MODULE – 2

REQUIREMENTS ANALYSIS AND DESIGN

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

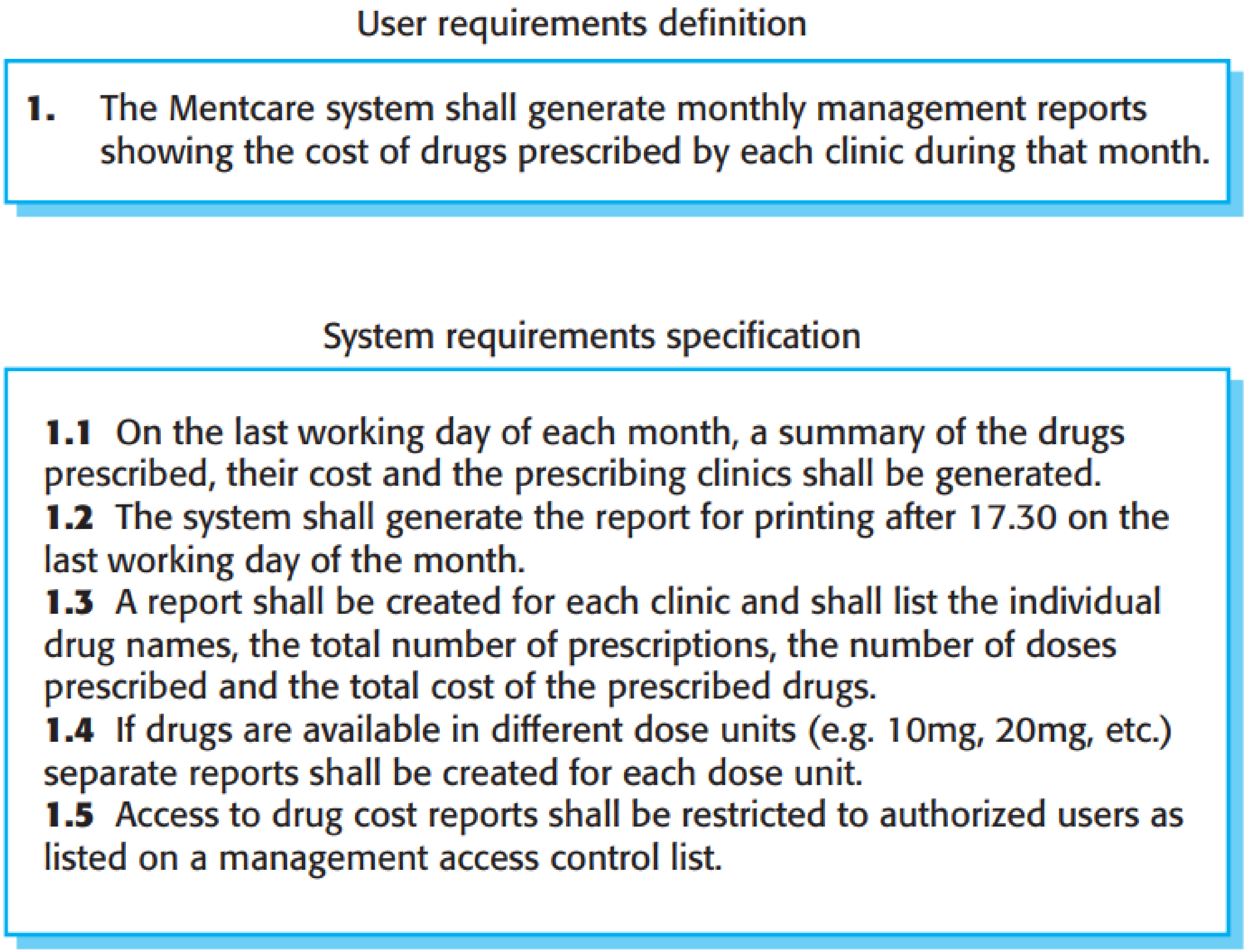
- **Requirements Engineering (RE)**
  - The process of finding out, analyzing, documenting and checking the services and constraints of a system.
  - The first stage of the software engineering process.

User Requirements

- High-level abstract requirements
- Statements, in a natural language plus diagrams, of what services the system is expected to provide to system users and the constraints under which it must operate.

System Requirements

- Detailed description of what the system should do.
- Detailed descriptions of the software system’s functions, services, and operational constraints.
- System requirements document (sometimes called a functional specification) should define exactly what is to be implemented.
- Often classified as functional or non-functional requirements.



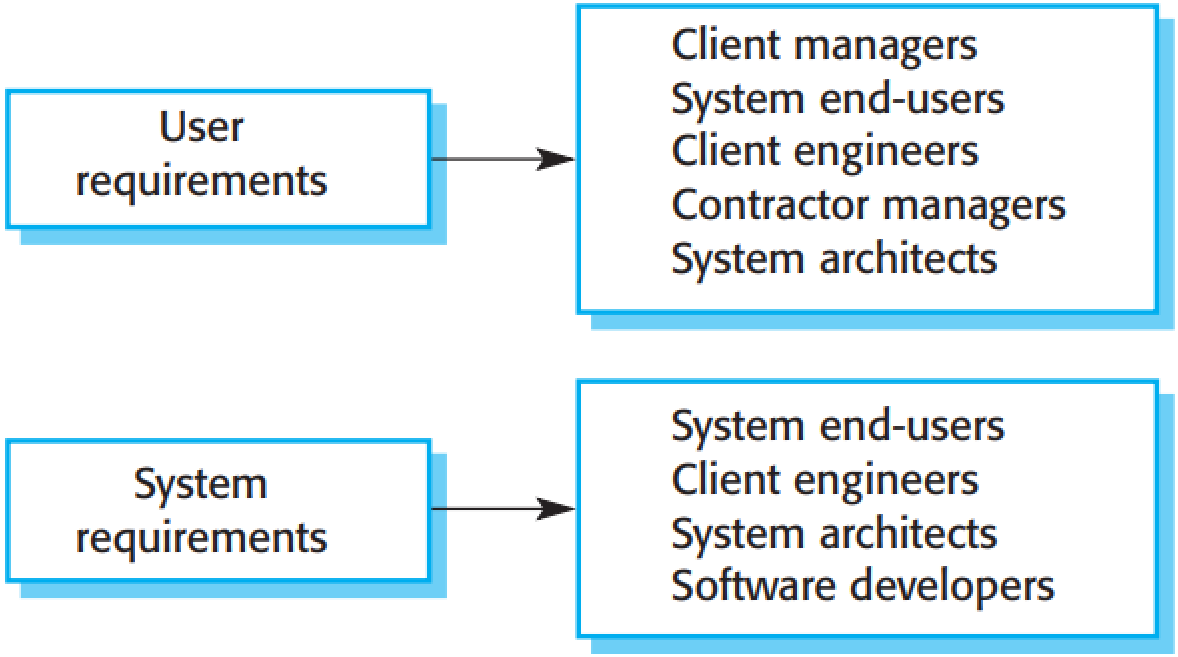
User requirements definition

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.

Fig: Mental health care patient information system (Mentcare) shows how a user requirement may be expanded into several system requirements



User requirements

Client managers
System end-users
Client engineers
Contractor managers
System architects

System requirements

System end-users
Client engineers
System architects
Software developers

Fig: Readers of different types of requirements specification

Functional Requirements

- Statements of services the system should provide, how the system should react to particular inputs, and how the system should behave in particular situations.
- Explicitly state what the system should not do.

Non-functional Requirements

- Constraints on the services or functions offered by the system.
- Include timing constraints, constraints on the development process, and constraints imposed by standards.

Functional Requirements

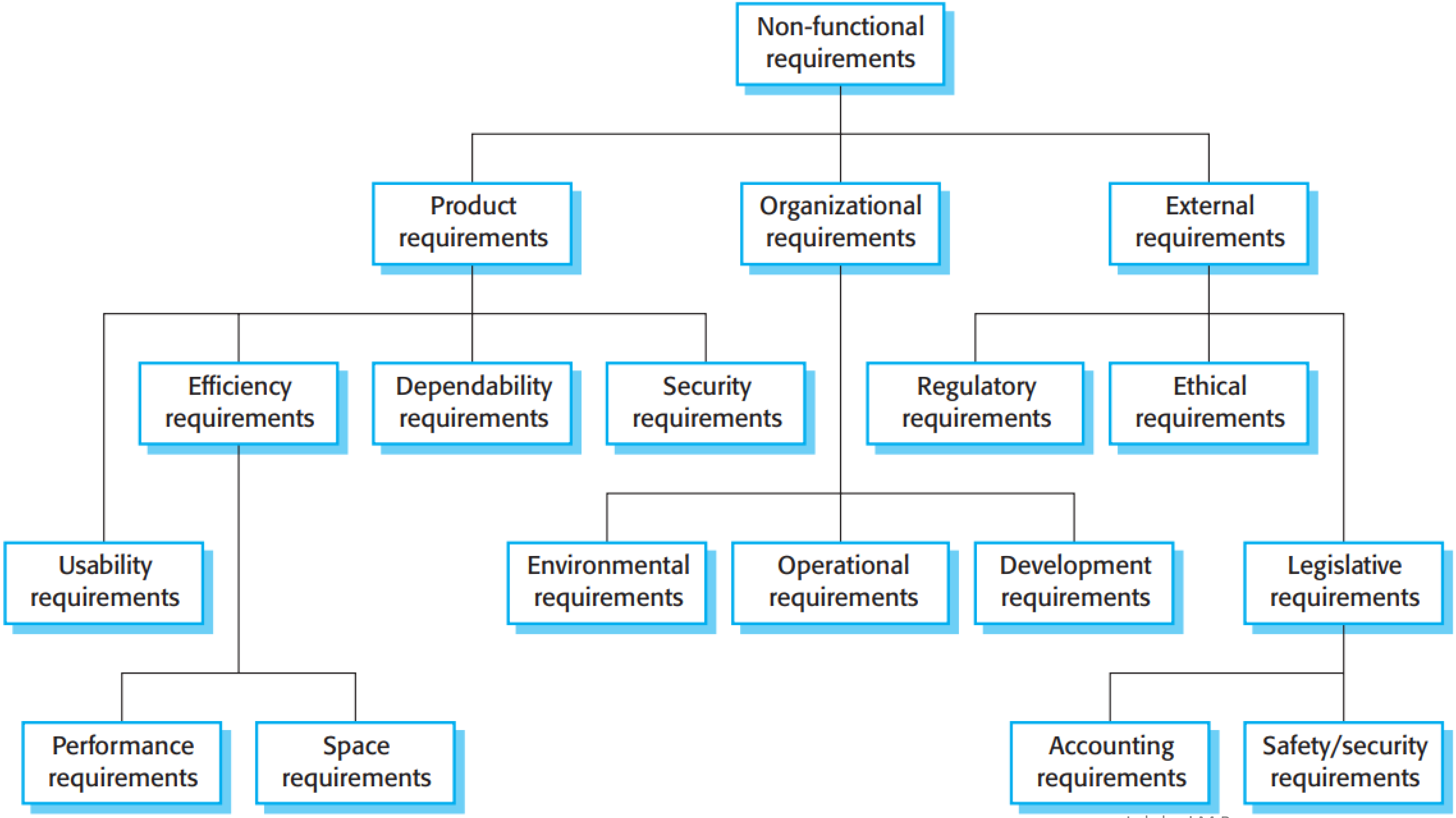
- Describe what the system should do.
- Depends on the type of software being developed, the expected users of the software, and the general approach taken by the organization when writing requirements.
- When expressed as user requirements, it should be written in natural language so that system users and managers can understand them.
- Functional system requirements are written for system developers.

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.

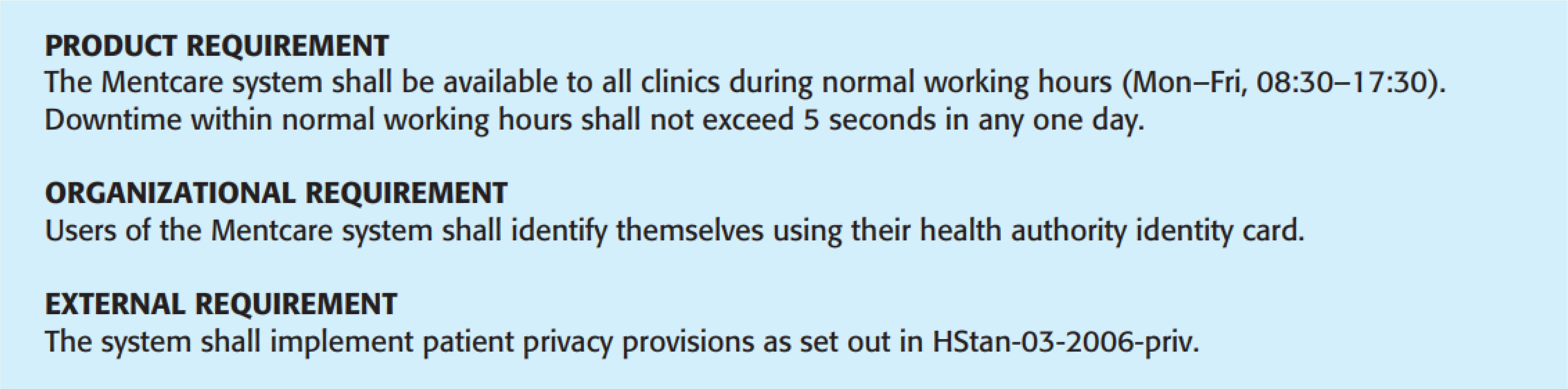
Non- functional Requirements

- Usually specify or constrain characteristics of the system as a whole.
- Often more critical than individual functional requirements.
- Failing to meet a non-functional requirement can mean that the whole system is unusable.
- Arise through user needs because of budget constraints, organizational policies, the need for interoperability with other software or hardware systems, or external factors such as safety regulations or privacy legislation.
- 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.


Non-functional requirements
Product requirements
Organizational requirements
External requirements
Efficiency requirements
Dependability requirements
Security requirements
Regulatory requirements
Ethical requirements
Usability requirements
Environmental requirements
Operational requirements
Development requirements
Legislative requirements
Performance requirements
Space requirements
Accounting requirements
Safety/security requirements

Lakshmi M B

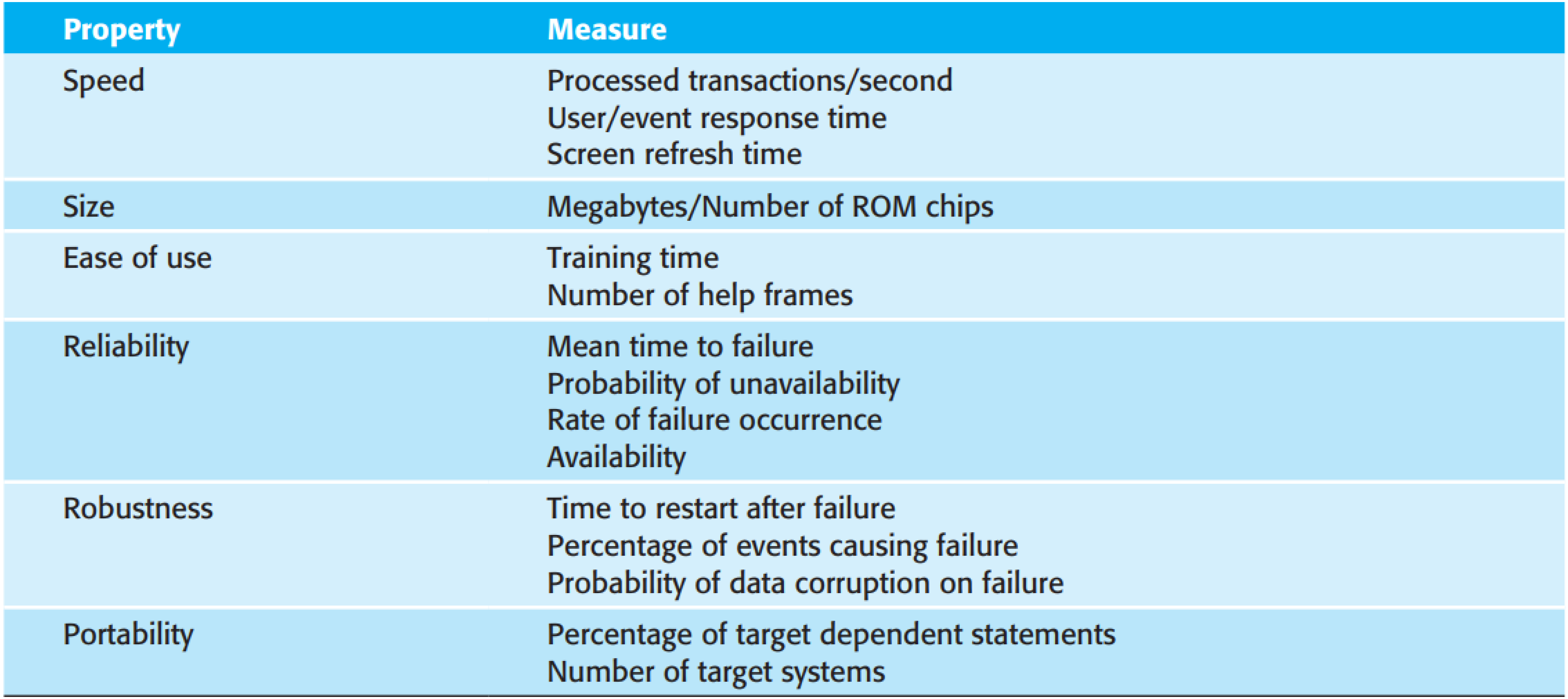
1. **Product requirements** → specify or constrain the runtime behavior of the software.
   - Examples include how fast the system must execute, how much memory it requires, etc.
2. **Organizational requirements** → broad system requirements derived from policies and procedures in the customer’s and developer’s organizations.
   - Examples include operational process requirements that define how the system will be used; development process requirements that specify the programming language; the development environment or process standards to be used; and environmental requirements that specify the operating environment of the system.
3. **External requirements** → derived from factors external to the system and its development process.
   - Example include regulatory requirements that set out what must be done for the system to be approved for use by a regulator, such as a nuclear safety authority; legislative requirements that must be followed to ensure that the system operates within the law; and ethical requirements that ensure that the system will be acceptable to its users and the general public.



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.



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

Fig: Metrics for specifying non-functional requirements

Requirements Engineering Processes

- 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.

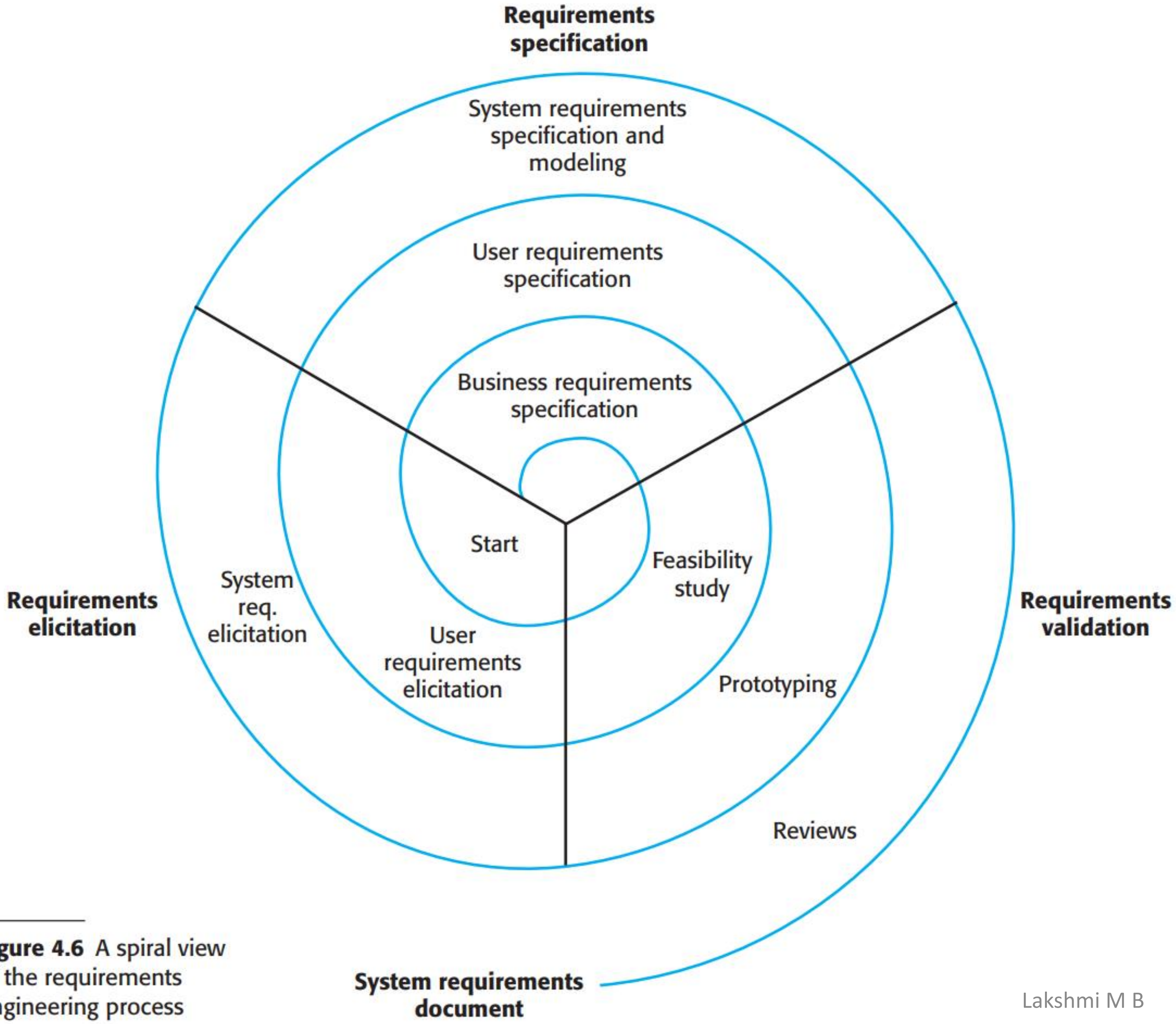

Requirements specification
System requirements specification and modeling
User requirements specification
Business requirements specification
Start
Feasibility study
Requirements elicitation
System req. elicitation
User requirements elicitation
Prototyping
Requirements validation
Reviews
System requirements document

Figure 4.6 A spiral view of the requirements engineering process

Requirements Elicitation

- An iterative process that can be represented as a spiral of activities—requirements discovery, requirements classification and organization, requirements negotiation, and requirements documentation.
- Aim → to understand the work that stakeholders do and how they might use a new system to help support that work.
- 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, etc.

