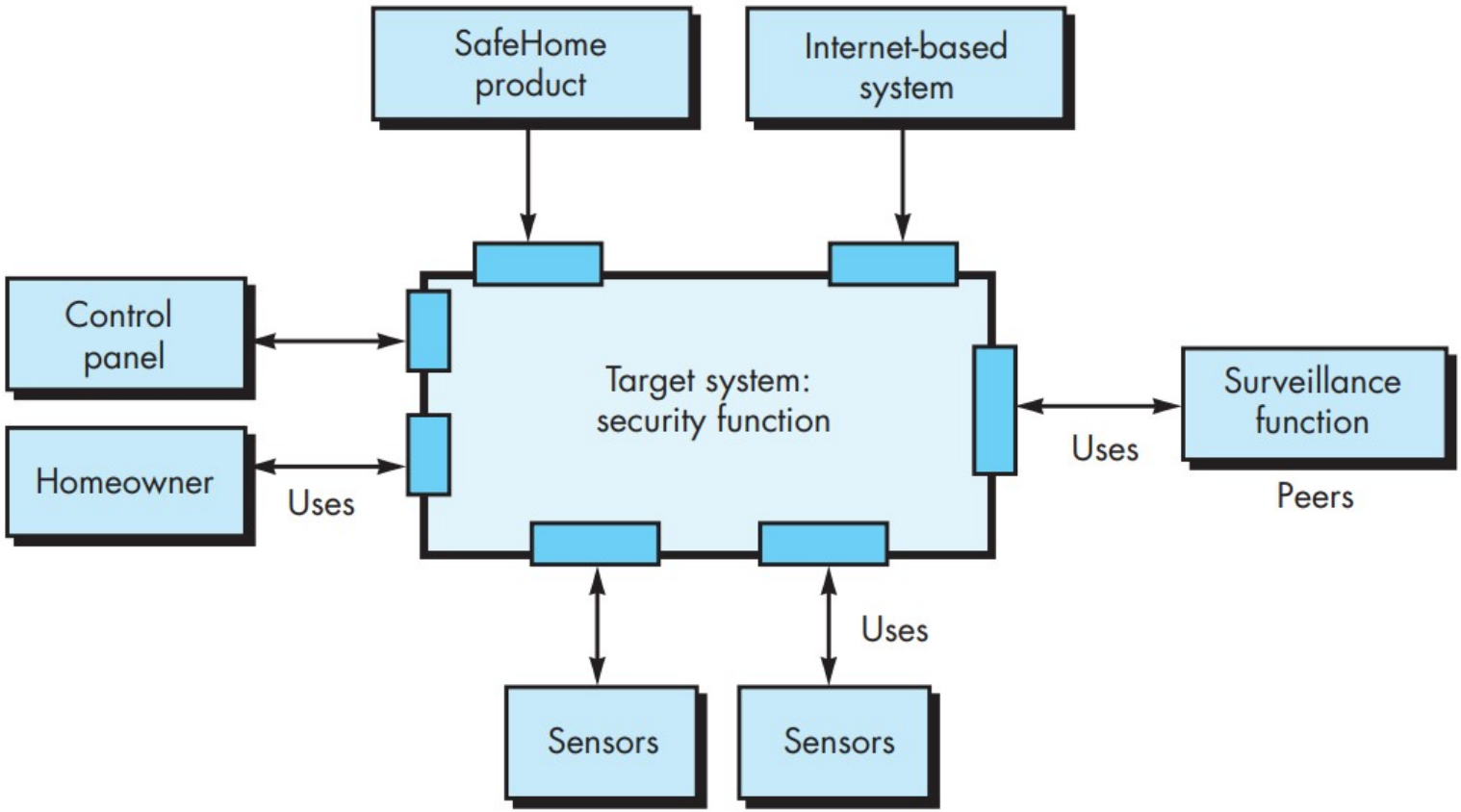


Figure 35: Architectural context diagram for the SafeHome security function

Representing the System in Context cont..

"Referring to the figure, systems that interoperate with the target system 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.