- Difficult process due to several reasons:
  1. Stakeholders often don’t know what they want from a computer system except in the most general terms.
  2. Stakeholders in a system naturally express requirements in their own terms and with implicit knowledge of their own work.
  3. Different stakeholders, with diverse requirements, may express their requirements in different ways.
  4. Political factors may influence the requirements of a system.
  5. 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.

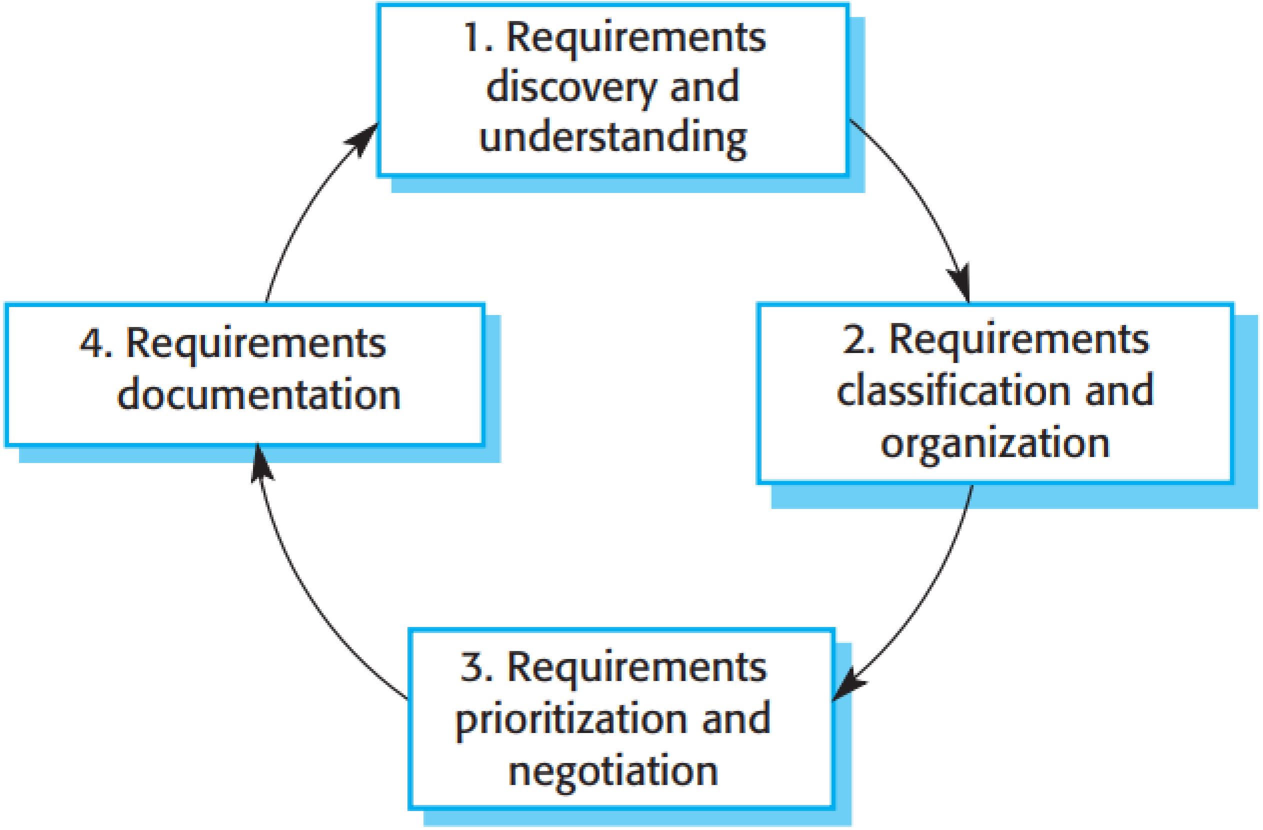

1. Requirements discovery and understanding
2. Requirements classification and organization
3. Requirements prioritization and negotiation
4. Requirements documentation

Fig: The requirements elicitation and analysis process

1. **Requirements discovery and understanding →** interacting with stakeholders of the system to discover their requirements.
2. **Requirements classification and organization →** takes the unstructured collection of requirements, groups related requirements and organizes them into coherent clusters.
3. **Requirements prioritization and negotiation →** prioritizing requirements and finding and resolving requirements conflicts through negotiation.
4. **Requirements documentation →** The requirements are documented and input into the next round of the spiral.

Requirements Elicitation Techniques

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.

1. Interviewing

- Interviews may be of two types:
  1. **Closed interviews** → where the stakeholder answers a predefined set of questions.
  2. **Open interviews** → in which there is no predefined agenda.

- To be an effective interviewer, you should bear two things in mind:
  1. You should be open-minded, avoid preconceived ideas about the requirements, and willing to listen to stakeholders.
  2. 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.

2. Ethnography

- An observational technique that can be used to understand operational processes and help derive requirements for software to support these processes.
- Value → it helps discover implicit system requirements that reflect the actual ways that people work, rather than the formal processes defined by the organization.
- 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.
    2. Requirements derived from cooperation and awareness of other people’s activities.
- Ethnography can be combined with the development of a system prototype.

Stories and Scenarios

- **Stories →** written as narrative text and present a high-level description of system use. The advantage of stories is that everyone can easily relate to them.
- **Scenarios →** usually structured with specific information collected such as inputs and outputs. Theses are descriptions of example user interaction sessions. A scenario starts with an outline of the interaction. During the elicitation process, details are added to create a complete description of that interaction. A scenario may include:
    1. A description of what the system and users expect when the scenario starts.
    2. A description of the normal flow of events in the scenario.
    3. A description of what can go wrong and how resulting problems can be handled.
    4. Information about other activities that might be going on at the same time.
    5. A description of the system state when the scenario ends.

Requirements Validation

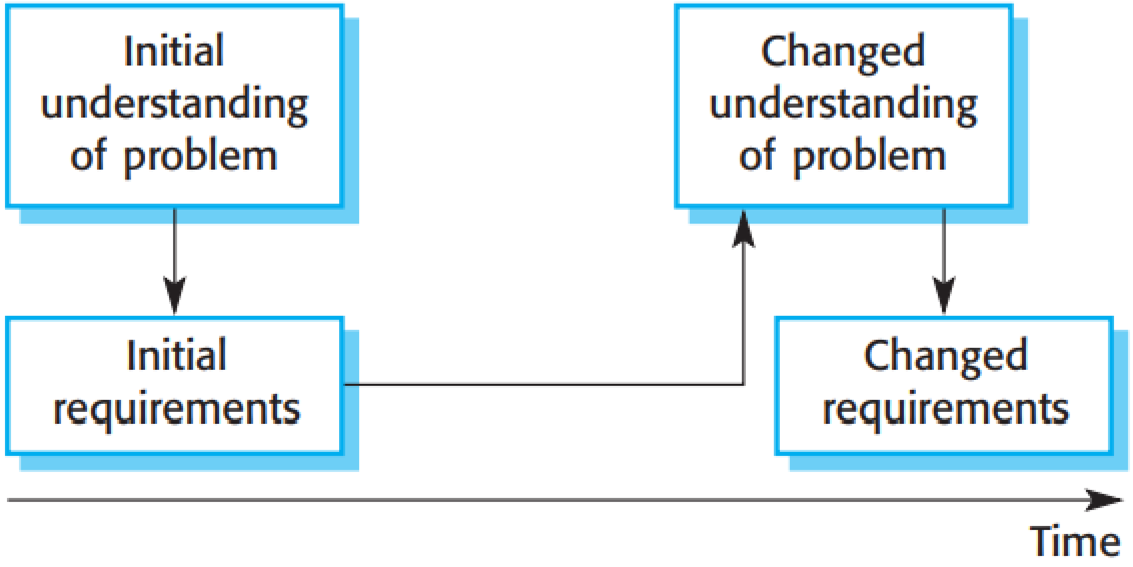
- The process of checking that requirements define the system that the customer really wants.
- Overlaps with elicitation and analysis as it is concerned with finding problems with the requirements.

- Different types of checks should be carried out on the requirements in the requirements document:
  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 techniques that are used either individually or in conjunction with one another:
    1. **Requirements reviews** → the requirements are analyzed systematically by a team of reviewers who check for errors and inconsistencies.
    2. **Prototyping** → 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.
    3. **Test-case generation** → requirements should be testable. Developing tests from the user requirements before any code is written is an integral part of test-driven development.

REQUIREMENTS CHANGE

- One reason → systems are often developed to address problems that cannot be completely defined.


Initial understanding of problem
Initial requirements
Changed understanding of problem
Changed requirements
Time

- Most changes to system requirements arise because of changes to the business environment of the system:
  1. The business and technical environment of the system always changes after installation.
  2. Requirements imposed by system customers because of organizational and budgetary constraints may conflict with end-user requirements.
  3. Priorities given to different requirements of diverse stakeholders may be conflicting or contradictory.

- As requirements are evolving, you need to keep track of individual requirements and maintain links between dependent requirements so that you can assess the impact of requirements changes. This formal process for making change proposals and linking these to system requirements is known as **requirements management.**
- It is often better for an independent authority, who can balance the needs of all stakeholders, to decide on the changes that should be accepted.

Requirements management planning

- Concerned with establishing how a set of evolving requirements will be managed.
- Issues that have to be decided:
  1. **Requirements identification** → each requirement must be uniquely identified.
  2. **A change management process** → the set of activities that assess the impact and cost of changes.
  3. **Traceability policies** → defines the relationships between each requirement and between the requirements and the system design that should be recorded. Also define how these records should be maintained.
  4. **Tool support** → range from specialist requirements management systems to shared spreadsheets and simple database systems.

- Requirements management needs automated support, and the software tools for this should be chosen during the planning phase. You need tool support for:

1. **Requirements storage** → requirements should be maintained in a secure, managed data store that is accessible to everyone involved in the requirements engineering process.
2. **Change management** → simplified if active tool support is available.
3. **Traceability management** → tool support for traceability allows related requirements to be discovered.

Requirements change management

- Should be applied to all proposed changes to a system’s requirements after the requirements document has been approved.
- It is essential because you need to decide if the benefits of implementing new requirements are justified by the costs of implementation.
- The advantage of using a formal process for change management is that all change proposals are treated consistently and changes to the requirements document are made in a controlled way.

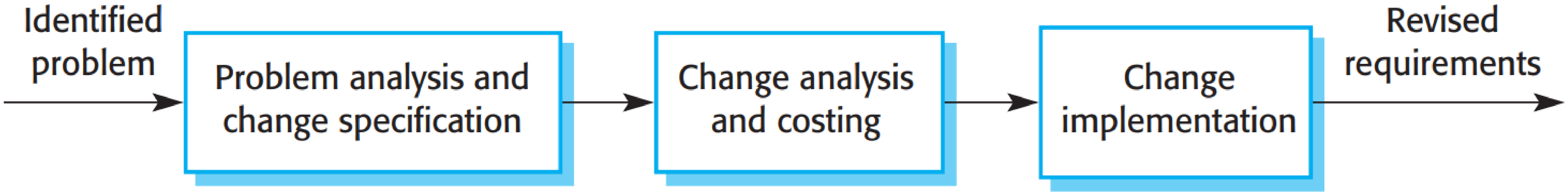
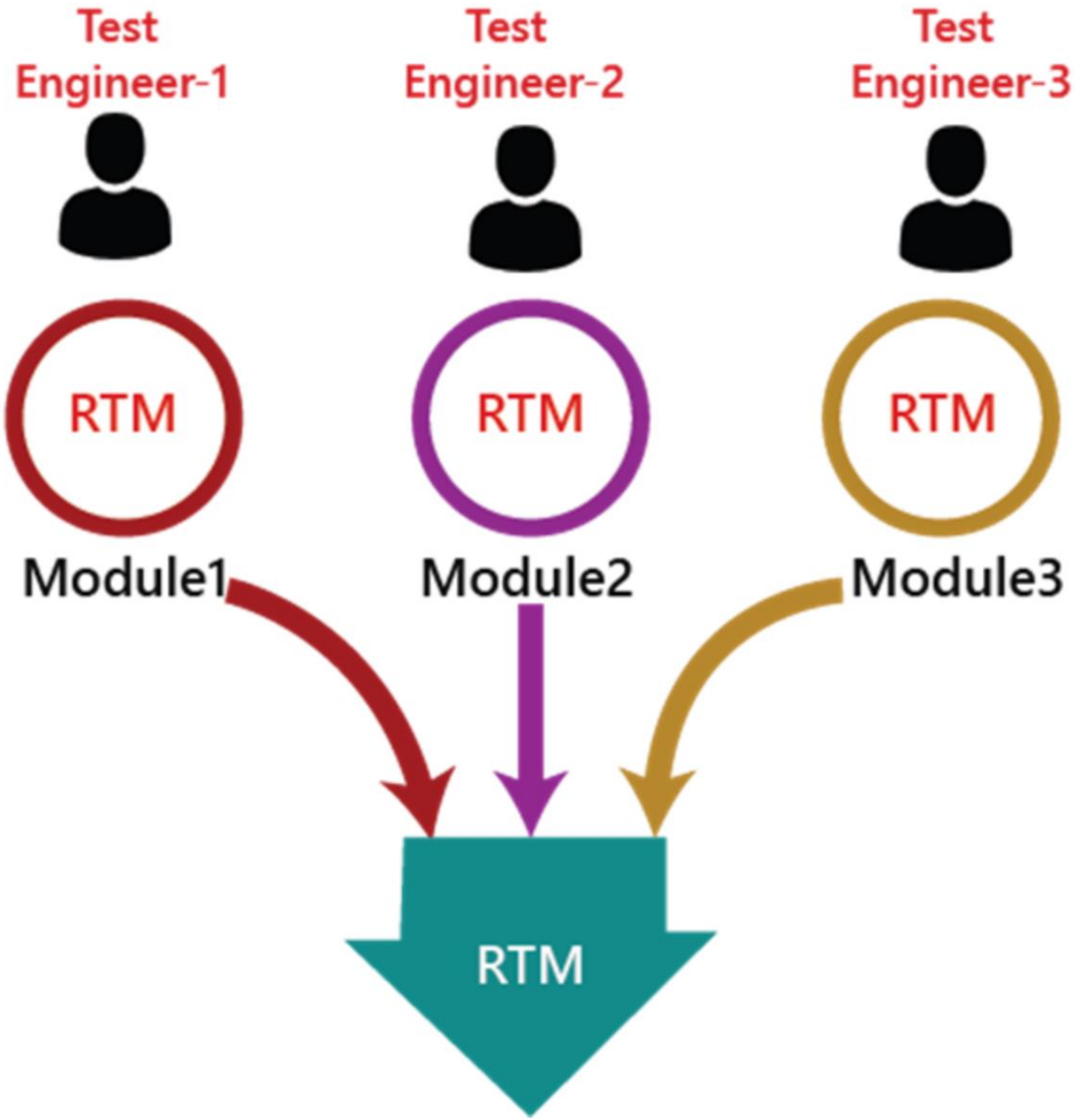

Identified problem
Problem analysis and change specification
Change analysis and costing
Change implementation
Revised requirements

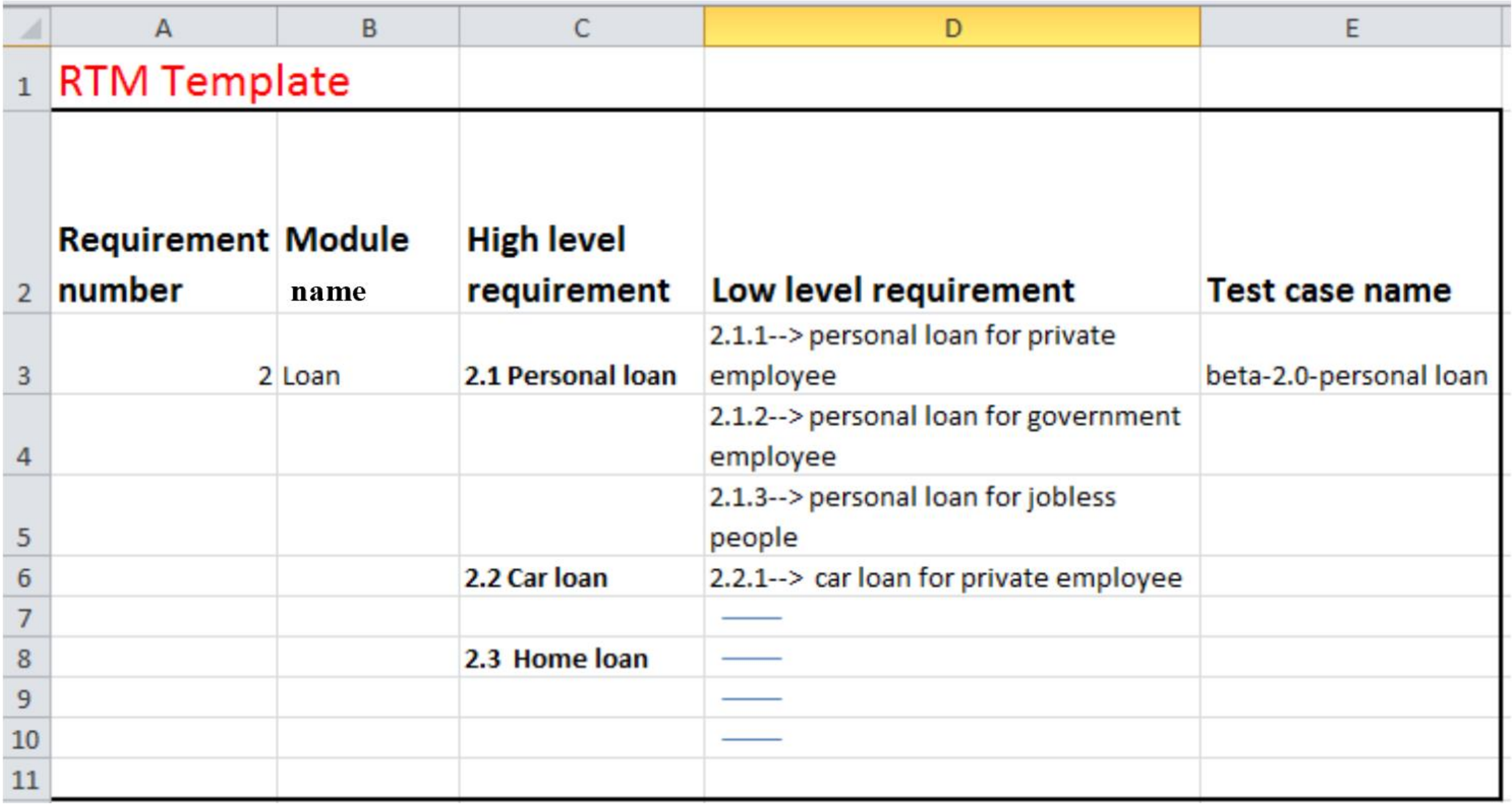
Figure 4.19
Requirements change management

- 3 principal stages to a change management process:
  1. **Problem analysis and change specification** → starts with an identified requirements problem or, sometimes, with a specific change proposal. It is then analyzed to check whether it is valid. This analysis is fed back to the change requestor who may respond with a more specific requirements change proposal, or decide to withdraw the request.
  2. **Change analysis and costing** → assessed using traceability information and general knowledge of the system requirements. The cost of making the change is estimated in terms of modifications to the requirements document. Once this analysis is completed, a decision is made as to whether or not to proceed with the requirements change.
  3. **Change implementation** → The requirements document and, the system design and implementation, are modified. Organize the requirements document so that changes can be done without extensive rewriting or reorganization. Changeability in documents is achieved by minimizing external references and making the document sections as modular as possible. Thus, individual sections can be changed and replaced without affecting other parts of the document.

TRACEABILITY MATRIX

- A table type document that is used in the development of software application to trace requirements.
- It can be used for both forward (from Requirements to Design or Coding) and backward (from Coding to Requirements) tracing.
- It is also known as **Requirement Traceability Matrix (RTM) or Cross Reference Matrix (CRM).**
- It is prepared before the test execution process to ensure that every requirement is covered in the form of a Test case so that we don't miss out any testing.
- We map all the requirements and corresponding test cases to ensure that we have written all the test cases for each condition.