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.
- The following archetypes can be used in home security system architecture :
  - Node.
  - Detector.
  - Indicator.
  - Controller.

Defining Archetypes

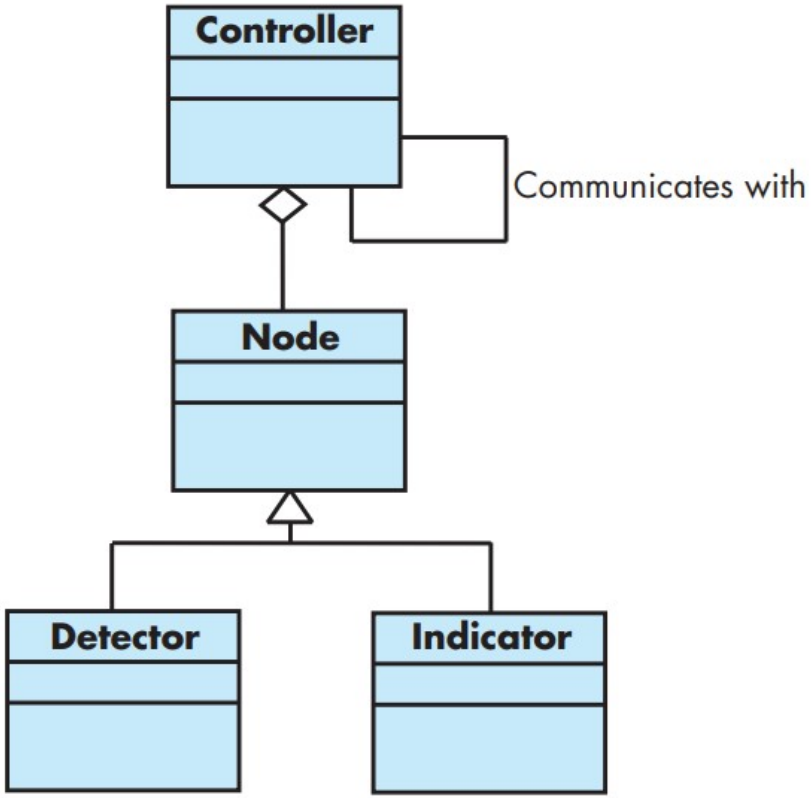


Figure 36: UML relationships for SafeHome security function archetypes

Defining Archetypes

- **Node.** Represents a cohesive collection of input and output elements of the home security function.
  - For example a node might be comprised 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.

Refining the Architecture into Components

- As the software architecture is refined into component.
- The components are derived from analysis class within application domain.
- Also, many infrastructure components are derived apart from application domain components. Eg. Memory management component.
- For eg, Based on the functionality, the following components are derived from SafeHome home security function:
  - External communication management—coordinates communication of the security function with external entities such as other Internet-based systems and external alarm notification.
  - Control panel processing—manages all control panel functionality.
  - Detector management—coordinates access to all detectors attached to the system.
  - Alarm processing—verifies and acts on all alarm conditions.
- Each of these top-level components would have to be elaborated iteratively and then positioned within the overall architecture.

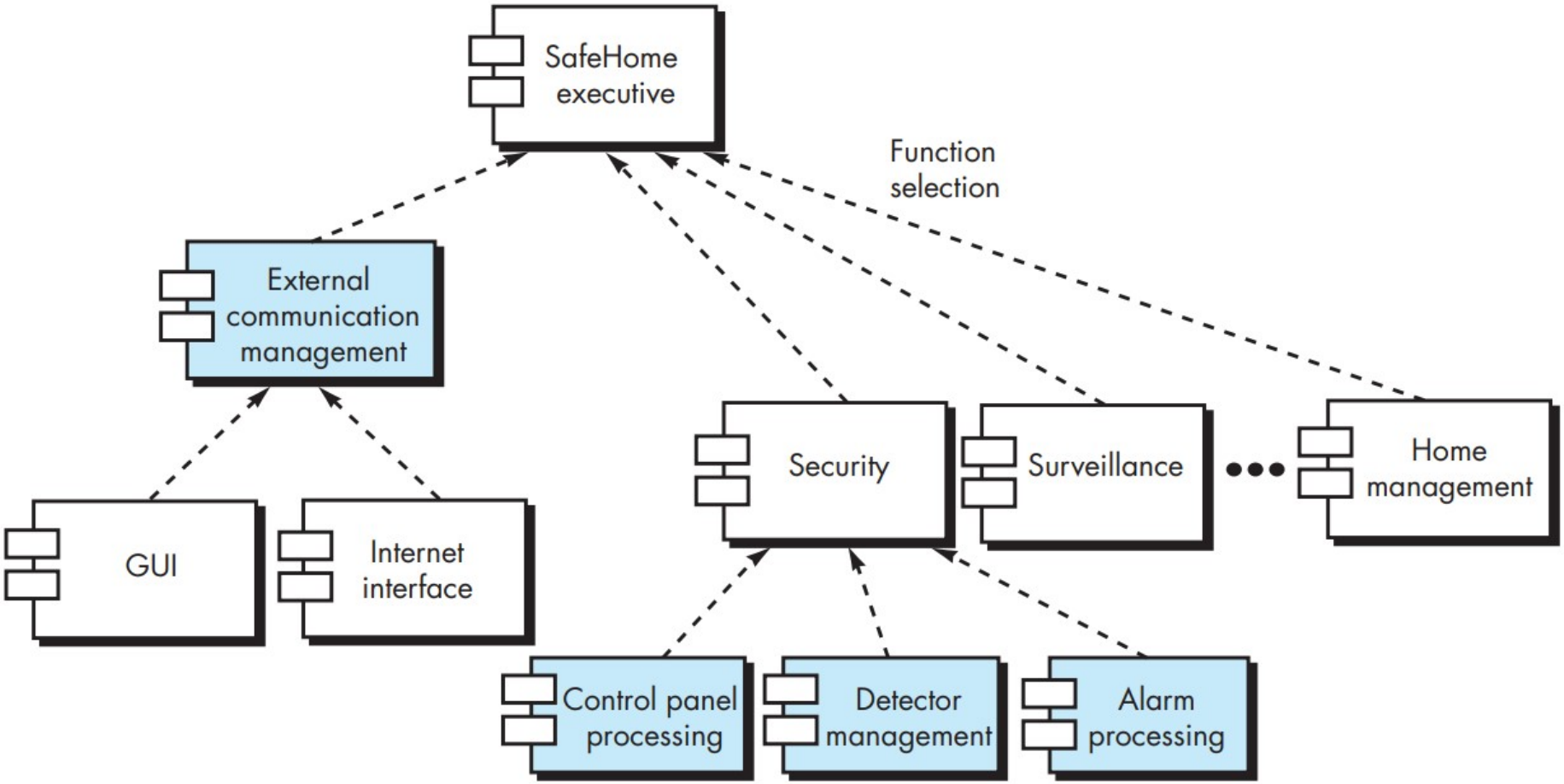