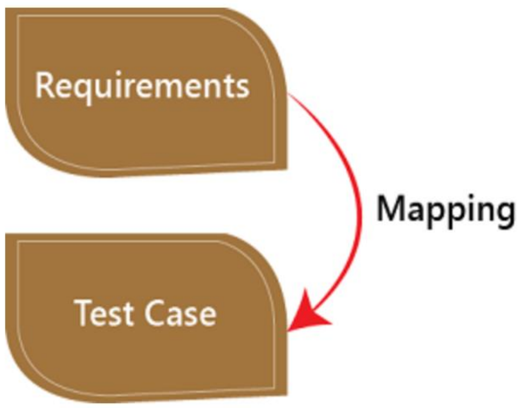
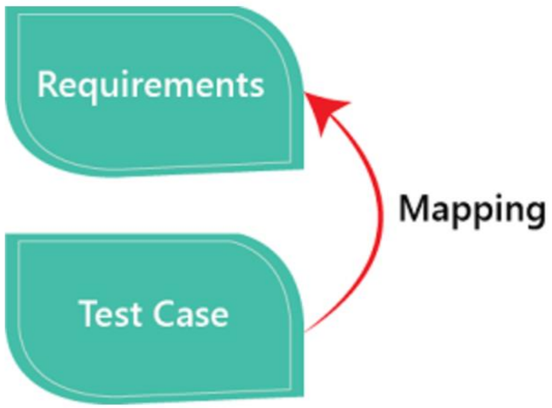
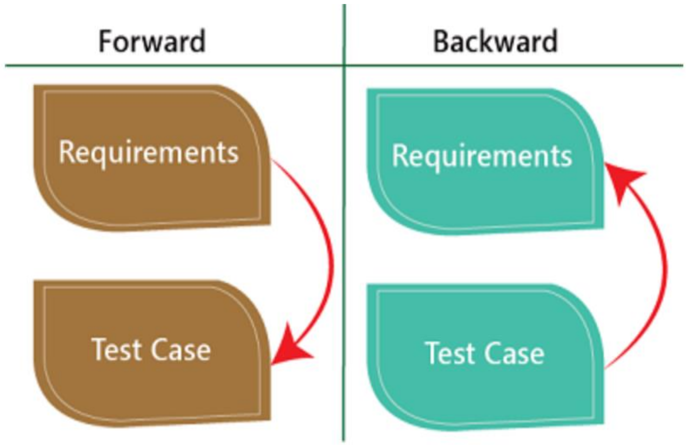

Test Engineer-1
Test Engineer-2
Test Engineer-3
RTM
RTM
RTM
Module1
Module2
Module3
RTM



RTM Template

Requirement number	Module name	High level requirement	Low level requirement	Test case name
2	Loan	**2.1 Personal loan**	2.1.1--> personal loan for private employee	beta-2.0-personal loan
			2.1.2--> personal loan for government employee	
			2.1.3--> personal loan for jobless people	
		2.2 Car loan	2.2.1--> car loan for private employee	
			——	
		2.3 Home loan	——	
			——	
			——	

- The traceability matrix can be classified into three different types which are as follows:
    1. Forward traceability
    2. Backward or reverse traceability
    3. Bi-directional traceability

Forward Traceability	Backward Traceability	Bi-directional Traceability
• Used to ensure that every business's needs or requirements are executed correctly in the application and also tested rigorously. • Requirements are mapped into the forward direction to the test cases.	• Used to check that we are not increasing the space of the product by enhancing the design elements, code, test other things which are not mentioned in the business needs. • Requirements are mapped into the backward direction to the test cases.	• A combination of forwarding and backward traceability matrix. • Used to make sure that all the business needs are executed in the test cases. • Also evaluates the modification in the requirement which is occurring due to the bugs in the application.
		 Forward Backward Requirements Requirements Test Case Test Case

- Goals of Traceability Matrix:
  - It helps in tracing the documents that are developed during various phases of SDLC.
  - It ensures that the software completely meets the customer's requirements.
  - It helps in detecting the root cause of any bug.
- Advantages of RTM:
  - With the help of the RTM document, we can display the complete test execution and bugs status based on requirements.
  - It is used to show the missing requirements or conflicts in documents.
  - We can ensure the complete test coverage, which means all the modules are tested.
  - It will also consider the efforts of the testing teamwork towards reworking or reconsidering on the test cases.

DEVELOPING USE CASES

- **Use case** → a list of actions or event steps typically defining the interactions between a role (known in the Unified Modeling Language (UML) as an actor) and a system to achieve a goal. The actor can be a human or other external system.
- A use case tells a stylized story about how an end user interacts with the system under a specific set of circumstances.
- Regardless of its form, a use case depicts the software or system from the end user’s point of view.

- The first step in writing a use case is to define the set of “actors” that will be involved in the story.
- Actors represent the roles that people (or devices) play as the system operates. An actor is anything that communicates with the system or product and that is external to the system itself.
- 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.

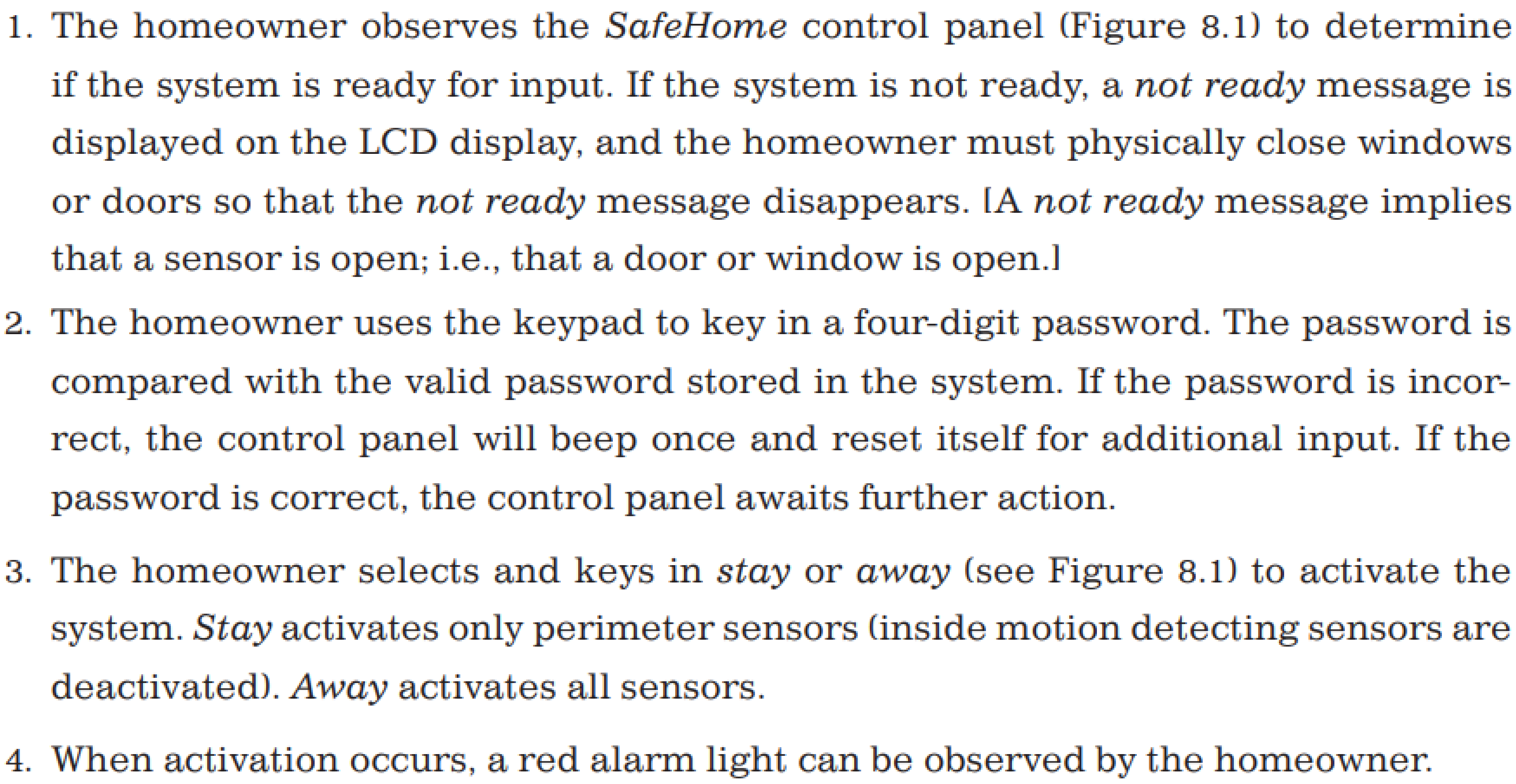
- Ex: consider a machine operator (a user) who interacts with the control computer for a manufacturing cell that contains a number of robots and numerically controlled machines.
- After careful review of requirements, the software for the control computer requires 4 different modes (roles) for interaction: programming mode, test mode, monitoring mode, and troubleshooting mode. Therefore, 4 actors can be defined → programmer, tester, monitor, and troubleshooter.

- It is possible to identify primary actors during the first iteration and secondary actors as more is learned about the system.
- **Primary actors →** interact to achieve required system function and derive the intended benefit from the system. They work directly and frequently with the software.
- **Secondary actors →** support the system so that primary actors can do their work.
- Once actors have been identified, use cases can be developed.

- 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?

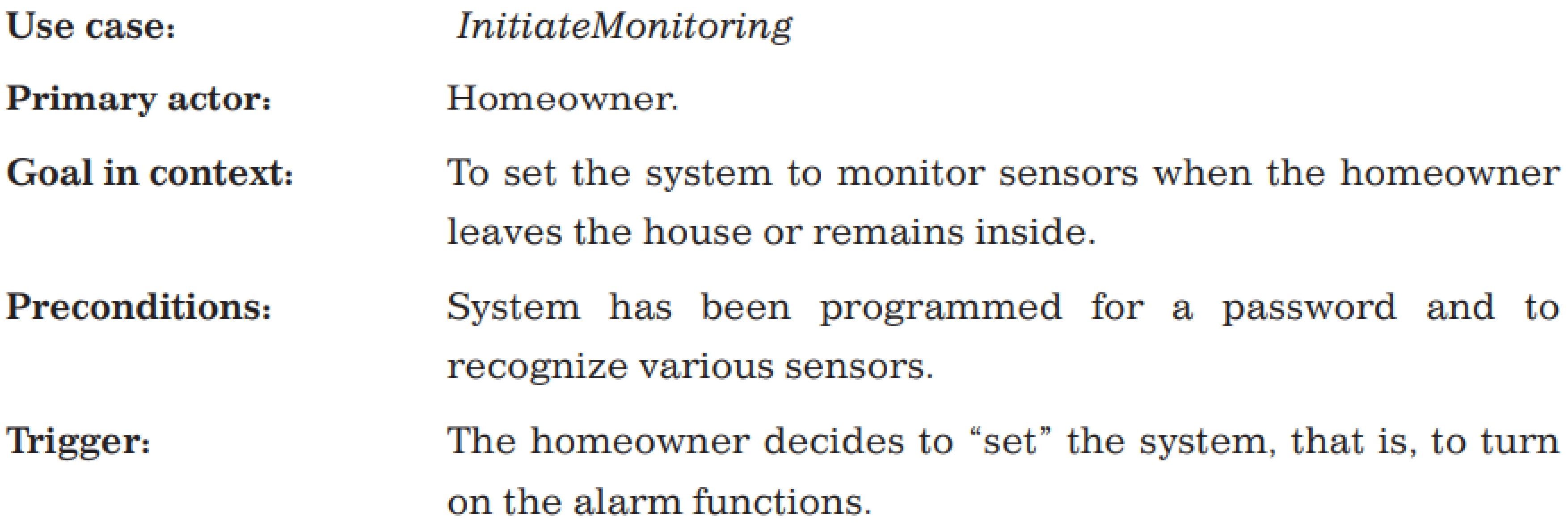
- Ex: basic SafeHome requirements define 4 actors:
    1. homeowner (a user),
    2. setup manager (likely the same person as homeowner, but playing a different role),
    3. sensors (devices attached to the system), and
    4. the monitoring and response subsystem (the central station that monitors 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:
    1. Enters a password to allow all other interactions,
    2. Inquires about the status of a security zone,
    3. Inquires about the status of a sensor,
    4. Presses the panic button in an emergency, and
    5. Activates/deactivates the security system.

- 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 (Figure 8.1) 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.
3. The homeowner selects and keys in *stay* or *away* (see Figure 8.1) 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.

- 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.

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.

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

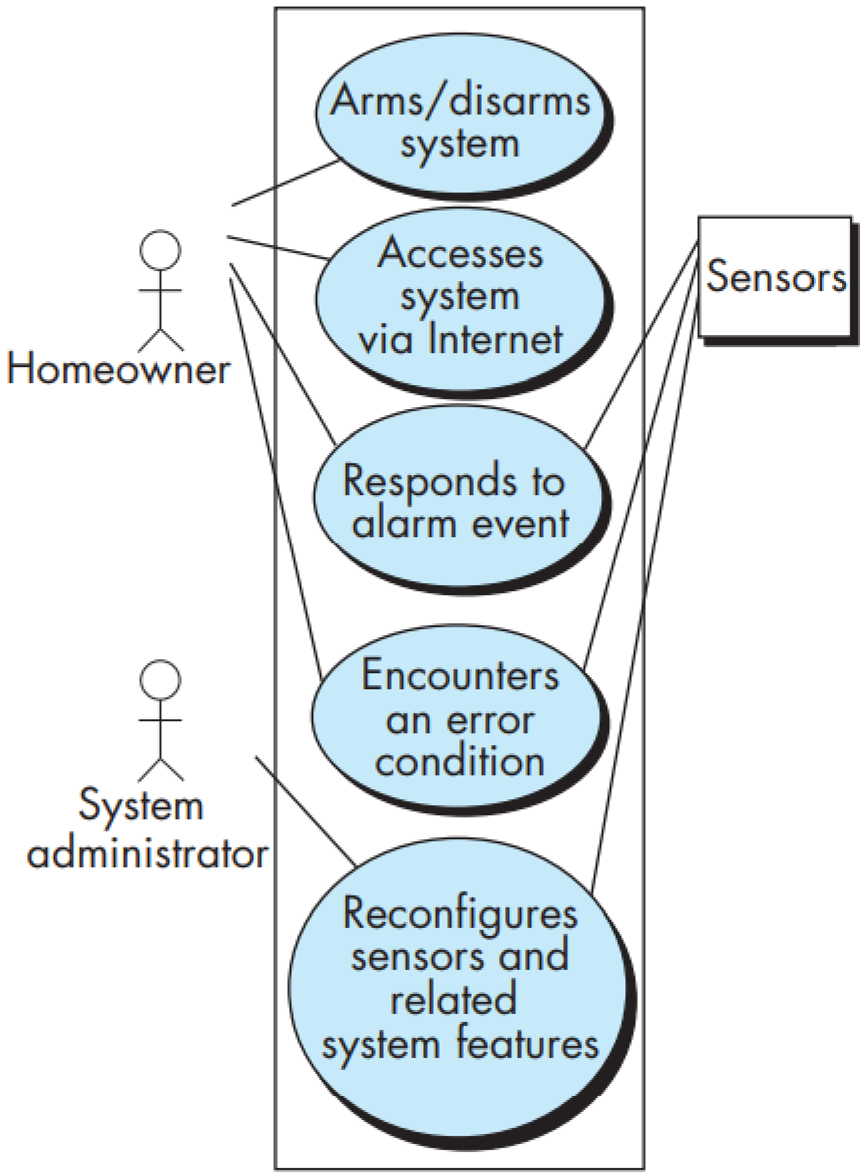
Open issues:

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

Use cases for other **homeowner** interactions would be developed in a similar manner. It is important to review each use case with care. If some element of the interaction is ambiguous, it is likely that a review of the use case will indicate a problem.

FIGURE 8.2

UML use case diagram for *SafeHome* home security function


Homeowner
System administrator
Arms/disarms system
Accesses system via Internet
Responds to alarm event
Encounters an error condition
Reconfigures sensors and related system features
Sensors

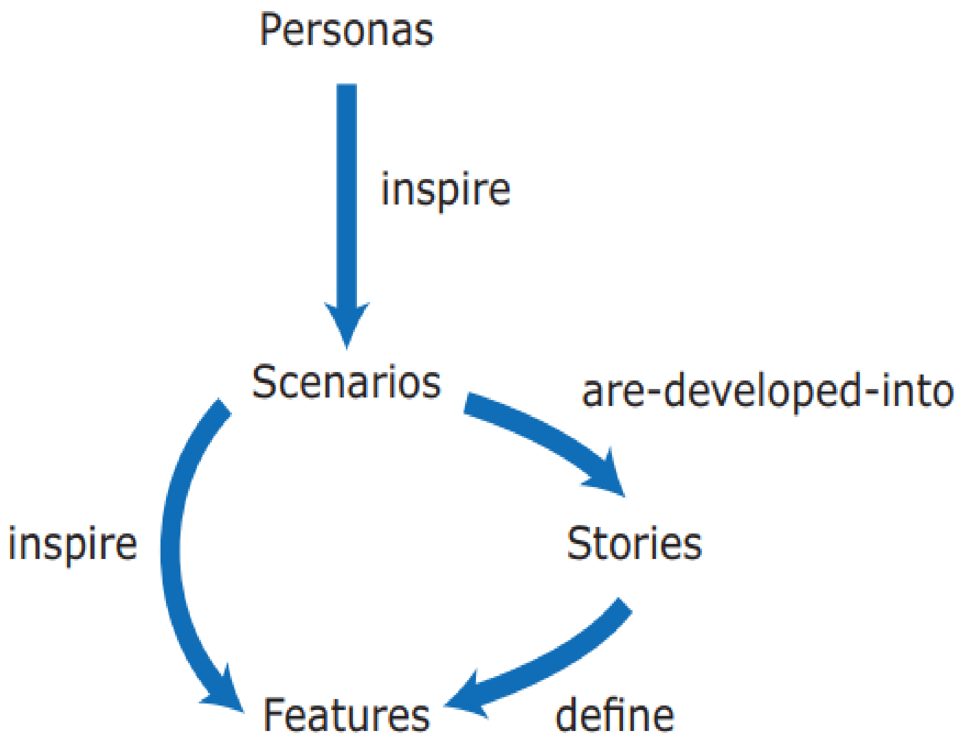


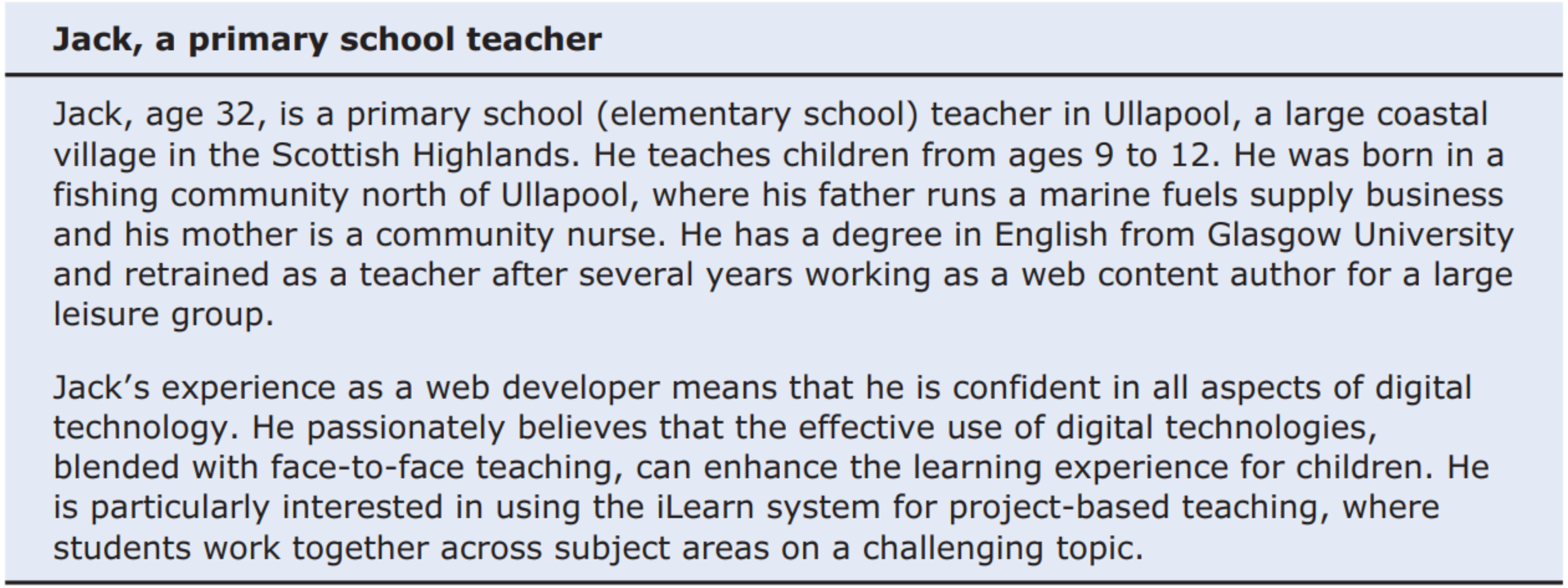
Fig: personas, scenarios, and user stories lead to features that might be implemented in a software product.

Personas

- Personas are about “imagined users,” character portraits of types of user that you think might adopt your product.
- Ex: if your product is aimed at managing appointments 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.

- 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.