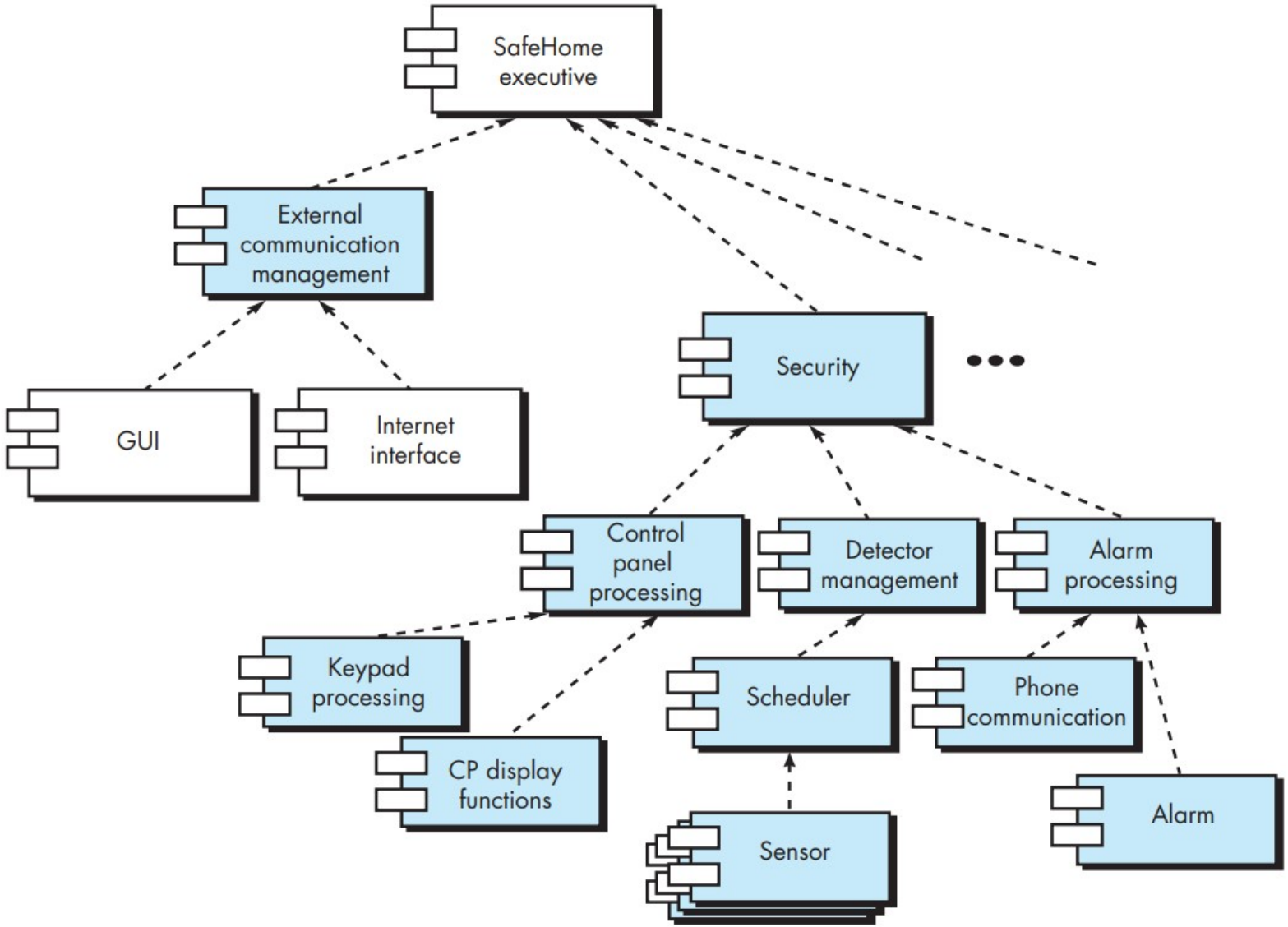


Figure 37: Overall architectural structure for SafeHome with top-level components

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.It means, again it simplify by more details.

Describing Instantiations of the System



Architectural Design for Web Apps

- WebApps are client-server applications
- 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.

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.

Component level design

- A software component is a modular building block for the computer software.
- Component is defined as a modular, deployable and replaceable part of the system which encloses the implementation and exposes a set of interfaces.

Component level design

The components has different views as follows:

- An object-oriented view
- The traditional view
- The Process related view

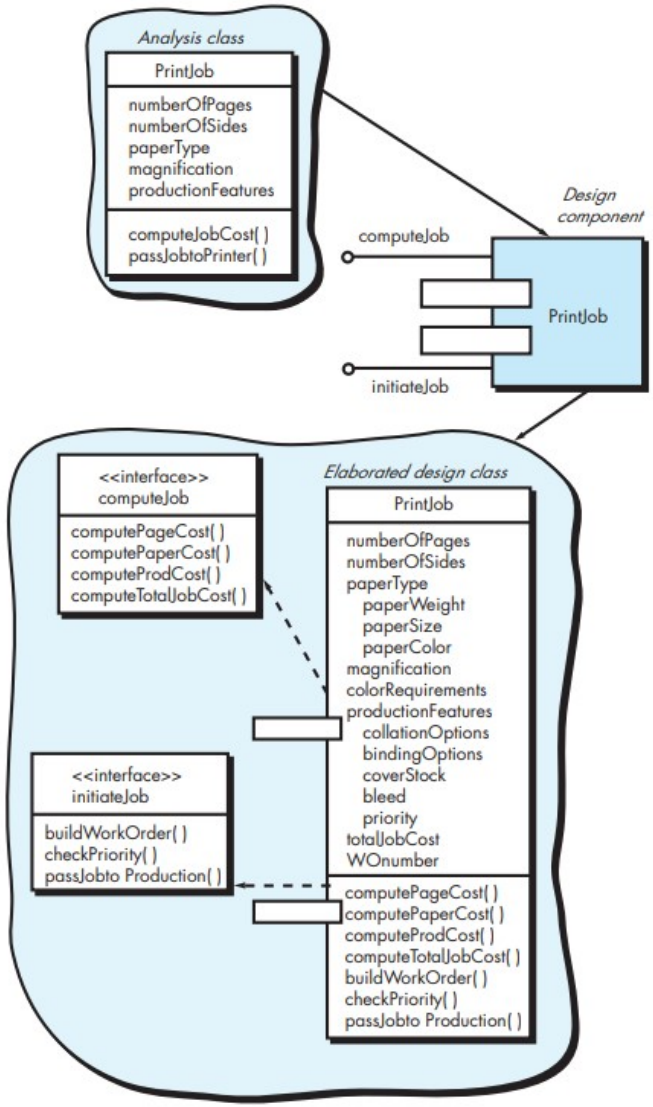
Component level design

An object-oriented view:

- An object-oriented view is a set of collaborating classes.
- The class inside a component is completely elaborated and it **consists of all the attributes and operations** which are applicable to its implementation.
- To achieve object-oriented design it elaborates analysis classes and the infrastructure classes.

Component level design

An object-oriented view:



Component level design

The traditional view

- In the context of traditional software engineering, a component is a **functional element of a program** that incorporates
  - **Processing logic**,
  - **The internal data structures** that are required to implement the processing logic,
  - **An interface** that enables the component to be invoked and data to be passed to it.
- A traditional component, also called a module,
- It resides in the software and serves **three important roles**
  - **Control component**
    - A control component coordinate is an invocation of all other problem domain components.
  - **Problem domain component**
    - A problem domain component implements a complete function which is needed by the customer.
  - **Infrastructure component**.
    - An infrastructure component is responsible for function which support the processing needed in the problem domain.