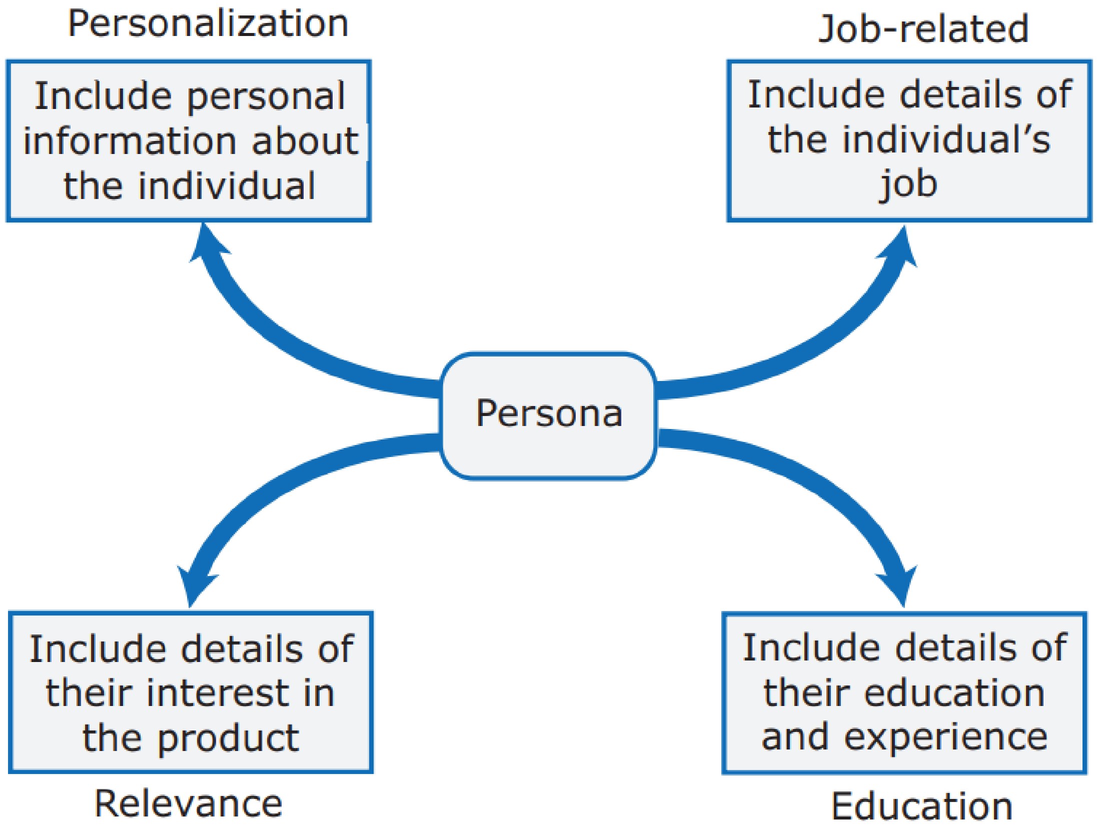
Table 3.1 A persona for a primary school teacher



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.

- There is no standard way of representing a persona.

Figure 3.4 Persona descriptions



Personalization

Include personal information about the individual

Job-related

Include details of the individual’s job

Persona

Include details of their interest in the product

Relevance

Include details of their education and experience

Education

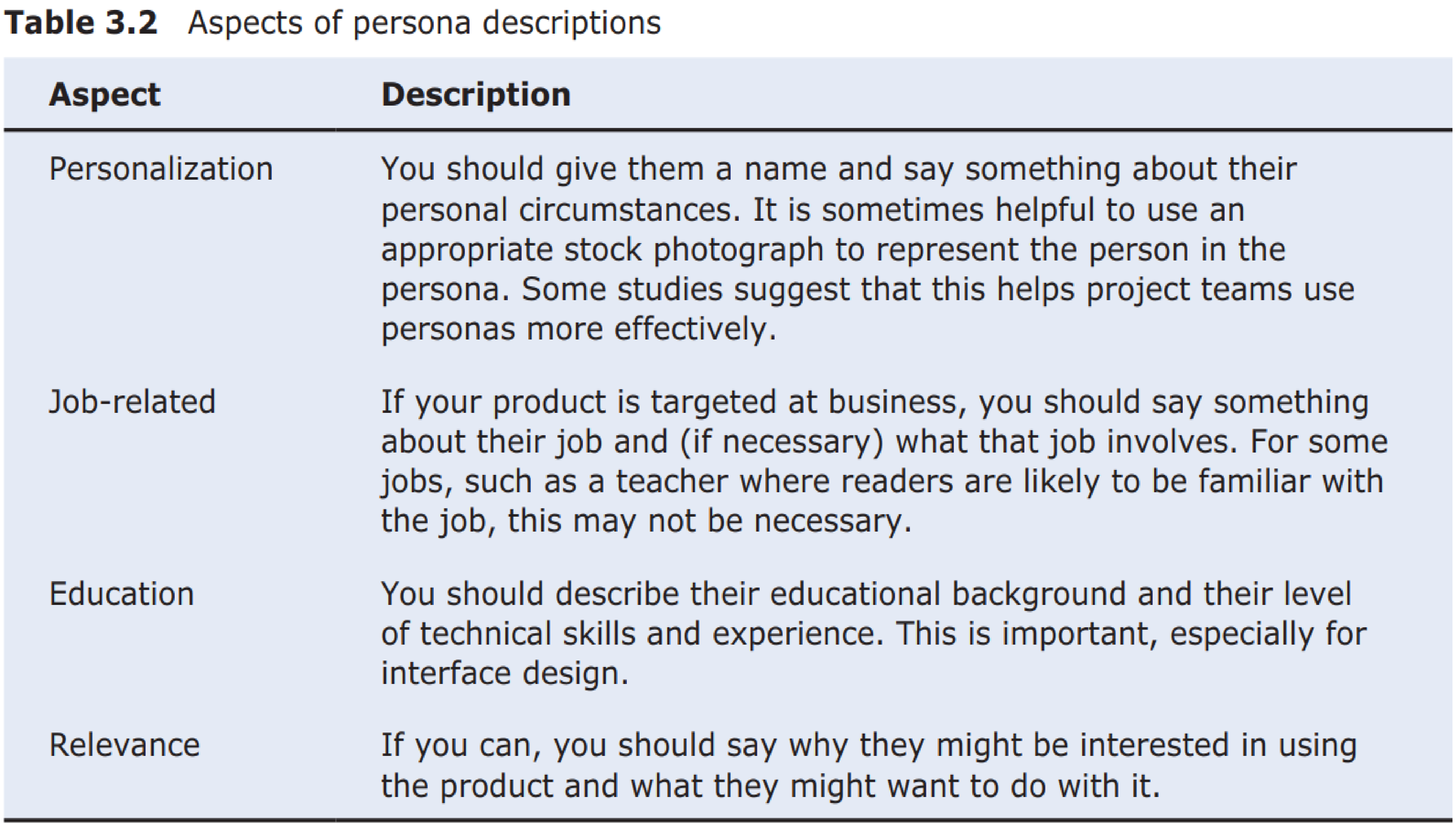


Table 3.2 Aspects of persona descriptions

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

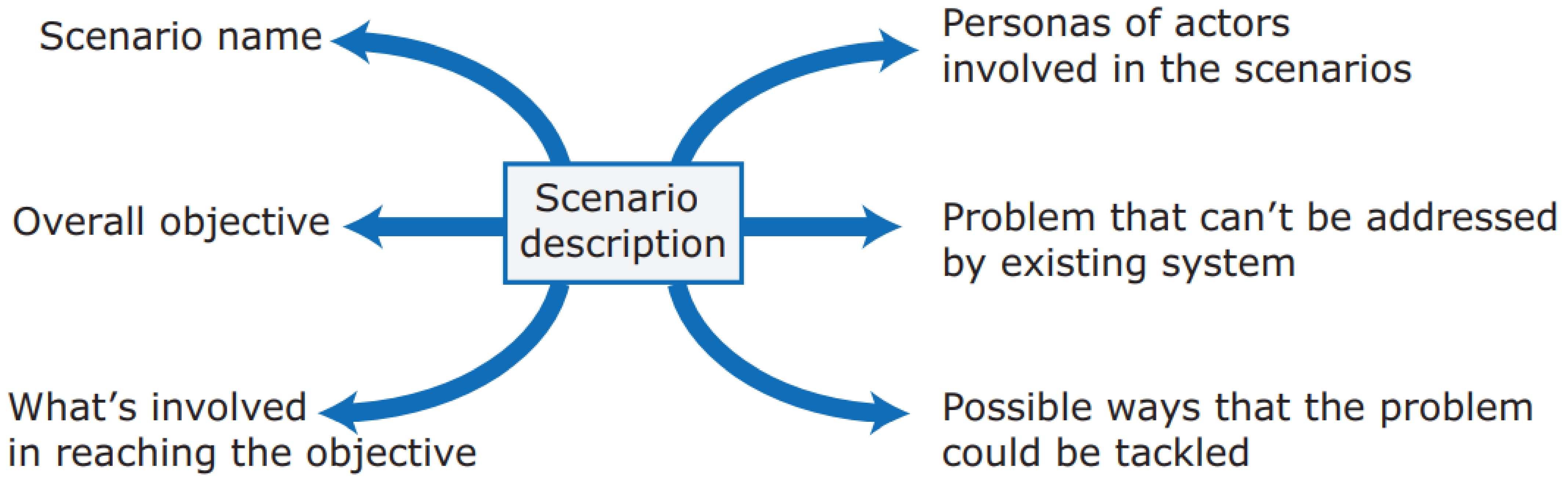
- In general, you don’t need more than 5 personas to help identify the key features of a system.
- Personas should be relatively short and easy to read.
- Personas are a tool that allows team members to “step into the users’ shoes.” Instead of thinking about what they would do in a particular situation, they can imagine how a persona would behave and react.
- They can help you check your ideas to ensure that you are not including product features that aren’t really needed.
- They help you to avoid making unwarranted assumptions, based on your own knowledge, and designing an overcomplicated or irrelevant product.

- **Proto-personas →** Personas that are developed on the basis of limited user information.
- Proto-personas may be created as a collective team exercise using whatever information is available about potential product users.
- They can never be as accurate as personas developed from detailed user studies.
- They represent the product users as seen by the development team.
- They allow the developers to build a common understanding of the potential product users.

SCENARIOS

- A scenario is a narrative that describes a situation in which a user is using your product’s features to do something that they want to do.
- It should briefly explain the user’s problem and present an imagined way that the problem might be solved.
- Scenarios are high-level stories of system use.
- They should describe a sequence of interactions with the system but should not include details of these interactions.
- They are the basis for both use cases, which are extensively used in object-oriented methods, and user stories, which are used in agile methods.
- Scenarios are used in the design of requirements and system features, in system testing, and in user interface design.

Figure 3.5 Elements of a scenario description



Scenario description
Scenario name
Overall objective
What's involved in reaching the objective
Personas of actors involved in the scenarios
Problem that can't be addressed by existing system
Possible ways that the problem could be tackled

- 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.

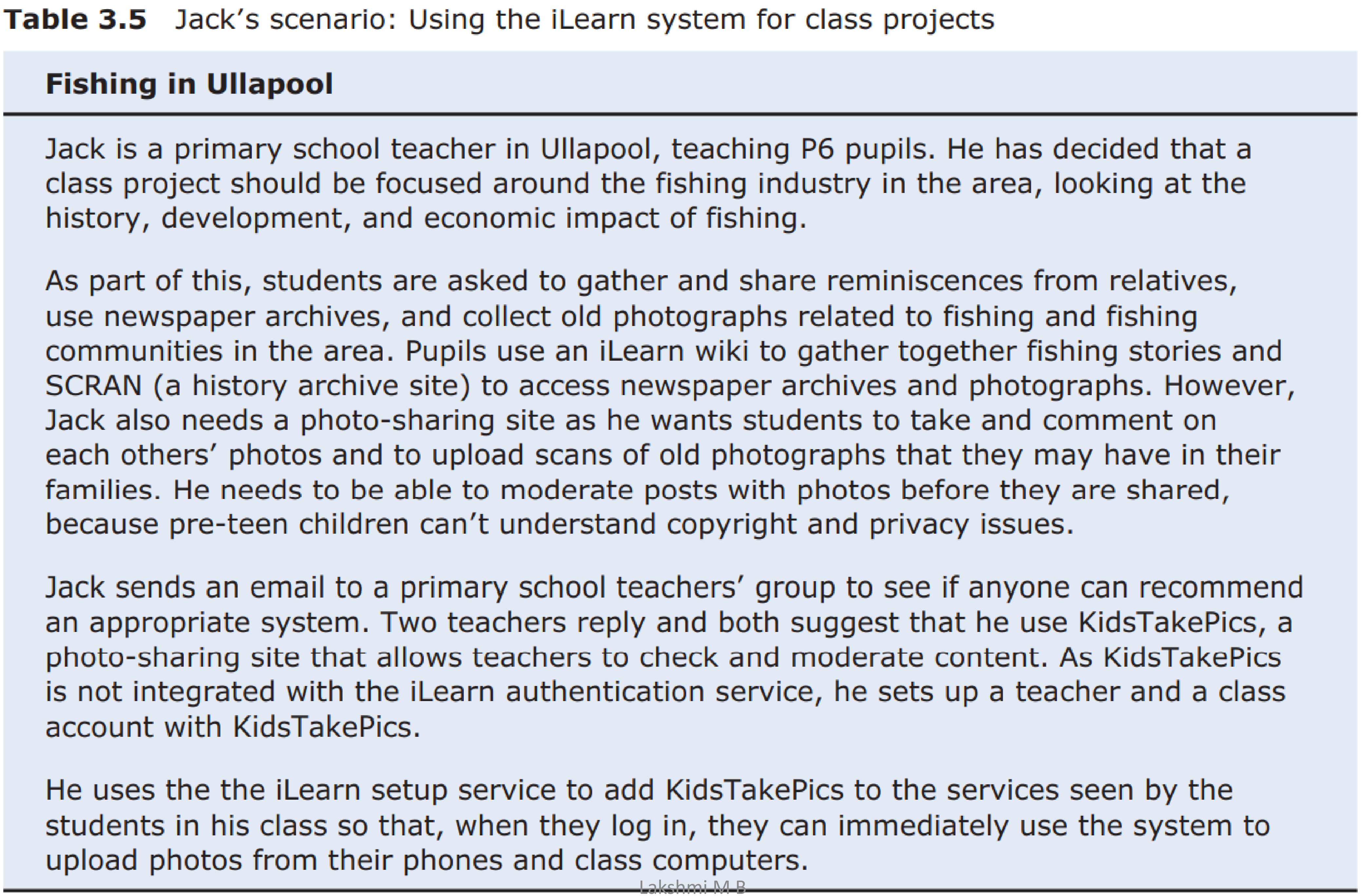


Table 3.5 Jack’s scenario: Using the iLearn system for class projects

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.

- Structured scenarios should include different fields such as :
    - what the user sees at the beginning of a scenario,
    - a description of the normal flow of events,
    - a description of what might go wrong, and so on.
- At the early stages of product design, the scenarios be narrative rather than structured.

Table 3.6 Using the iLearn system for administration

Emma is teaching the history of World War I to a class of 14-year-olds (S3). A group of S3 students are visiting the historic World War I battlefields in northern France. She wants to set up a “battlefields group” where the students who are attending the trip can share their research about the places they are visiting as well as their pictures and thoughts about the visit.

From home, she logs onto the iLearn system using her Google account credentials. Emma has two iLearn accounts—her teacher account and a parent account associated with the local primary school. The system recognizes that she is a multiple account owner and asks her to select the account to be used. She chooses the teacher account and the system generates her personal welcome screen. As well as her selected applications, this also shows management apps that help teachers create and manage student groups.

Emma selects the “group management” app, which recognizes her role and school from her identity information and creates a new group. The system prompts for the class year (S3) and subject (history) and automatically populates the new group with all S3 students who are studying history. She selects those students going on the trip and adds her teacher colleagues, Jamie and Claire, to the group.

She names the group and confirms that it should be created. The app sets up an icon on her iLearn screen to represent the group, creates an email alias for the group, and asks Emma if she wishes to share the group. She shares access with everyone in the group, which means that they also see the icon on their screen. To avoid getting too many emails from students, she restricts sharing of the email alias to Jamie and Claire.

The group management app then asks Emma if she wishes to set up a group web page, wiki, and blog. Emma confirms that a web page should be created and she types some text to be included on that page.

She then accesses Flickr using the icon on her screen, logs in, and creates a private group to share trip photos that students and teachers have taken. She uploads some of her own photos from previous trips and emails an invitation to join the photo-sharing group to the battlefields email list. Emma uploads material from her own laptop that she has written about the trip to iLearn and shares this with the battlefields group. This action adds her documents to the web page and generates an alert to group members that new material is available.

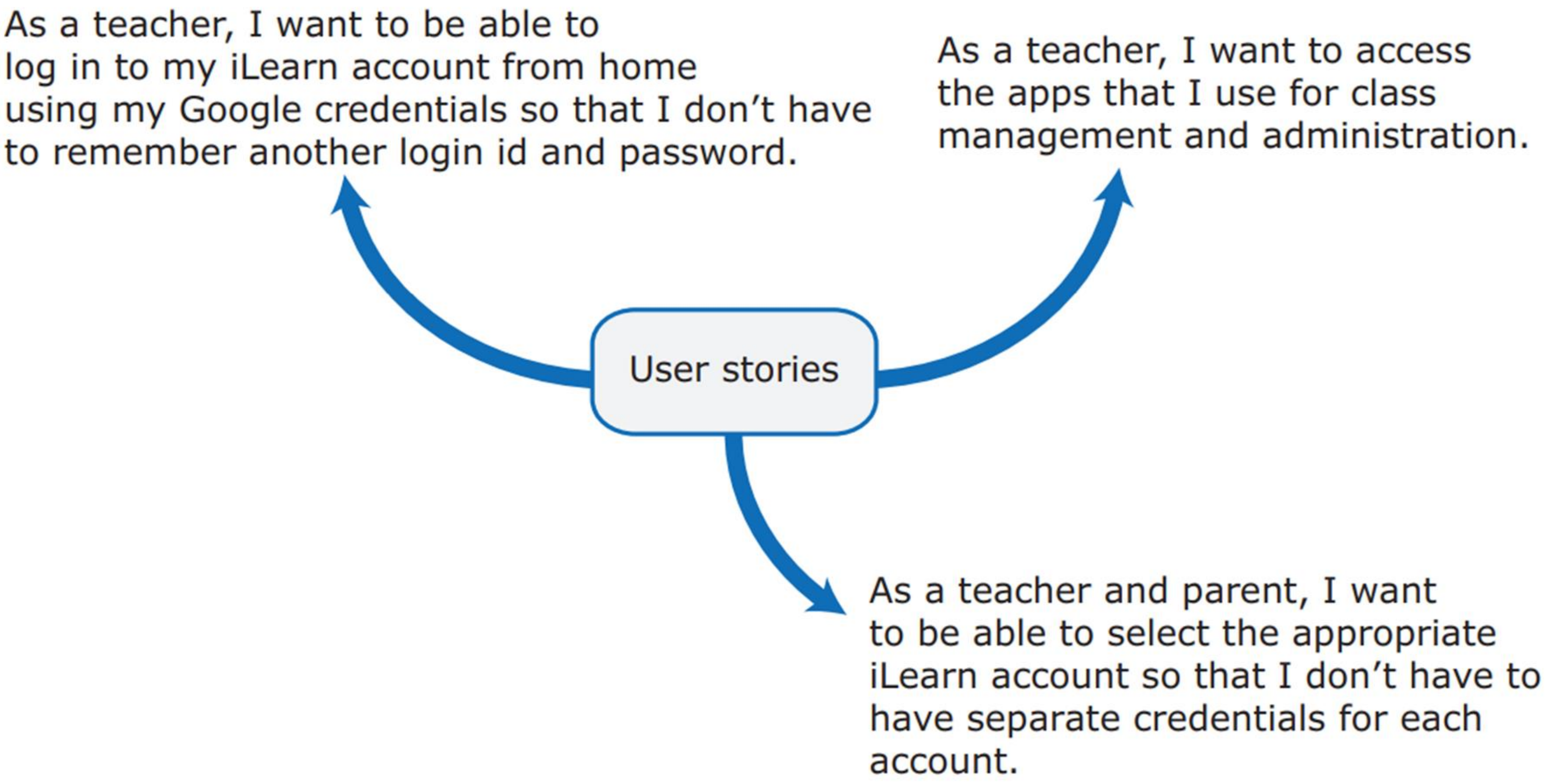
Writing Scenarios

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

USER STORIES

- These are finer-grain narratives that set out in a more detailed and structured way a single thing that a user wants from a software system.
- User stories are not intended for planning but for helping with feature identification.
- Aim to develop stories that are helpful in one of 2 ways:
  - as a way of extending and adding detail to a scenario;
  - as part of the description of the system feature that you have identified.

Figure 3.6 User stories from Emma’s scenario



User stories

As a teacher, I want to be able to log in to my iLearn account from home using my Google credentials so that I don’t have to remember another login id and password.

As a teacher, I want to access the apps that I use for class management and administration.

As a teacher and parent, I want to be able to select the appropriate iLearn account so that I don’t have to have separate credentials for each account.

- 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.