Component level design

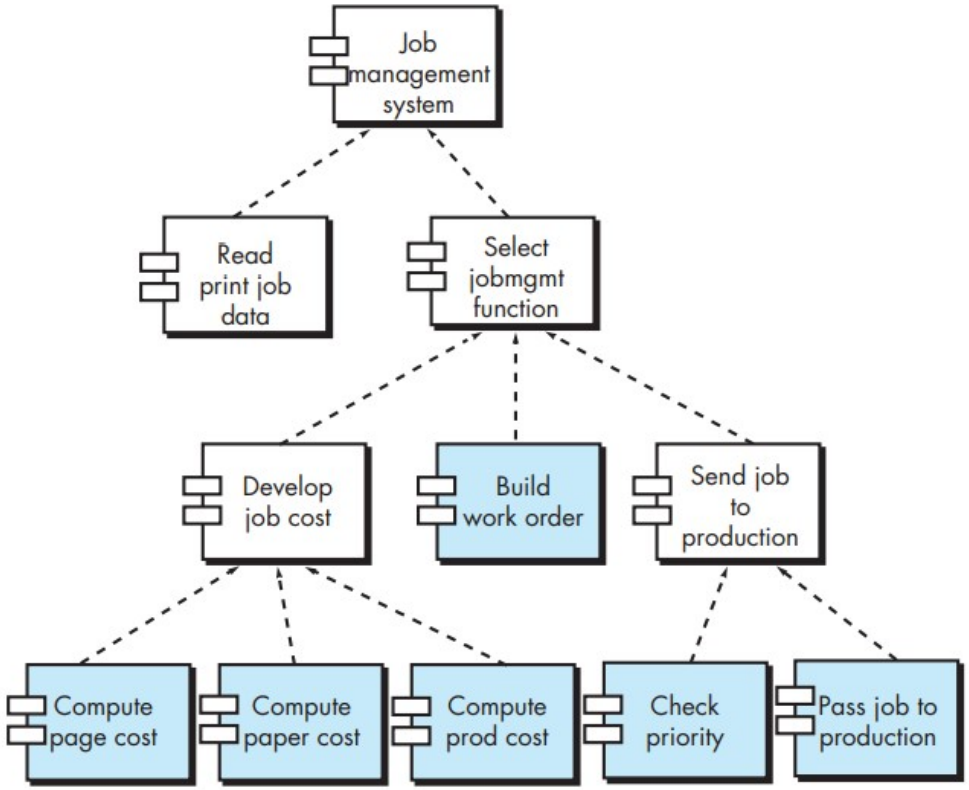


Figure 40: Structure chart for a traditional system

A Process-Related View

- This view highlights the building system out of existing components.
- The design patterns are selected from a catalog and used to populate the architecture.

Class-based design components

- The principles for class-based design component are as follows:
  1. Open Closed Principle (OCP)
  2. The Liskov Substitution Principle (LSP)
  3. Dependency Inversion Principle (DIP)
  4. The Interface Segregation Principle (ISP)
  5. The Release Reuse Equivalency Principle (REP)
  6. The common closure principle (CCP)
  7. The Common Reuse Principle (CRP)

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.

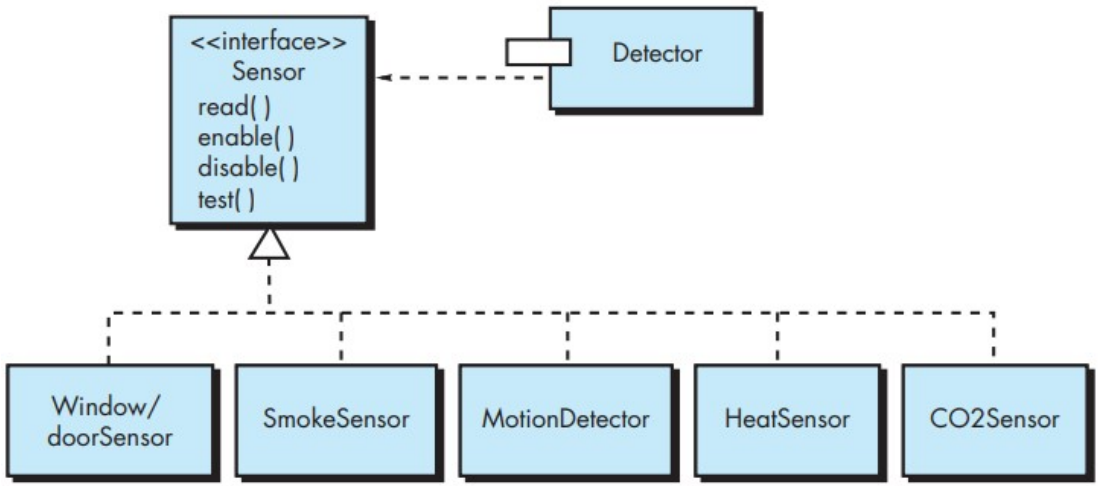


Figure 41: Following the OCP

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

Component level design

Dependency Inversion Principle (DIP)

- **"Depend on abstractions. Do not dependon concretions".**
- The more a component depends on other concrete components, the more difficult it will be to extend

The Interface Segregation (Separation) 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.

Component level design

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.

Component level design

The Common Closure Principle (CCP).