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

Figure 3.8 Feature design

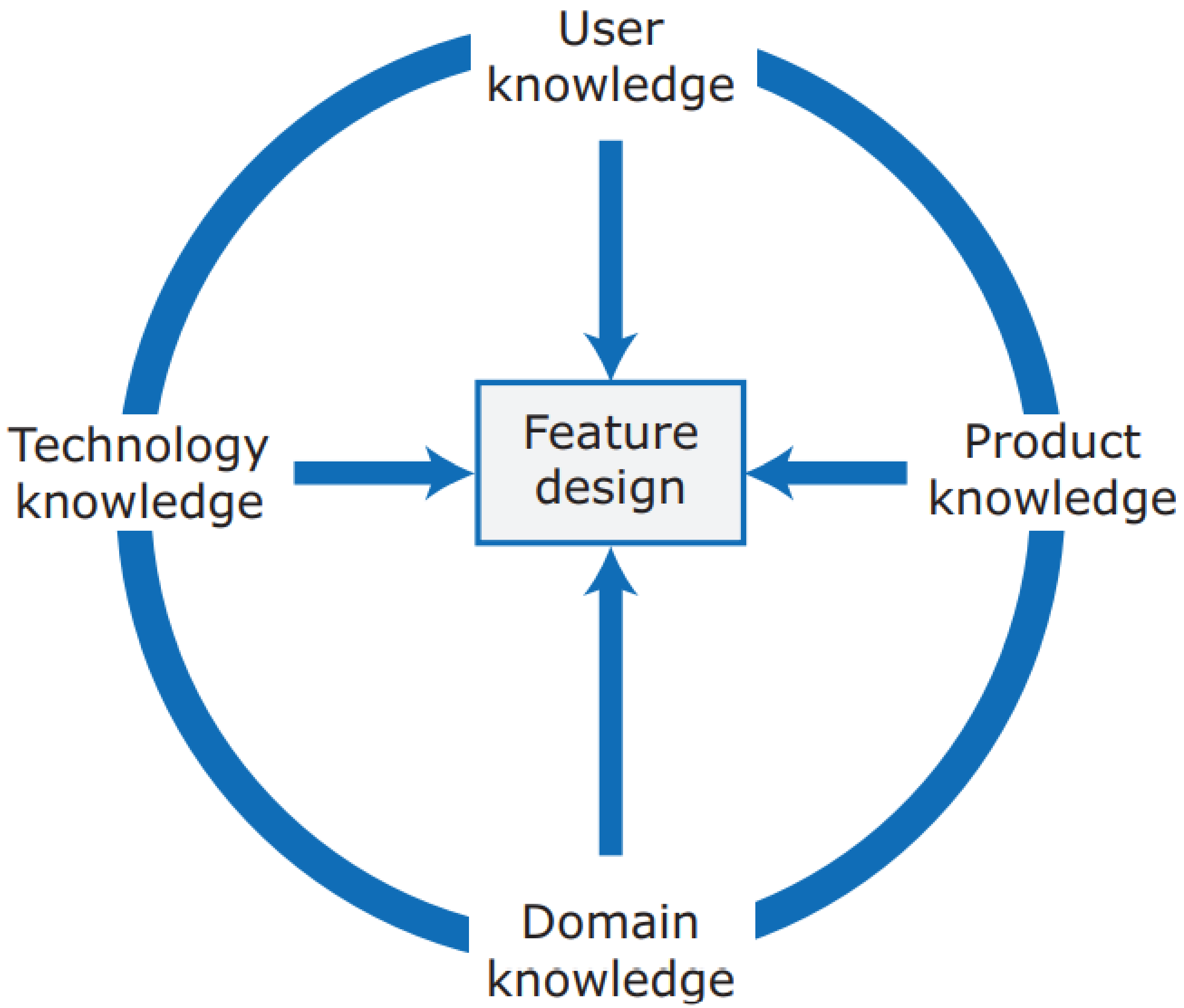
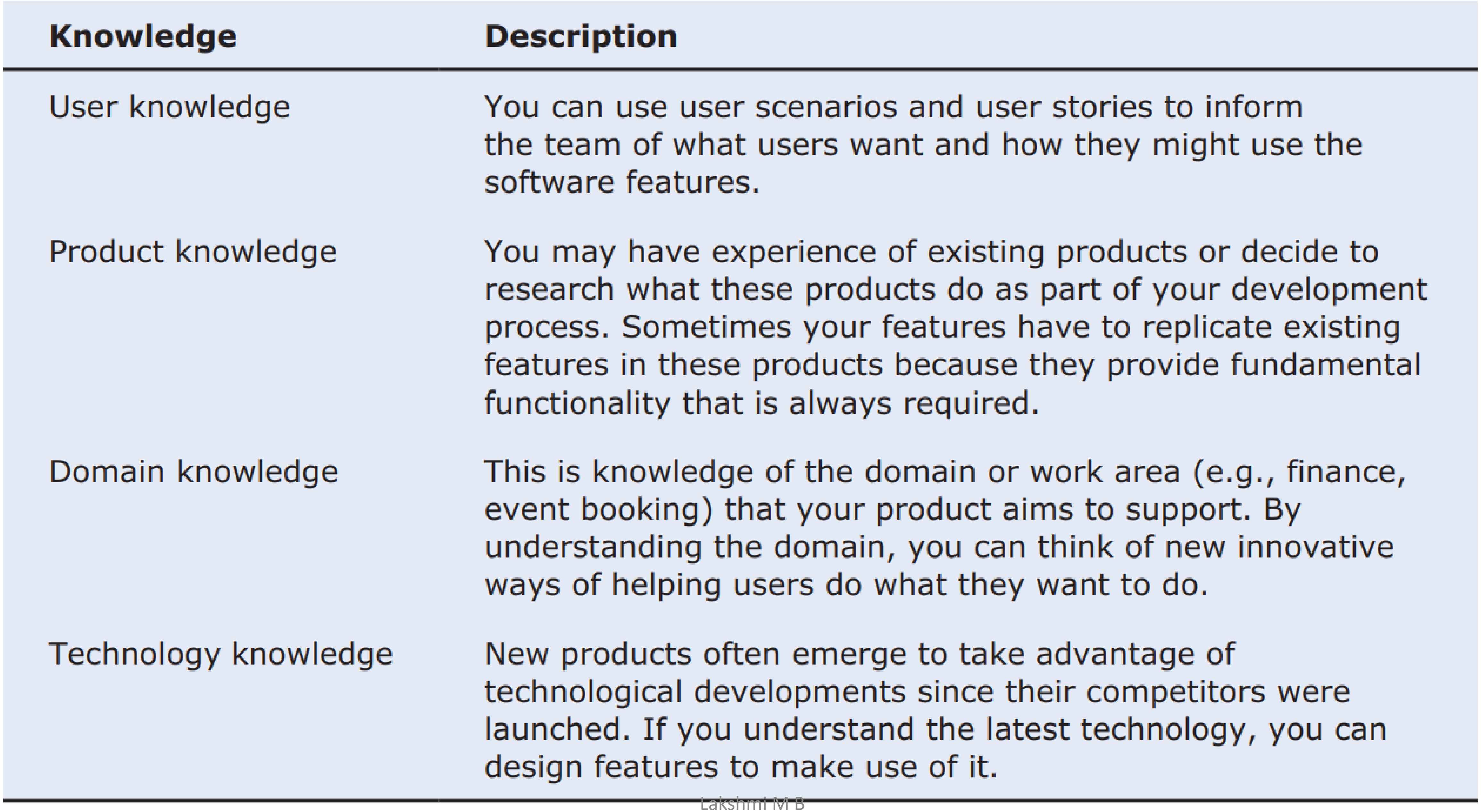
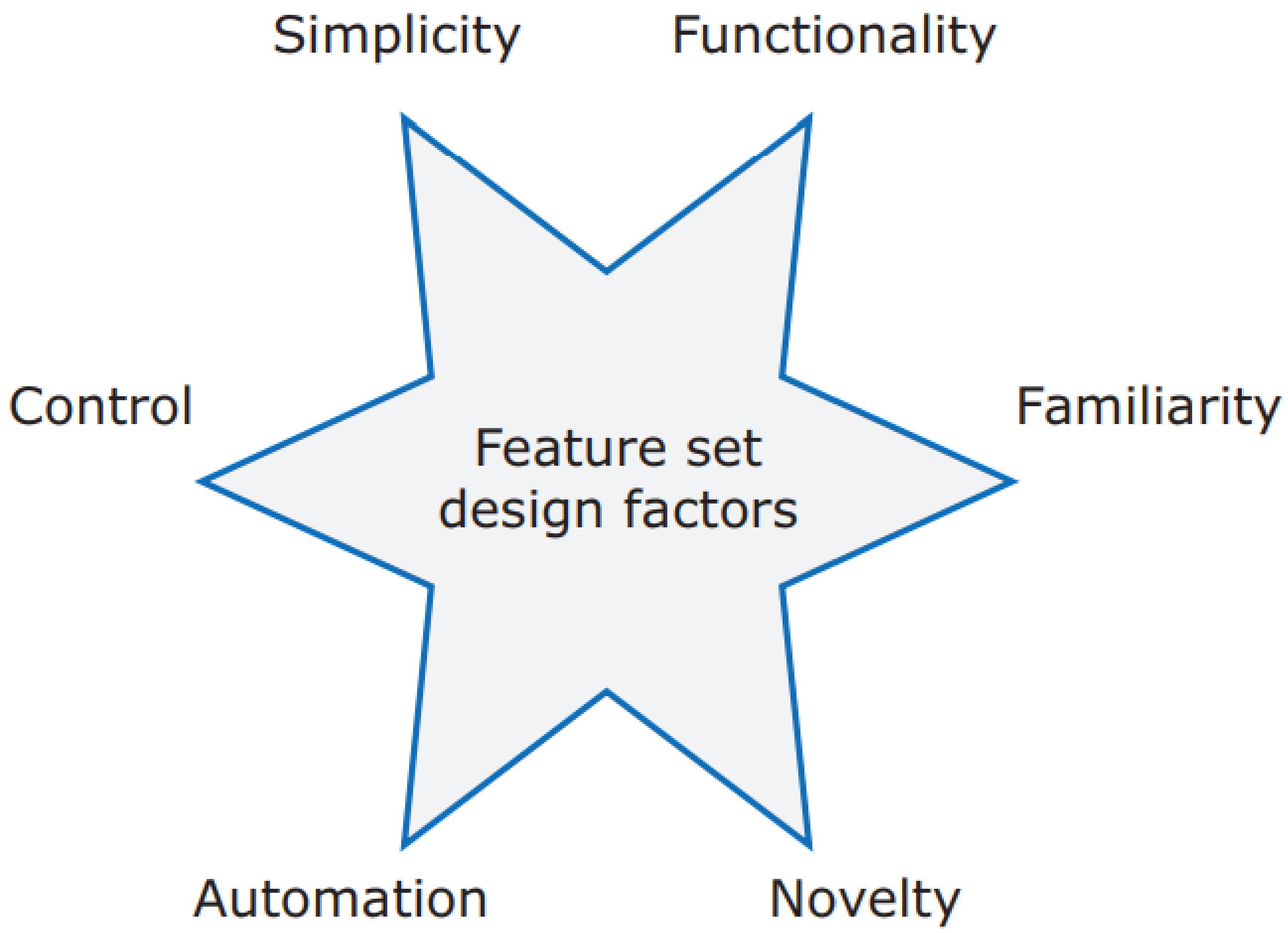


Table 3.8 Knowledge required for feature design



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

Figure 3.9 Factors in feature set design



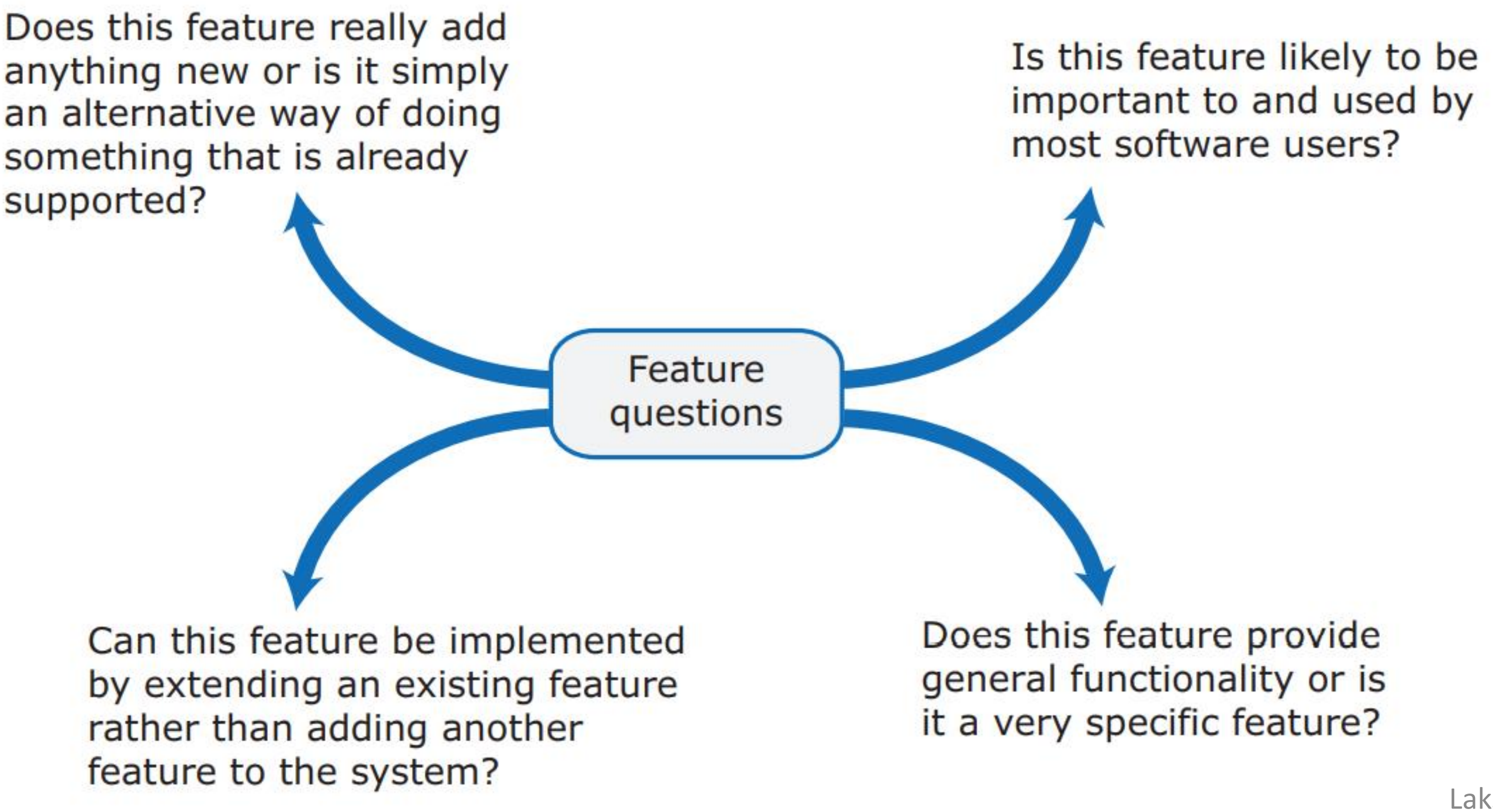
1. **Simplicity and functionality** → a simple, easy-to-use system and including enough functionality to attract users with a variety of needs.
2. **Familiarity and novelty** → Users prefer that new software should support the familiar everyday tasks that are part of their work or life. To encourage users to adopt your system, you need to include new features that will convince users that your product can do more than its competitors.
3. **Automation and control** → think carefully about what can be automated, how it is automated, and how users can configure the automation so that the system can be tailored to their preferences.

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

- Feature creep happens for 3 reasons:
    1. Product managers and marketing executives discuss the functionality they need with a range of different product users. Different users have slightly different needs or may do the same thing but in slightly different ways.
    2. Competitive products are introduced with slightly different functionality to your product. There is marketing pressure to include comparable functionality so that market share is not lost to these competitors. This can lead to “feature wars,” where competing products become more and more bloated as they replicate the features of their competitors.
    3. The product tries to support both experienced and inexperienced users. Easy ways of implementing common actions are added for inexperienced users and the more complex features to accomplish the same thing are retained because experienced users prefer to work that way.

- To avoid feature creep, the product manager and the development team should review all feature proposals and compare new proposals to features that have already been accepted for implementation.

Figure 3.10 Avoiding feature creep



Feature questions

Does this feature really add anything new or is it simply an alternative way of doing something that is already supported?

Is this feature likely to be important to and used by most software users?

Can this feature be implemented by extending an existing feature rather than adding another feature to the system?

Does this feature provide general functionality or is it a very specific feature?

Feature Derivation

- Include:
  - a feature that allows users to access and use existing web-based resources;
  - a feature that allows the system to exist in multiple different configurations;
  - a feature that allows user configuration of the system to create a specific environment.
- The approach of highlighting phrases in a narrative description can be used when analyzing scenarios to find system features.
- Feature identification should be a team activity, and as features are identified, the team should discuss them and generate ideas about related features.
  - Collaborative writing
  - Blogs and web pages

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

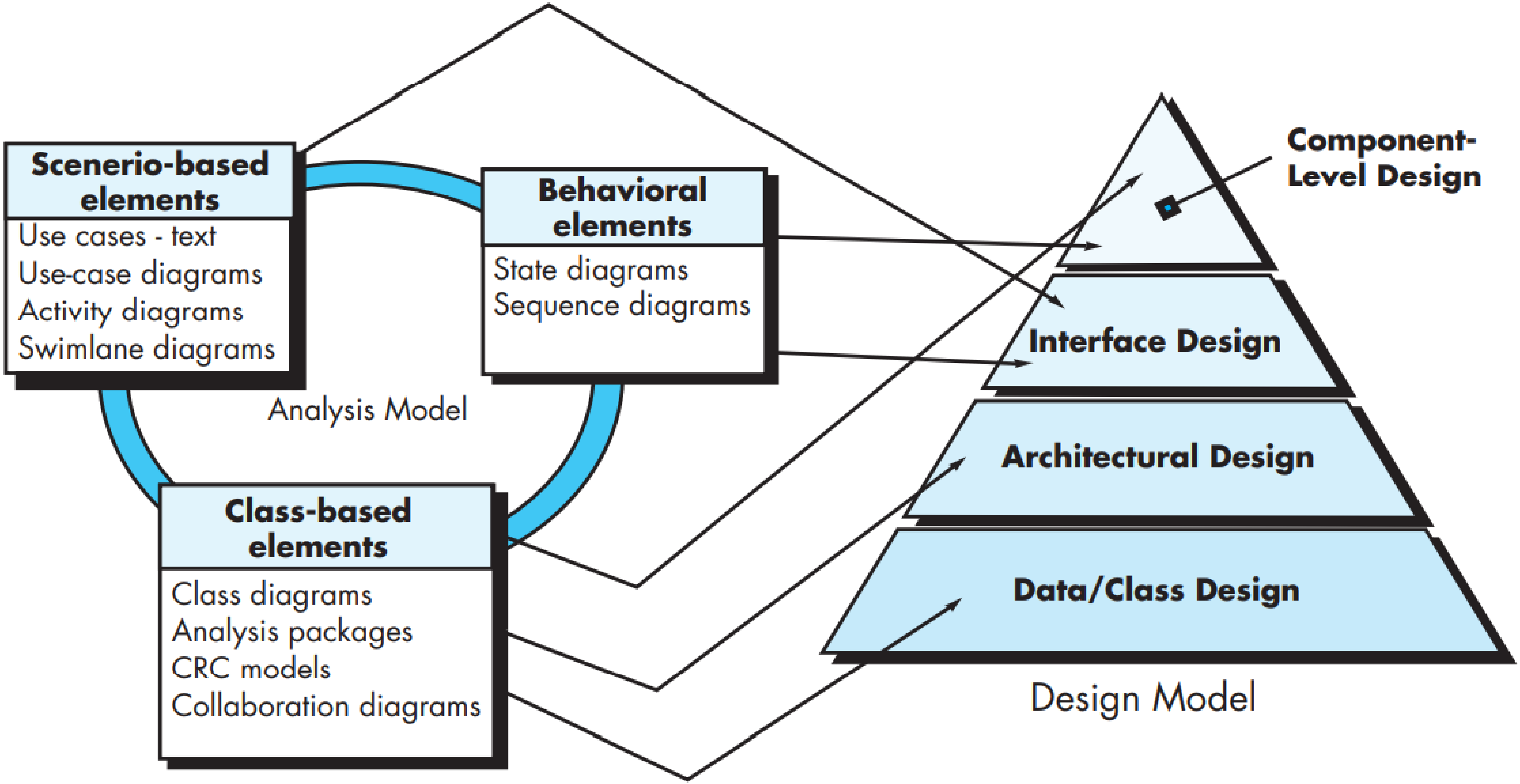
DESIGN CONCEPTS

- Software design encompasses the set of principles, concepts, and practices that lead to the development of a high-quality system or product.
- The goal of design is to produce a model or representation that exhibits firmness, commodity, and delight.
- It changes continually as new methods, better analysis, and broader understanding evolve.

Design within the context of software engineering

- Beginning once software requirements have been analyzed and modeled, software design is the last software engineering action within the modeling activity and sets the stage for construction (code generation and testing).
- The flow of information during software design is illustrated in the figure.

FIGURE 12.1 Translating the requirements model into the design model



- **Data/Class Design →** transforms class models into design class realizations and the requisite data structures required to implement the software. The objects and relationships provide the basis for the data design activity.
- **Architectural Design →** defines the relationship between major structural elements of the software, the architectural styles and patterns that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which architecture can be implemented. The architectural design representation—the framework of a computer-based system—is derived from the requirements model.

- **Interface Design →** describes how the software communicates with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior. Therefore, usage scenarios and behavioral models provide much of the information required for interface design.
- **Component-Level Design →** transforms structural elements of the software architecture into a procedural description of software components. Information obtained from the class-based models and behavioral models serve as the basis for component design.

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

- 3 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 should exhibit an architecture that (1) has been created using recognizable architectural styles or patterns, (2) is composed of components that exhibit good design, and (3) can be implemented in an evolutionary fashion, thereby facilitating implementation and 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.
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

1. Abstraction:

- At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment.
- At lower levels of abstraction, a more detailed description of the solution is provided.
- 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.

2. Architecture:

- Architecture is the structure or organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components.
- One goal of software design is to derive an architectural rendering of a system.
- A set of architectural patterns enables a software engineer to reuse design-level concepts.
- **Structural properties** → "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."

- **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.