- **"Classes that change together belong together."**
- 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 willrequire modification. Thisleadstomore effective change control and release management

Component level design

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

Component-Level Design Guidelines

- In addition to the principles , a set of pragmatic (Practical) design guidelines can be applied as component-level design proceeds.
- These guidelines apply to
  - Components,
  - Interfaces
  - Dependencies and inheritance

Component level design

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

Interfaces

- Interfaces provide important information about communication and collaboration.

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).

Component level design

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.
- Levels of cohesion OR different types of cohesion
  - Functional
  - Layer
  - Communicational

Component level design

Functional

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

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.

Component level design

Coupling

- 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 to **keep coupling as low as is possible**.

Component level design

Types of Coupling

- **Content Coupling:**
  - It occurs when one component "surreptitiously modifies data that is internal to another component".
  - It violates information hiding.
- **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.

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.
  1. Identify Design Classes in Problem Domain
  2. Identify Infrastructure Design Classes
  3. Elaborate Design Classes
  4. Describe Persistent Data Sources
  5. Elaborate Behavioral Representations
  6. Elaborate Deployment Diagrams
  7. Refactor Design And Consider Alternatives

Component level design

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
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**.
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.

Component level design

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.

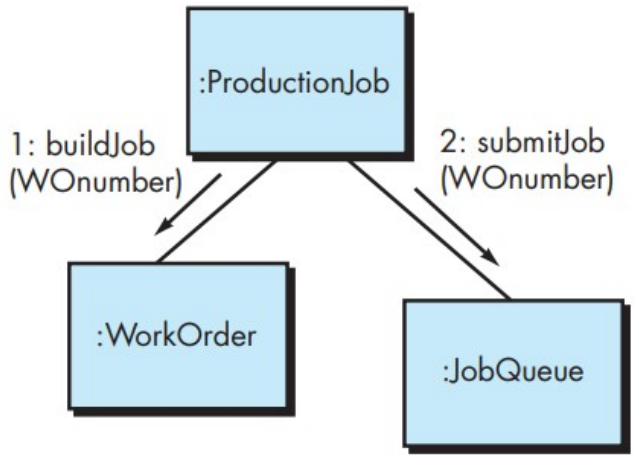


Figure 42: Collaboration diagram with messaging

Component level design

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. “.

Component level design

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.

Component level design

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.

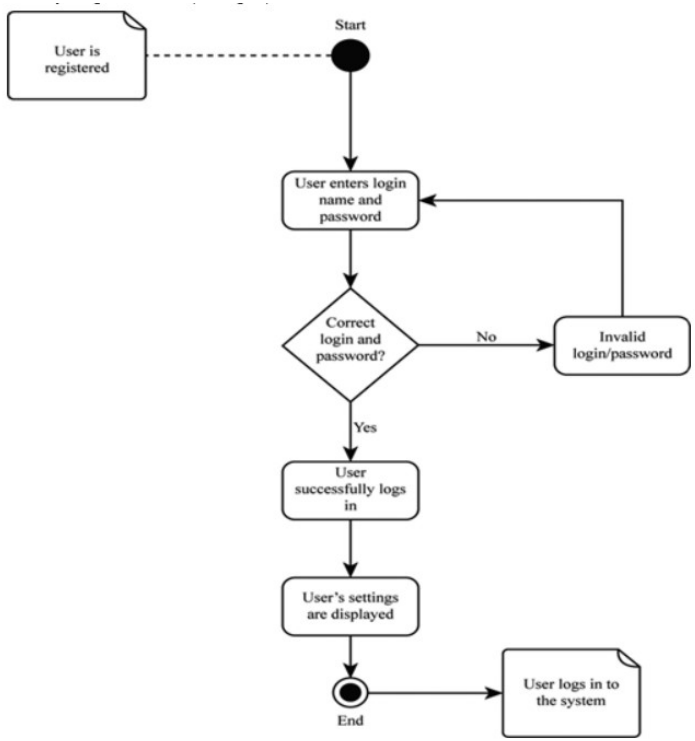


Figure 43: UML activity diagram for simple login procedure:

Component level design

4 Describe persistent data sources (databases and files) and identify the classes required to manage them.

- The persistent data sources are described (databases and files) and the classes required to manage them are identified.