3. Patterns:

- A named nugget of insight which conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns.
- Describes a design structure that solves a particular design problem within a specific context and amid “forces” that may have an impact on the manner in which the pattern is applied and used.
- These provide a description that enables a designer to determine:

  (1) whether the pattern is applicable to the current work,

  (2) whether the pattern can be reused (hence, saving design time), and

  (3) whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

4. Separation of Concerns:

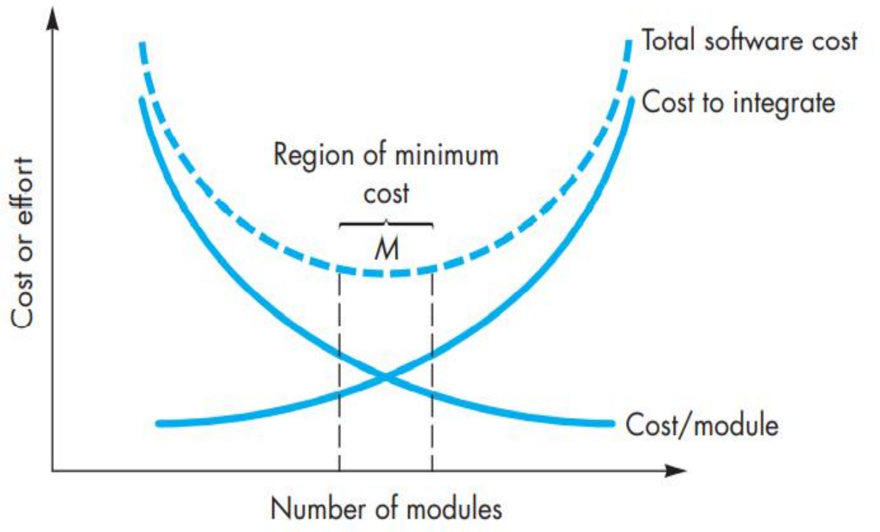
- Suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently.
- **Concern** → a feature or behavior that is specified as part of the requirements model for the software.
- By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.

5. Modularity:

- Software is divided into separately named and addressable components, sometimes called modules, that are integrated to satisfy problem requirements.
- **Modularity** → the single attribute of software that allows a program to be intellectually manageable".

FIGURE 12.2

Modularity and software cost



6. Information Hiding :

- The principle of information hiding suggests that modules should be specified and designed so that information (algorithms and data) contained within a module is inaccessible to other modules that have no need for such information.
- The intent of information hiding is to hide the details of data structures and procedural processing behind a module interface. Knowledge of the details need not be known by users of the module.

7. Functional Independence :

- Functional independence is achieved by developing modules with “ single-minded” function and an “aversion” to excessive interaction with other modules.
- Functional independence is a key to good design, and design is the key to software quality.
- Independence is assessed using 2 qualitative criteria:
  1. **Cohesion** → an indication of the relative functional strength of a module.
  2. **Coupling** → an indication of the relative interdependence among modules.

8. 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. Abstraction enables you to specify procedure and data internally but suppress the need for “outsiders” to have knowledge of low-level details. Refinement helps you to reveal low-level details as design progresses

9. Aspects :

- As design begins, requirements are refined into a modular design representation.
- 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”.
- An aspect is a representation of a crosscutting concern.
- An aspect is implemented as a separate module (component) rather than as software fragments.

10. Refactoring :

- A reorganization technique that simplifies the design (or code) of a component without changing its function or behavior.
- Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure.
- When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design.

11. Object-Oriented Design Concepts :

- OO design concepts –
  - classes and objects,
  - inheritance,
  - messages, and
  - polymorphism.

12. Design Classes:

- These refine the analysis classes by providing design detail that will enable the classes to be implemented, and implement a software infrastructure that supports the business solution.
- 5 different types of design classes:
  1. **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.
  2. **Business domain classes** → identify the attributes and services (methods) that are required to implement some element of the business domain that was defined by one or more analysis classes.
  3. **Process classes** → implement lower-level business abstractions required to fully manage the business domain classes.
  4. **Persistent classes** → represent data stores (e.g., a database) that will persist beyond the execution of the software.
  5. **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.

- 4 characteristics of a well-formed design class:
    1. Complete and sufficient
    2. Primitiveness
    3. High cohesion
    4. Low coupling

13. Dependency Inversion:

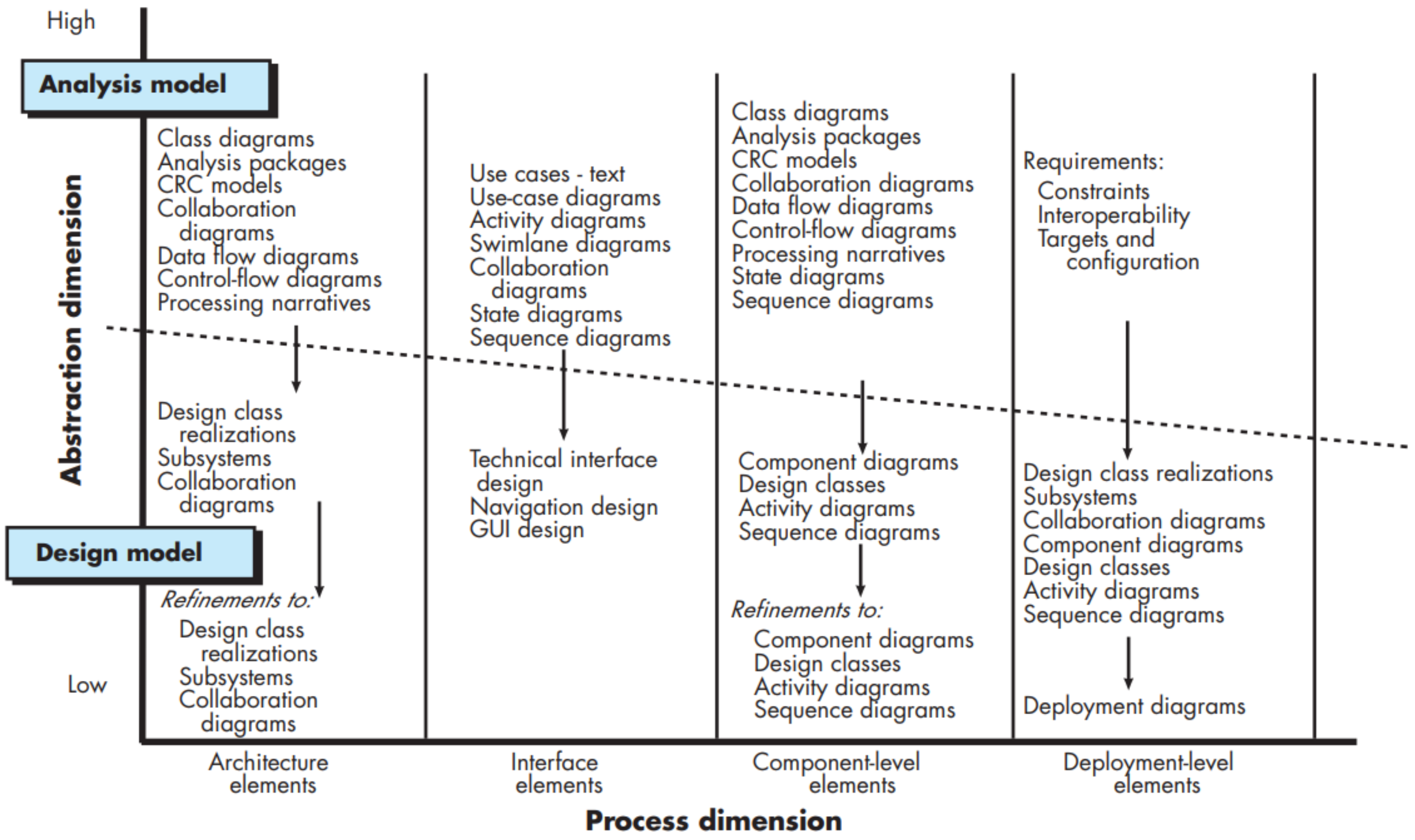
- The dependency inversion principle states: High-level modules (classes) should not depend [directly] upon low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.

14. Design for Test:

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

THE DESIGN MODEL

FIGURE 12.4 Dimensions of the design model



High

Analysis model

Class diagrams
Analysis packages
CRC models
Collaboration diagrams
Data flow diagrams
Control-flow diagrams
Processing narratives

Use cases - text
Use-case diagrams
Activity diagrams
Swimlane diagrams
Collaboration diagrams
State diagrams
Sequence diagrams

Class diagrams
Analysis packages
CRC models
Collaboration diagrams
Data flow diagrams
Control-flow diagrams
Processing narratives
State diagrams
Sequence diagrams

Requirements:
Constraints
Interoperability
Targets and configuration

Abstraction dimension

Design class realizations
Subsystems
Collaboration diagrams

Technical interface design
Navigation design
GUI design

Component diagrams
Design classes
Activity diagrams
Sequence diagrams

Design class realizations
Subsystems
Collaboration diagrams
Component diagrams
Design classes
Activity diagrams
Sequence diagrams

Design model

Refinements to:
Design class realizations
Subsystems
Collaboration diagrams

Refinements to:
Component diagrams
Design classes
Activity diagrams
Sequence diagrams

Deployment diagrams

Low

Architecture elements

Interface elements

Component-level elements

Deployment-level elements

Process dimension

- 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 refined iteratively.
- The design model has 4 major elements:
  1. Data Design Elements,
  2. Architectural Design Elements,
  3. Interface Design Elements, and
  4. Component-Level Design Elements.

1. Data Design Elements:

- Data design (or data architecting) creates a model of data and/or information that is represented at a high level of abstraction (the customer/user’s view of data).
- Then refined into progressively more implementation-specific representations that can be processed by the computer-based system.
- At the program-component level, the design of data structures and the associated algorithms is essential to the creation of high- quality applications.
- At the application level, the translation of a data model (derived as part of requirements engineering) into a database is pivotal to achieving the business objectives of a system.
- At the business level, the collection of information stored in disparate databases and reorganized into a “data warehouse” enables data mining or knowledge discovery that can have an impact on the success of the business itself.

2. Architectural Design Elements:

- The architectural design for software is the equivalent to the floor plan of a house.
- The architectural model is derived from 3 sources:
    1. information about the application domain for the software to be built;
    2. specific requirements model elements such as use cases or analysis classes, their relationships and collaborations for the problem at hand; and
    3. the availability of architectural styles and patterns.
- The architectural design element is usually depicted as a set of interconnected subsystems, often derived from analysis packages within the requirements model.
- Each subsystem may have its own architecture.

3. Interface Design Elements:

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

4. Component-Level Design Elements:

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

4. Deployment-Level Design Elements:

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

ARCHITECTURAL DESIGN
SOFTWARE ARCHITECTURE

1. 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.
- Software architecture must model the structure of a system and the manner in which data and procedural components collaborate with one another.

- The architecture is not the operational software. Rather, it is a representation that enables you to:
    1. analyze the effectiveness of the design in meeting its stated requirements,
    2. consider architectural alternatives at a stage when making design changes is still relatively easy, and
    3. reduce the risks associated with the construction of the software.
- In the context of architectural design, a software component can be something as simple as a program module or an object- oriented class, but it can also be extended to include databases and “middleware” that enable the configuration of a network of clients and servers.
- The properties of components are those characteristics that are necessary to an understanding of how the components interact with other components.
- A design is an instance of an architecture similar to an object being an instance of a class.

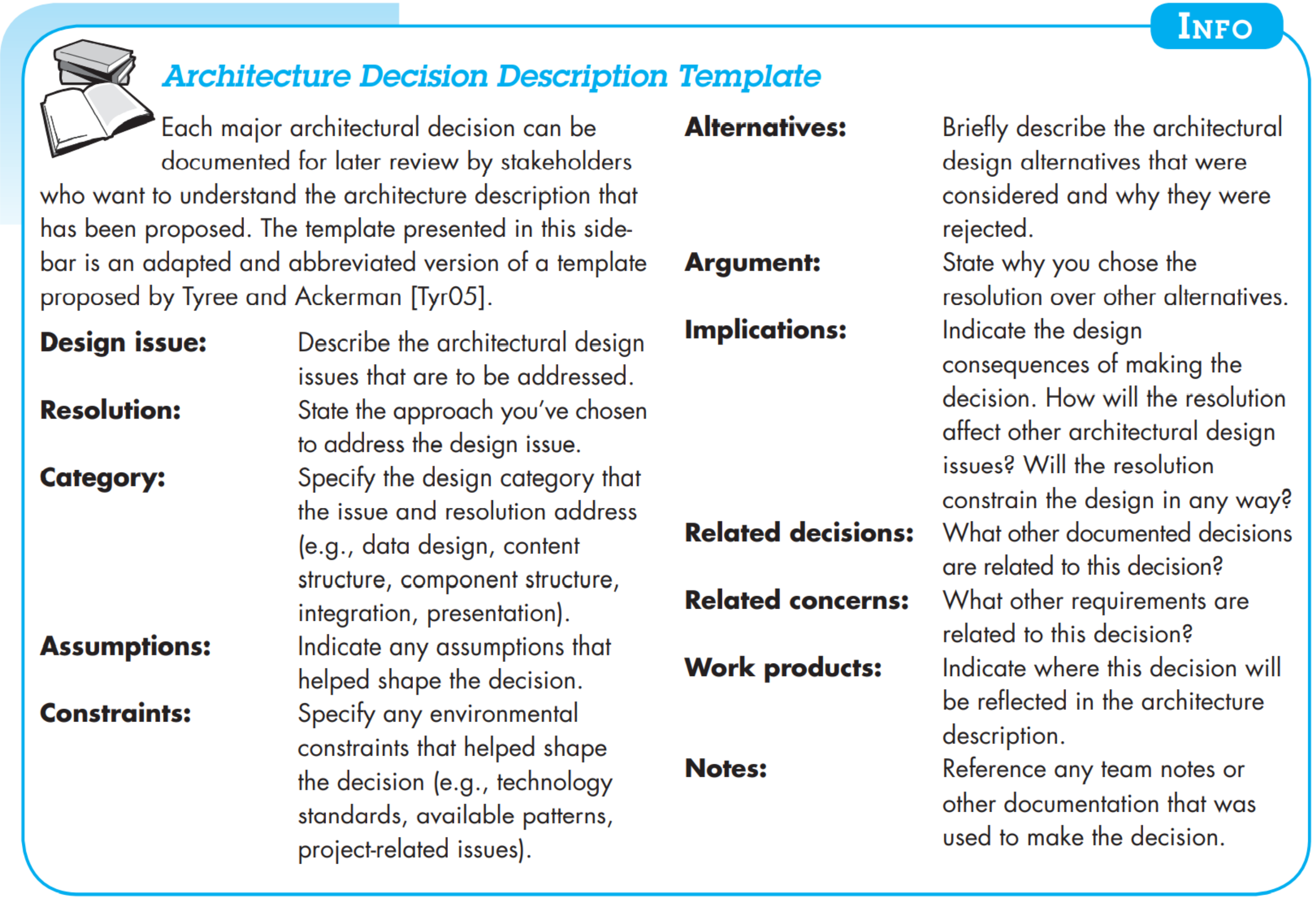
2. Why is Architecture Important?

- Three key reasons:
  1. Software architecture provides a representation that facilitates communication among all stakeholders.
  2. The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows.
  3. Architecture “constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together”.

3. Architectural Descriptions

- An architectural description is actually a set of work products that reflect different views of the system.
- **Metaphors** represent different views of the same architecture, that stakeholders use to understand the term software architecture.
- **Blueprint metaphor** → write programs to implement a system.
- **Language metaphor** → views architecture as a facilitator of communication across stakeholder groups.
- **Decision metaphor →** represents architecture as the product of decisions involving trade-offs among properties such as cost, usability, maintainability, and performance.
- **Literature metaphor** → used to document architectural solutions constructed in the past.

3. Architectural Decisions



INFO

Architecture Decision Description Template

Each major architectural decision can be documented for later review by stakeholders who want to understand the architecture description that has been proposed. The template presented in this sidebar is an adapted and abbreviated version of a template proposed by Tyree and Ackerman [Tyr05].

Design issue: Describe the architectural design issues that are to be addressed.

Resolution: State the approach you've chosen to address the design issue.

Category: Specify the design category that the issue and resolution address (e.g., data design, content structure, component structure, integration, presentation).

Assumptions: Indicate any assumptions that helped shape the decision.

Constraints: Specify any environmental constraints that helped shape the decision (e.g., technology standards, available patterns, project-related issues).

Alternatives: Briefly describe the architectural design alternatives that were considered and why they were rejected.

Argument: State why you chose the resolution over other alternatives.

Implications: Indicate the design consequences of making the decision. How will the resolution affect other architectural design issues? Will the resolution constrain the design in any way?

Related decisions: What other documented decisions are related to this decision?

Related concerns: What other requirements are related to this decision?

Work products: Indicate where this decision will be reflected in the architecture description.

Notes: Reference any team notes or other documentation that was used to make the decision.

Architectural Styles

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

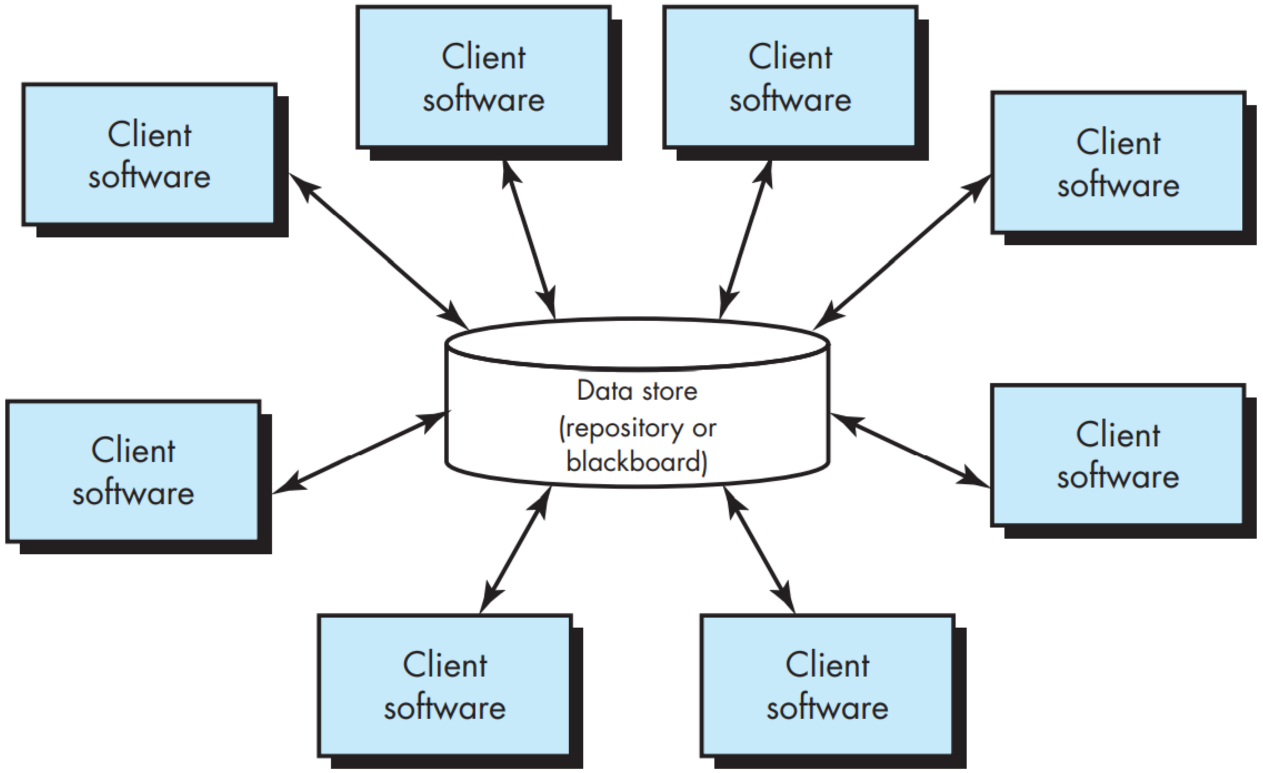
- An architectural pattern also imposes a transformation on the design of an architecture.
- A pattern differs from a style in a number of fundamental ways:
    1. the scope of a pattern is less broad, focusing on one aspect of the architecture rather than the architecture in its entirety,
    2. a pattern imposes a rule on the architecture, describing how the software will handle some aspect of its functionality at the infrastructure level
    3. architectural patterns tend to address specific behavioral issues within the context of the architecture.
- Patterns can be used in conjunction with an architectural style to shape the overall structure of a system.

1. A Brief Taxonomy of Architectural Styles

- **Data-centered Architectures:**

FIGURE 13.1

Data-centered architecture

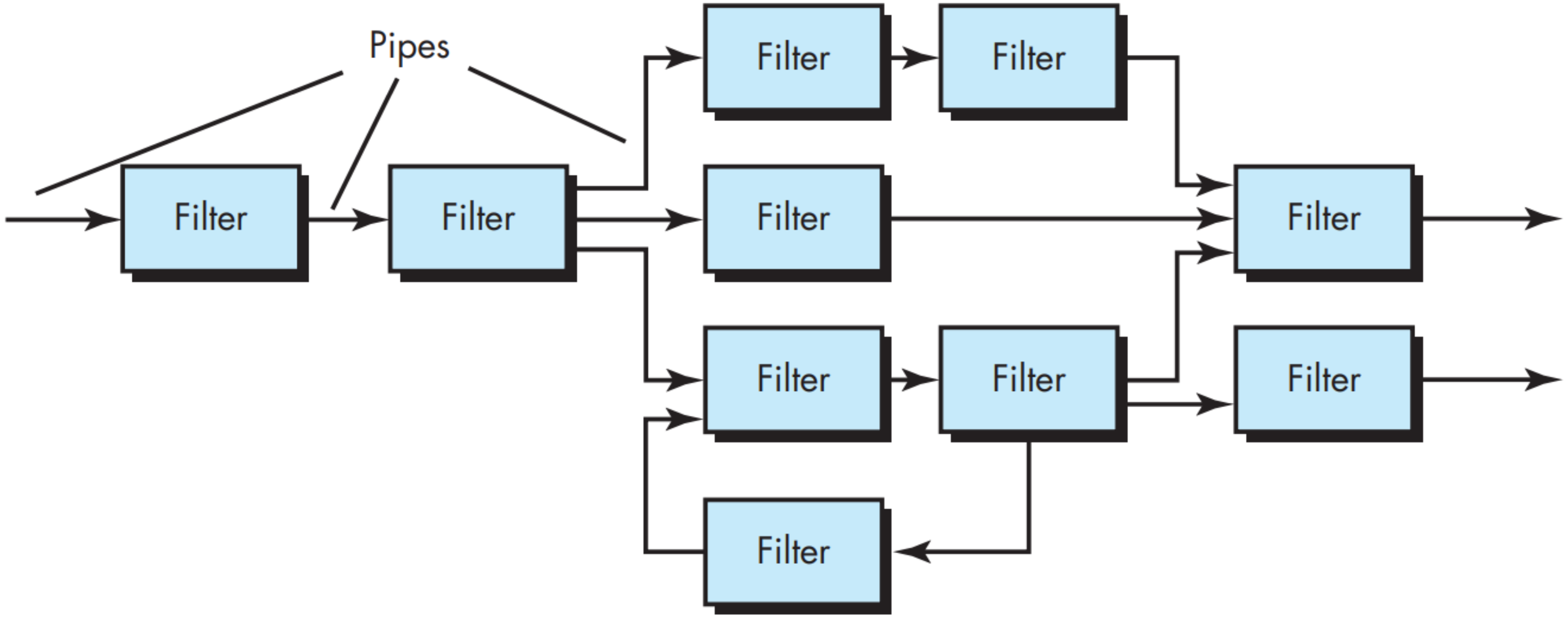


- A data store resides at the center of this architecture and is accessed frequently by other components.
- Client software accesses a central repository.
- In some cases the data repository is passive i.e., client software accesses the data independent of any changes to the data or the actions of other client software.
- A variation on this approach transforms the repository into a “blackboard” that sends notifications to client software when data of interest to the client changes.
- This promotes integrability.
- Data can be passed among clients using the blackboard mechanism (i.e., the blackboard component serves to coordinate the transfer of information between clients).

- **Data-flow Architectures:**

FIGURE 13.2

Data-flow architecture



Pipes and filters

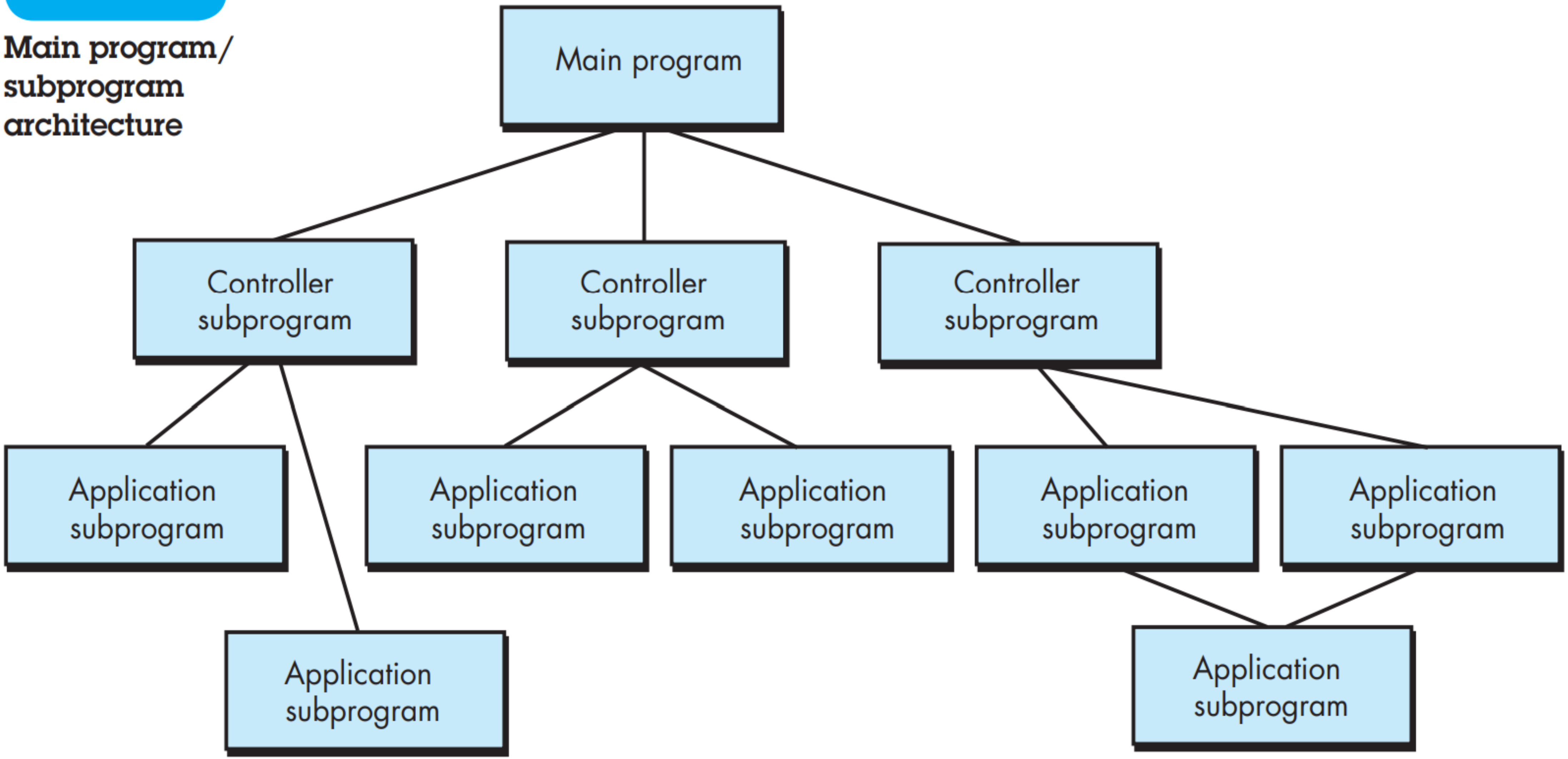
- This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data.
- A pipe-and-filter pattern has a set of components, called **filters**, connected by pipes that transmit data from one component to the next.
- Each filter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

- **Call and Return Architectures:**
  - This architectural style enables you to achieve a program structure that is relatively easy to modify and scale.
  - A number of substyles exist within this category:
    a) **Main program/subprogram architectures** → This classic program structure decomposes function into a control hierarchy where a "main" program invokes a number of program components, which in turn may invoke still other components.

    b) **Remote procedure call architectures** → The components of a main program/ subprogram architecture are distributed across multiple computers on a network.

Figure 13.3

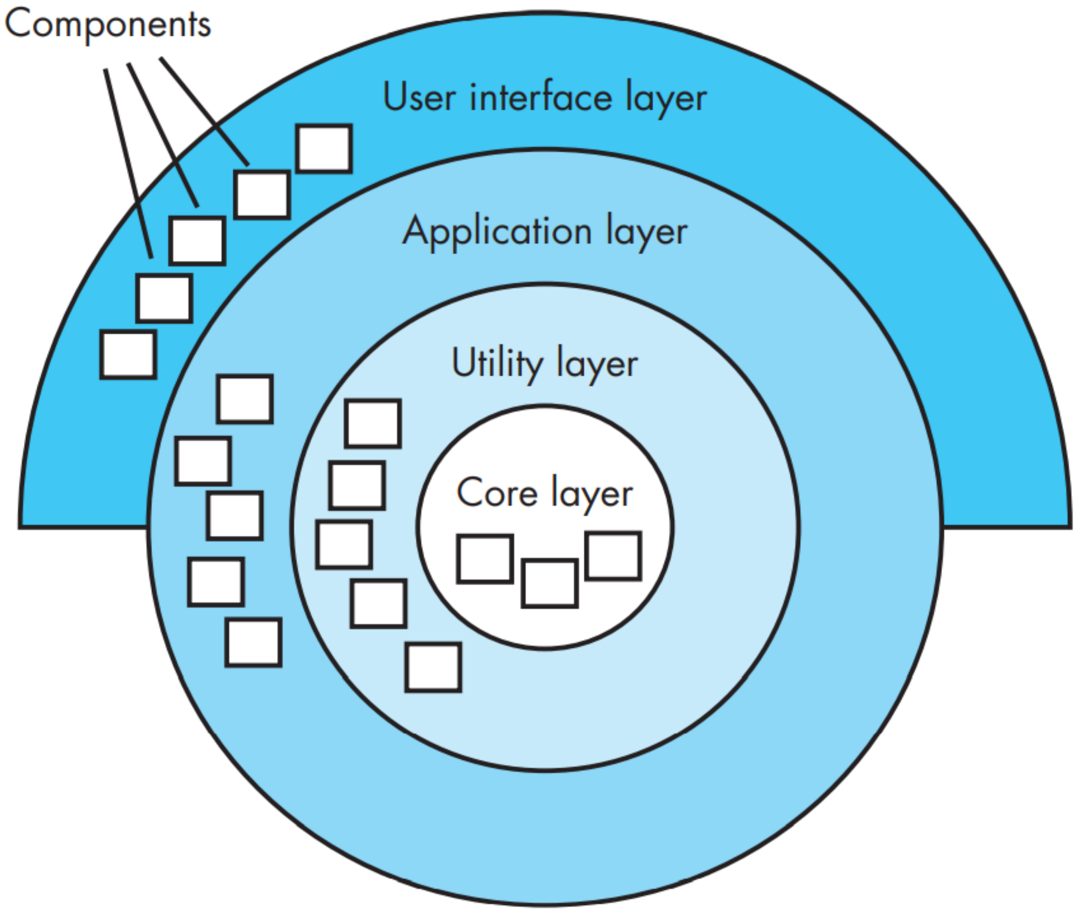
Main program/ subprogram architecture



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

Figure 13.4

Layered architecture



- 2 complementary concepts for choosing the right architecture style:
  1. **Problem frames** → describe characteristics of recurring problems, without being distracted by references to details of domain knowledge or programming solution implementations.
  2. **Domain-driven design** → suggests that the software design should reflect the domain and the domain logic of the business problem you want to solve with your application.
- A problem frame is a generalization of a class of problems that might be used to solve the problem at hand.
- 5 fundamental problem frames:
  1. simple work pieces (tools),
  2. required behavior (data-centered),
  3. commanded behavior (command processor),
  4. information display (observer), and
  5. transformation (pipe and filter variants).

2. Architectural Patterns

- Architectural patterns address an application-specific problem within a specific context and under a set of limitations and constraints.
- The pattern proposes an architectural solution that can serve as the basis for architectural design.

3. Organization and Refinement

- It is important to establish a set of design criteria that can be used to assess an architectural design that is derived.
- Certain questions based on **Control** and **Data** provide insight into an architectural style.
- These questions provide the designer with an early assessment of design quality and lay the foundation for more detailed analysis of the architecture.

Architectural Considerations

1. **Economy** → Many software architectures suffer from unnecessary complexity driven by the inclusion of unnecessary features or non-functional requirements.
2. **Visibility** → Poor visibility arises when important design and domain concepts are poorly communicated to those who must complete the design and implement the system.
3. **Spacing** → Separation of concerns in a design without introducing hidden dependencies is a desirable design concept (sometimes referred to as spacing). Sufficient spacing leads to modular designs, but too much spacing leads to fragmentation and loss of visibility.
4. **Symmetry** → Architectural symmetry implies that a system is consistent and balanced in its attributes. Symmetric designs are easier to understand, comprehend, and communicate.
5. **Emergence** → Emergent, self-organized behavior and control are often the key to creating scalable, efficient, and economic software architectures.

- These considerations do not exist in isolation. They interact with each other and are moderated by each other.

Architectural Design

- As architectural design begins, context must be established.
- To accomplish this, the external entities that interact with the software and the nature of their interaction are described.
- This information can generally be acquired from the requirements model.
- Once context is modeled and all external software interfaces have been described, you can identify a set of architectural archetypes.

- **Archetype** → an abstraction (similar to a class) that represents one element of system behavior.
- The set of archetypes provides a collection of abstractions that must be modeled architecturally if the system is to be constructed, but the archetypes themselves do not provide enough implementation detail.
- Therefore, the designer specifies the structure of the system by defining and refining software components that implement each archetype.
- This process continues iteratively until a complete architectural structure has been derived.

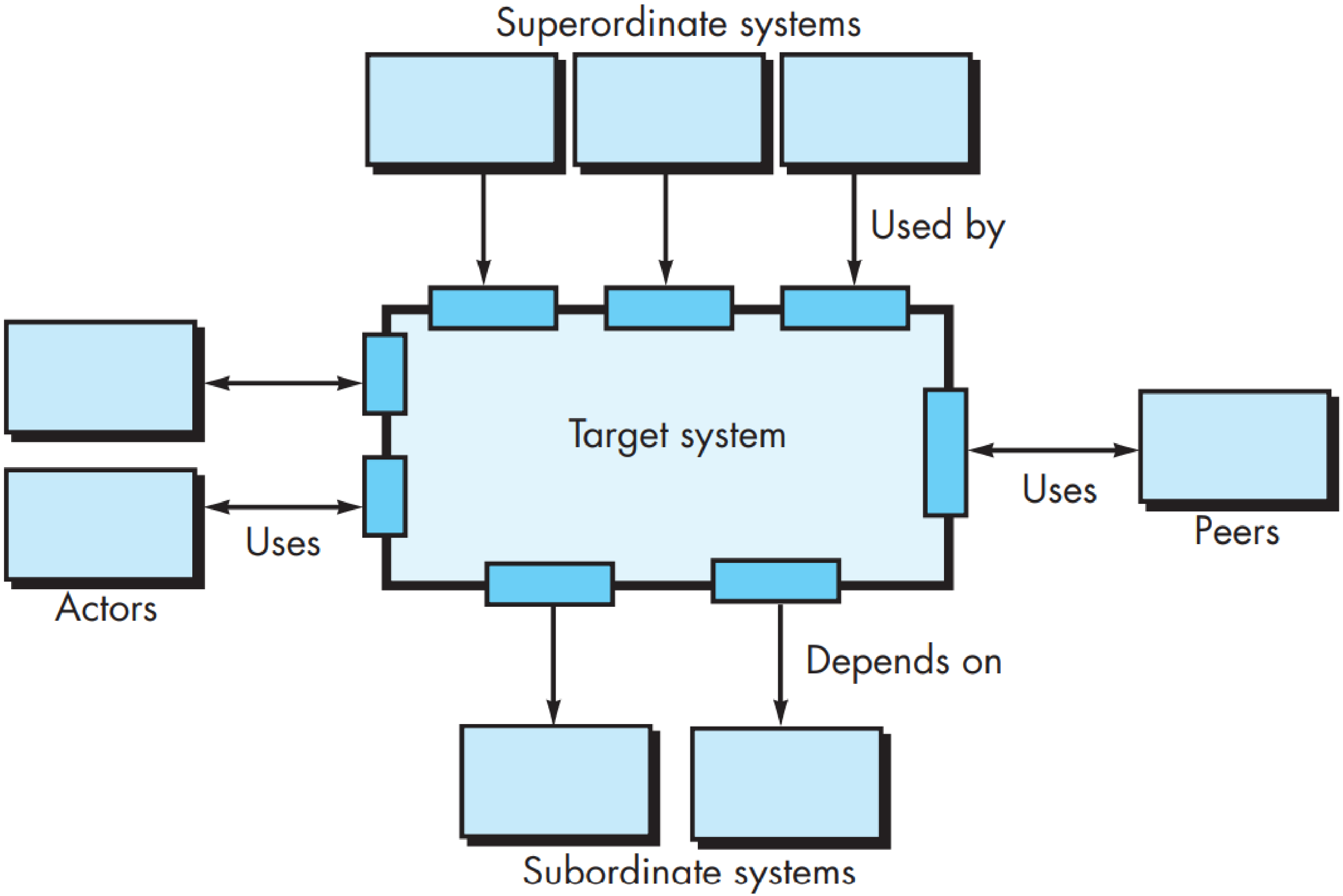
1. Representing the System in Context

- At the architectural design level, a software architect uses an **architectural context diagram (ACD)** to model the manner in which software interacts with entities external to its boundaries.
- The generic structure of the architectural context diagram is illustrated in the figure.

Figure 13.5

Architectural context diagram

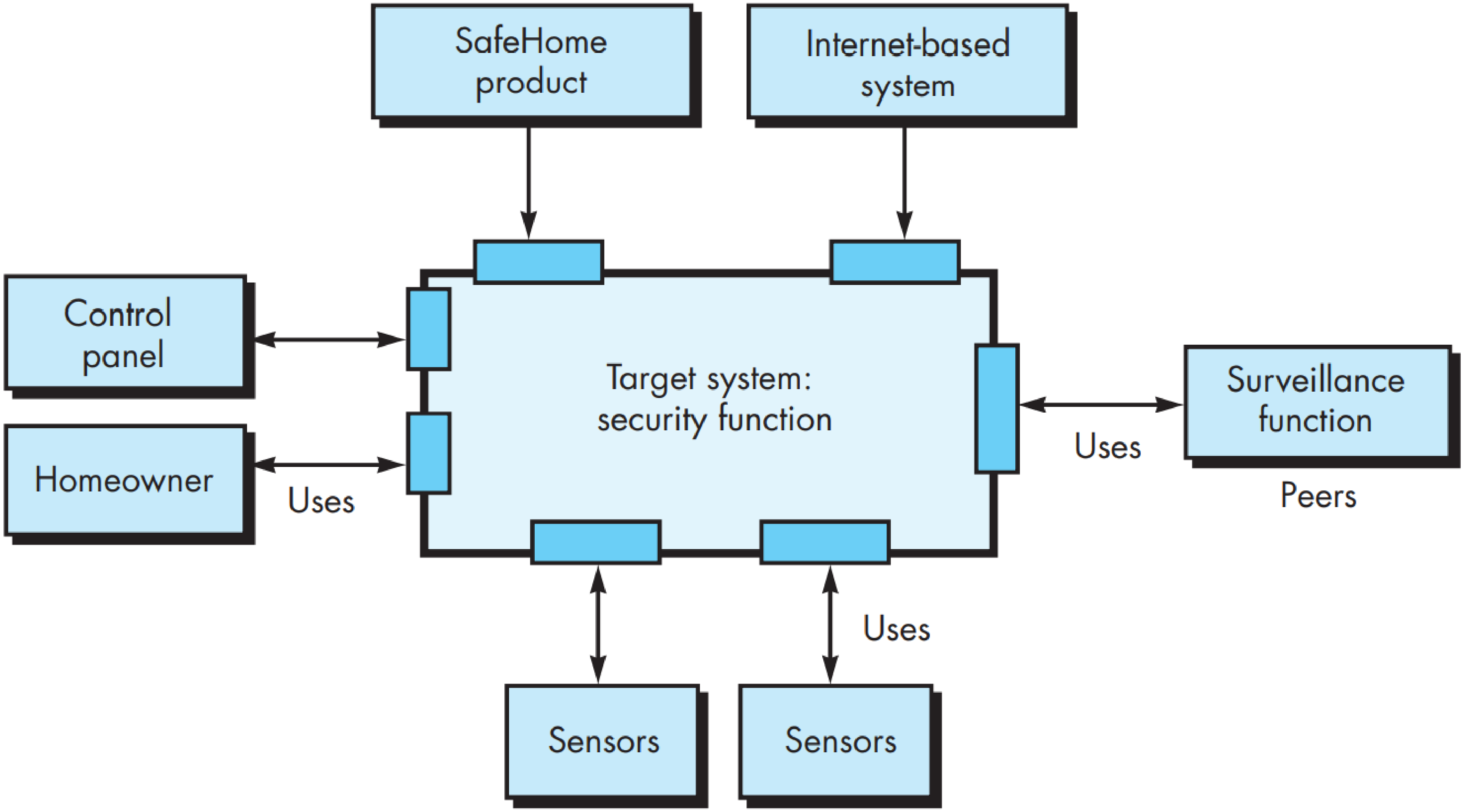
Source: Adapted from [Bos00].



- The systems that interoperate with the target system are represented as:
    1. **Superordinate systems** → those systems that use the target system as part of some higher-level processing scheme.
    2. **Subordinate systems** → those systems that are used by the target system and provide data or processing that are necessary to complete target system functionality.
    3. **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.
    4. **Actors** → entities (people, devices) that interact with the target system by producing or consuming information that is necessary for requisite processing.
- Each of these external entities communicates with the target system through an interface.

FIGURE 13.6

Architectural context diagram for the *SafeHome* security function



2. 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.
- Archetypes are the abstract building blocks of an architectural design.
- Ex: SafeHome home security function, the following archetypes are defined:
  1. **Node** → Represents a cohesive collection of input and output elements of the home security function.
  2. **Detector** → An abstraction that encompasses all sensing equipment that feeds information into the target system.
  3. **Indicator** → An abstraction that represents all mechanisms for indicating that an alarm condition is occurring.
  4. **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.

3. Refining the Architecture into Components

- As the software architecture is refined into components, the structure of the system begins to emerge.
- The application domain is one source for the derivation and refinement of components.
- Another source is the infrastructure domain.
- The architecture must accommodate many infrastructure components that enable application components but have no business connection to the application domain.
- The interfaces depicted in the architecture context diagram imply one or more specialized components that process the data that flows across the interface.

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

4. Describing Instantiations of the System

- **Instantiation of the architecture** → the architecture is applied to a specific problem with the intent of demonstrating that the structure and components are appropriate.

5. Architectural Design for Web Apps

- **WebApps** → are client-server applications typically structured using multi-layered architectures, including:
    1. a user interface or view layer,
    2. a controller layer which directs the flow of information to and from the client browser based on a set of business rules, and
    3. a content or model layer that may also contain the business rules for the WebApp.

- The user interface for a WebApp is designed around the characteristics of the web browser running on the client machine.
- The architectural design of a WebApp is also influenced by the structure of the content that needs to be accessed by the client.
- The architectural components (Web pages) of a WebApp are designed to allow control to be passed to other system components, allowing very flexible navigation structures.
- The physical location of media and other content resources also influences the architectural choices made by software engineers.