Component level design

5 Develop and elaborate behavioural representations for a class or component.

- It is sometimes necessary to model the behavior of a design class.
- The behavior of the system is elaborated using a state diagram to depict the transition of states during work flow.

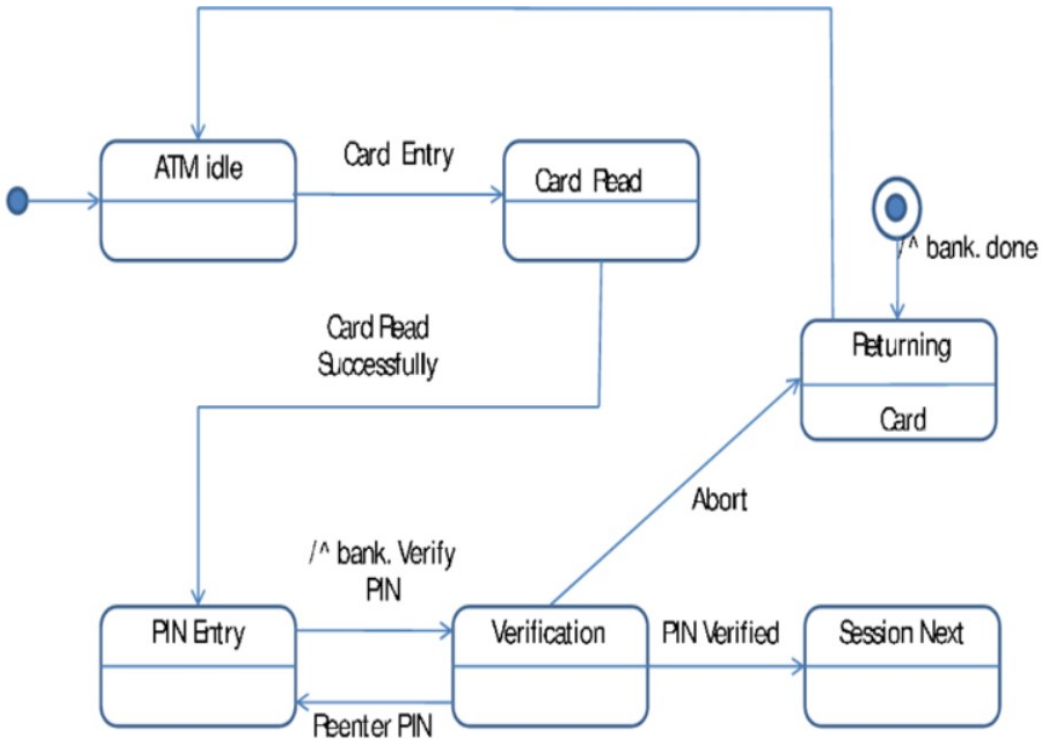


Figure 44: Statechart fragment for Simple ATM Diagram

Component level design

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.

Component level design

7 Refactor every component-level design representation and always consider alternatives

- Design is an iterative process. The first component-level model we create will not be as complete,consistent, or accurate as the nth iteration you apply to the model. It is essential to refactor as design work is conducted.

Component level design

Component Level Design for WebApps

- A WebApp component is
  - A well-defined cohesive (Interrelated) function that manipulates content or provides computational or data processing for an end user
  - 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
    - Functional design.

Component level 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.
- However, as the size and complexity (of the WebApp, content objects, and their interrelationships) grows, it may be necessary to organize content in a way that allows easier reference and design.
- For example, consider a Web-based video surveillance capability within SafeHomeAssured.com. Among many capabilities, the user can select and control any of the cameras represented as part of a floor plan, require video-capture thumbnail images from all the cameras, and display streaming video from any one camera.

Component level design

Functional Design at the Component Level

- Modern Web applications deliver increasingly sophisticated processing functions that
  - Perform localized processing to generate content and navigation capability in a dynamic fashion,
  - Provide computation or data processing capability that is appropriate for the WebApp's business domain,
  - Provide sophisticated database query and access,
  - Establish data interfaces with external corporate systems.