6. Architectural Design for Mobile Apps

- Mobile apps are typically structured using multi-layered architectures, including:
  1. a user interface layer,
  2. a business layer, and
  3. a data layer.
- Mobile devices differ from one another in terms of their physical characteristics, software, and hardware. Each of these attributes shapes the direction of the architectural alternatives that can be selected.
- A number of considerations that can influence the architectural design of a mobile app are:
  1. the type of web client (thin or rich) to be built,
  2. the categories of devices (e.g., smartphones, tablets) that are supported,
  3. the degree of connectivity (occasional or persistent) required,
  4. the bandwidth required,
  5. the constraints imposed by the mobile platform,
  6. the degree to which reuse and maintainability are important, and
  7. device resource constraints (e.g., battery life, memory size, processor speed).

COMPONENT LEVEL DESIGN

What is a component?

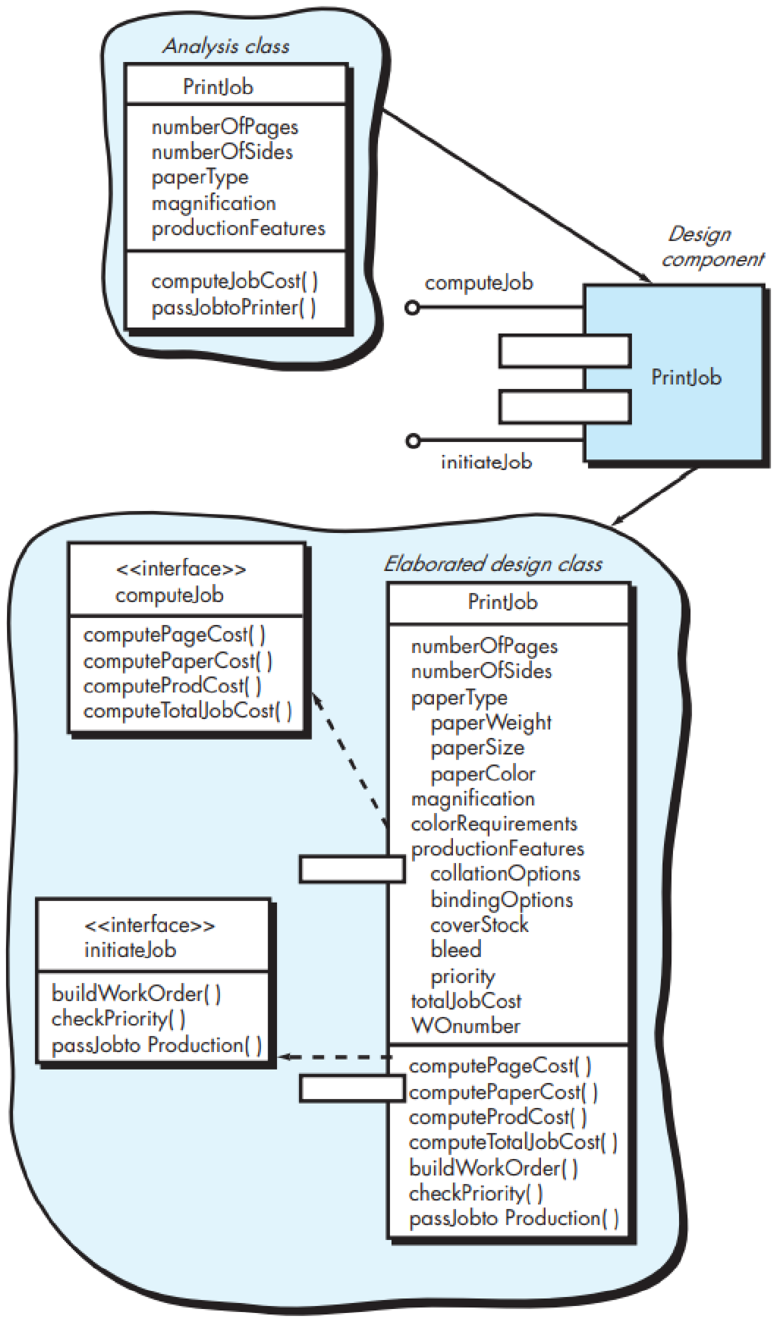
- A **component** is a modular building block for computer software.
- The *OMG Unified Modeling Language Specification* defines a **component** as “a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces.”

1. An Object-Oriented View

- From an object-oriented viewpoint, a **component** is a set of collaborating classes.
- Each class within a component has been fully elaborated to include all attributes and operations that are relevant to its implementation.
- As part of the design elaboration, all interfaces must also be defined.
- Analysis modeling and design modeling are both iterative actions. Elaborating the original analysis class may require additional analysis steps, which are then followed with design modeling steps to represent the elaborated design class (the details of the component).

FIGURE 14.1

Elaboration of a design component



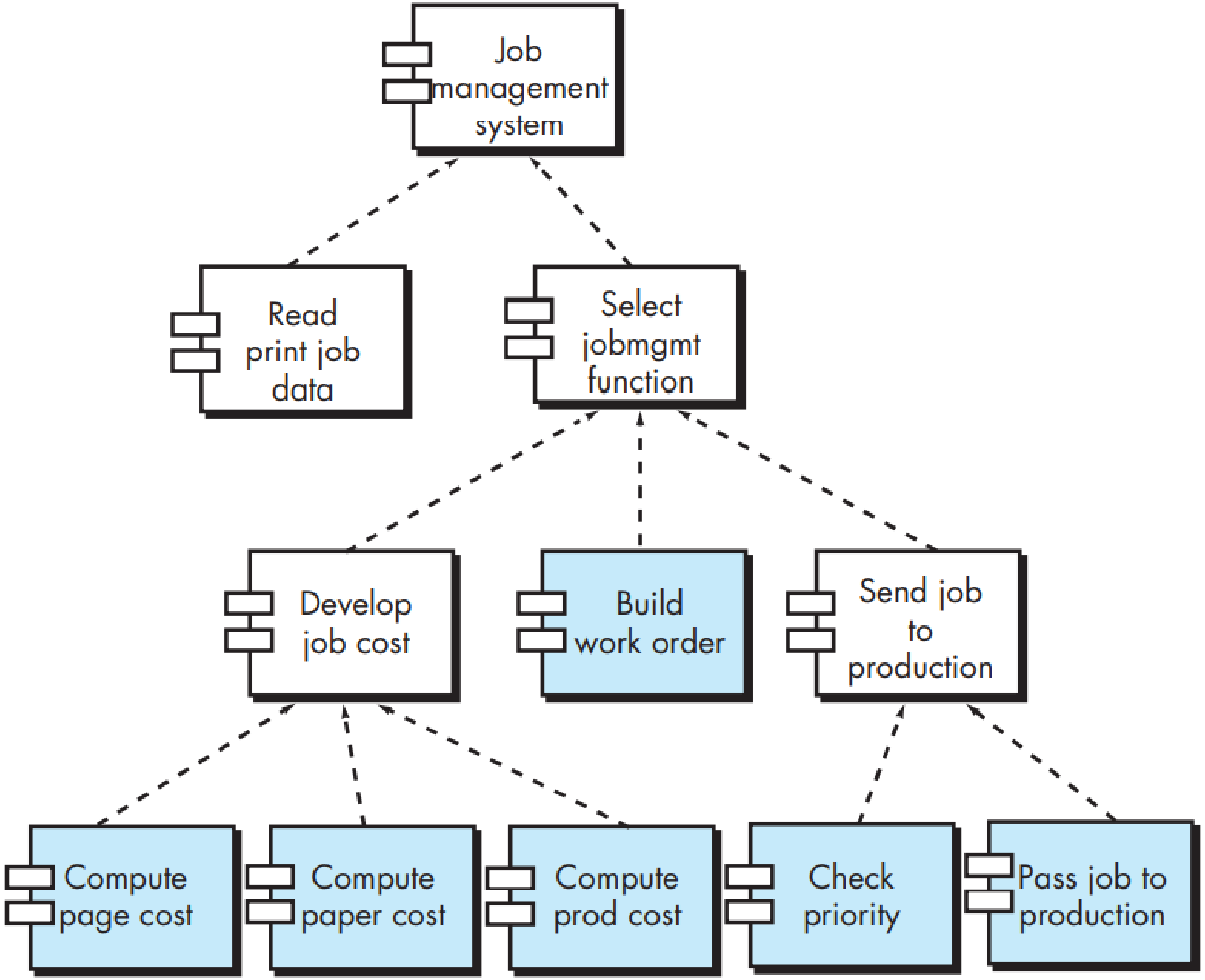
- The elaboration activity is applied to every component defined as part of the architectural design.
- Once it is completed, further elaboration is applied to each attribute, operation, and interface.
- The data structures appropriate for each attribute must be specified.
- In addition, the algorithmic detail required to implement the processing logic associated with each operation is designed.
- Finally, the mechanisms required to implement the interface are designed. For object-oriented software, this may encompass the description of all messaging that is required to effect communication between objects within the system.

2. 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, and an interface that enables the component to be invoked and data to be passed to it.
- A traditional component, also called a **module**, resides within the software architecture and serves one of three important roles:
  1. a **control component** that coordinates the invocation of all other problem domain components,
  2. a **problem domain component** that implements a complete or partial function that is required by the customer, or
  3. an **infrastructure component** that is responsible for functions that support the processing required in the problem domain.

- Like object-oriented components, traditional software components are derived from the analysis model.
- The component elaboration element of the analysis model serves as the basis for the derivation.
- Each component represented in the component hierarchy is mapped into a module hierarchy.
- Control components (modules) reside near the top of the hierarchy (program architecture), and problem domain components tend to reside toward the bottom of the hierarchy.
- To achieve effective modularity, design concepts like functional independence are applied as components are elaborated.

FIGURE 14.2 Structure chart for a traditional system



3. A Process-Related View

- The object-oriented and traditional views of component-level design → you have to create a new component based on specifications derived from the requirements model.
- A catalog of proven design or code-level components is made available to you as design work proceeds.
- As the software architecture is developed, you choose components or design patterns from the catalog and use them to populate the architecture.
- Because these components have been created with reusability in mind, a complete description of their interface, the function(s) they perform, and the communication and collaboration they require are all available to you.

Designing Class-Based Components

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

1. Basic Design Principles

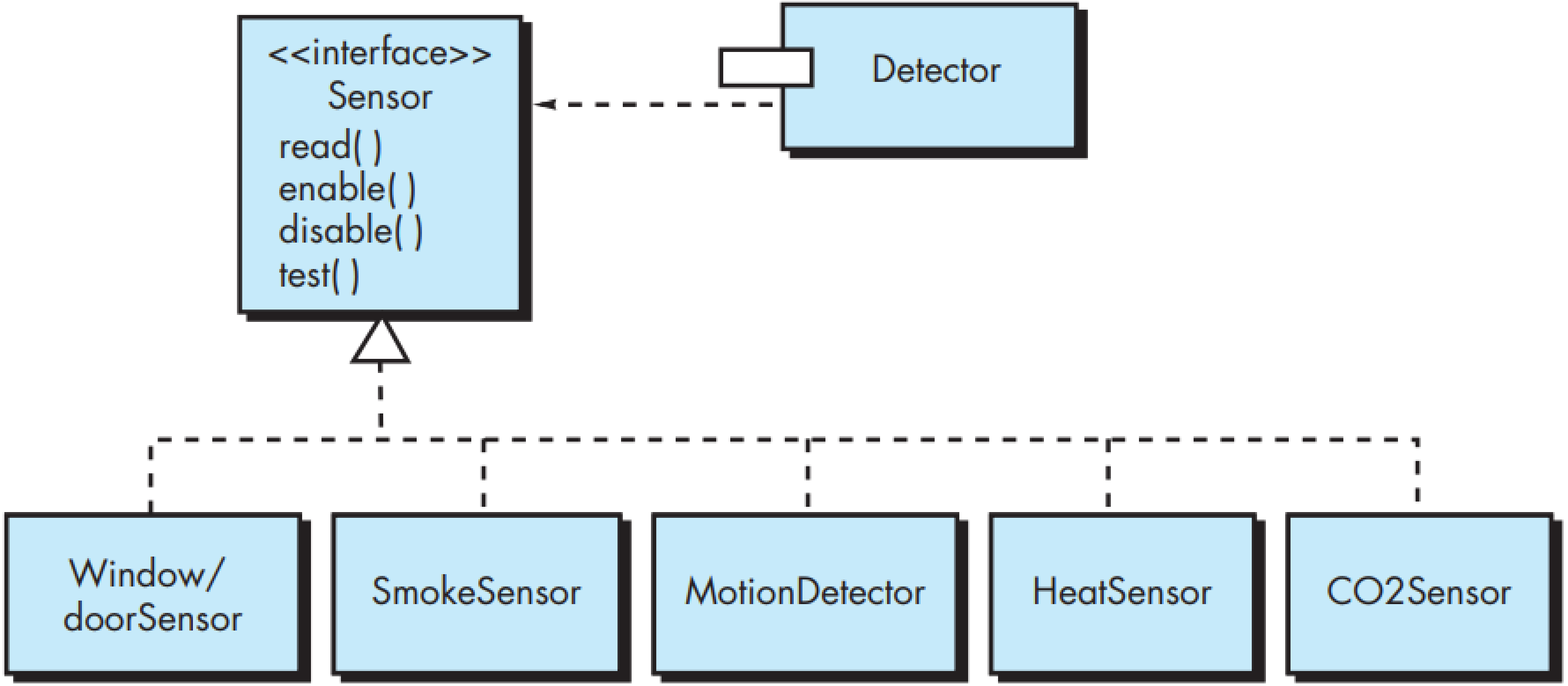
- 4 basic design principles:
  1. The Open-Closed Principle (OCP)
  2. The Liskov Substitution Principle (LSP)
  3. Dependency Inversion Principle (DIP)
  4. The Interface Segregation Principle (ISP)
- Additional design principles:
  1. The Release Reuse Equivalency Principle (REP)
  2. The Common Closure Principle (CCP)
  3. The Common Reuse Principle (CRP)

The Open-Closed Principle (OCP):

- “ A module [component] should be open for extension but closed for modification”.
- Stated simply, you should specify the component in a way that allows it to be extended (within the functional domain that it addresses) without the need to make internal (code or logic-level) modifications to the component itself.
- To accomplish this, you create abstractions that serve as a buffer between the functionality that is likely to be extended and the design class itself.
- Ex: assume that the SafeHome security function makes use of a Detector class that must check the status of each type of security sensor. If internal processing logic is implemented as a sequence of if-then-else constructs, each addressing a different sensor type, the addition of a new sensor type will require additional internal processing logic. This is a violation of OCP.

FIGURE 14.4

Following the OCP



The Liskov Substitution Principle (LSP):

- “ Subclasses should be substitutable for their base classes”.
- 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.
- A “contract” is a precondition that must be true before the component uses a base class and a postcondition that should be true after the component uses a base class.
- When you create derived classes, be sure they conform to the pre- and postconditions.

Dependency Inversion Principle (DIP):

- “ Depend on abstractions. Do not depend on concretions”.
- The more a component depends on other concrete components (rather than on abstractions such as an interface), the more difficult it will be to extend.

The Interface Segregation Principle (ISP):

- “Many client-specific interfaces are better than one general purpose interface”.
- There are many instances in which multiple client components use the operations provided by a server class.
- ISP suggests that you 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 the specialized interfaces.

- Ex: the FloorPlan class that is used for the SafeHome security and surveillance functions.
- For the security functions, FloorPlan is used only during configuration activities and uses the operations placeDevice(), showDevice(), groupDevice(), and removeDevice() → to place, show, group, and remove sensors from the floor plan.
- The SafeHome surveillance function uses the 4 operations noted for security, but also requires special operations to manage cameras: showFOV() and showDeviceID().
- Hence, the ISP suggests that client components from the two SafeHome functions have specialized interfaces defined for them.
- The interface for security would encompass only the operations placeDevice(), showDevice(), groupDevice(), and removeDevice().
- The interface for surveillance would incorporate the operations placeDevice(), showDevice(), groupDevice(), and removeDevice(), along with showFOV() and showDeviceID()

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.
- Rather than addressing each class individually, it is often advisable to group reusable classes into packages that can be managed and controlled as newer versions evolve.

The Common Closure Principle (CCP):

- “Classes that change together belong together.”
- When classes are packaged as part of a design, they should address the same functional or behavioral area.
- When some characteristic of that area must change, it is likely that only those classes within the package will require modification. This leads to more effective change control and release management.

The Common Reuse Principle (CRP):

- “Classes that aren’t reused together should not be grouped together”.
- When one or more classes with a package changes, the release number of the package changes.
- All other classes or packages that rely on the package that has been changed must now update to the most recent release of the package and be tested.
- 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.

2. Component-Level Design Guidelines

- These guidelines are apply to components, their interfaces, and the dependencies and inheritance characteristics that have an impact on the resultant design.

1. **Components:** Naming conventions should be established for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model. Architectural component names should be drawn from the problem domain and should have meaning to all stakeholders who view the architectural model.
2. **Interfaces:** provide important information about communication and collaboration. Recommendations that are intended to simplify the visual nature of UML component diagrams : (1) lollipop representation of an interface should be used in lieu of the more formal UML box and dashed arrow approach, when diagrams grow complex; (2) for consistency, interfaces should flow from the left-hand side of the component box; (3) only those interfaces that are relevant to the component under consideration should be shown, even if other interfaces are available.
3. **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). In addition, components' interdependencies should be represented via interfaces, rather than by representation of a component-to-component dependency.

3. Cohesion

- It 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.
- Different types of cohesion:
  1. **Functional:** occurs when a module performs one and only one computation and then returns a result.
  2. **Layer:** occurs when a higher layer accesses the services of a lower layer, but lower layers do not access higher layers.
  3. **Communicational:** All operations that access the same data are defined within one class.
- Classes and components that exhibit functional, layer, and communicational cohesion are relatively easy to implement, test, and maintain.

4. 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.
- Class coupling can manifest itself in a variety of ways:

  1. **Content coupling** occurs when one component "surreptitiously modifies data that is internal to another component". This violates information hiding—a basic design concept.

  2. **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.

  3. **External coupling** occurs when a component communicates or collaborates with infrastructure components (e.g., operating system functions, database capability, telecommunication 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:
- **Step 1: Identify all design classes that correspond to the problem domain**.

  Using the requirements and architectural model, each analysis class and architectural component is elaborated.
- **Step 2. Identify all design classes that correspond to the infrastructure domain.**

  These classes are not described in the requirements model and are often missing from the architecture model, but they must be described at this point. The classes and components in this category include GUI components, operating system components, and object and data management components.

- **Step 3: Elaborate all design classes that are not acquired as reusable components.**

  Elaboration requires that all interfaces, attributes, and operations necessary to implement the class be described in detail. Design heuristics (e.g., component cohesion and coupling) must be considered as this task is conducted.

  - **Step 3a. Specify message details when classes or components collaborate.**

    The requirements model makes use of a collaboration diagram to show how analysis classes collaborate with one another. Although this design activity is optional, it can be used as a precursor to the specification of interfaces that show how components within the system communicate and collaborate. Messages are passed between objects.

- **Step 3b. Identify appropriate interfaces for each component.**

  Within the context of component-level design, a UML interface is “a group of externally visible (i.e., public) operations. The interface contains no internal structure, it has no attributes, no associations. . .” Stated more formally, an interface is the equivalent of an abstract class that provides a controlled connection between design classes. Every operation within the abstract class (the interface) should be cohesive.

- **Step 3c. Elaborate attributes and define data types and data structures required to implement them.**

  In general, data structures and types used to define attributes are defined within the context of the programming language that is to be used for implementation. UML defines an attribute’s data type using the following syntax:

  name : type-expression = initial-value {property string}

  where name is the attribute name, type expression is the data type, initial value is the value that the attribute takes when an object is created, and property-string defines a property or characteristic of the attribute.

- **Step 3d. Describe processing flow within each operation in detail.**

  This may be accomplished using a programming language-based pseudocode or with a UML activity diagram. Each software component is elaborated through a number of iterations that apply the stepwise refinement concept. The first iteration defines each operation as part of the design class. In every case, the operation should be characterized in a way that ensures high cohesion. The next iteration does little more than expand the operation name.

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

  Databases and files normally transcend the design description of an individual component. In most cases, these persistent data stores are initially specified as part of architectural design. However, as design elaboration proceeds, it is often useful to provide additional detail about the structure and organization of these persistent data sources.

- **Step 5: Develop and elaborate behavioral representations for a class or component.**

  UML state diagrams were used as part of the requirements model to represent the externally observable behavior of the system and the more localized behavior of individual analysis classes. During component-level design, it is sometimes necessary to model the behavior of a design class. The dynamic behavior of an object is affected by events that are external to it and the current state of the object. To understand the dynamic behavior of an object, you should examine all use cases that are relevant to the design class throughout its life. These use cases provide information that helps you to delineate the events that affect the object and the states in which the object resides as time passes and events occur. The transitions between states (driven by events) is represented using a UML statechart. The transition from one state (represented by a rectangle with rounded corners) to another occurs as a consequence of an event that takes the form:

  Event-name (parameter-list) [guard-condition] / action expression

  where event-name identifies the event, parameter-list incorporates data that are associated with the event, guard-condition is written in Object Constraint Language (OCL) and specifies a condition that must be met before the event can occur, and action expression defines an action that occurs as the transition takes place.

- **Step 6: Elaborate deployment diagrams to provide additional implementation detail.**

  Deployment diagrams are used as part of architectural design and are represented in descriptor form. In this form, major system functions are represented within the context of the computing environment that will house them. During component-level design, deployment diagrams can be elaborated to represent the location of key packages of components.

- **Step 7: Refactor every component-level design representation and always consider alternatives.**

  Design is an iterative process. The first component-level model you 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 for Web-Apps

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

1. Content Design at the Component Level

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

2. Functional Design at the Component Level

- WebApp functionality is delivered as a series of components developed in parallel with the information architecture to ensure consistency.
- We begin by considering both the requirements model and the initial information architecture and then examining how functionality affects the user’s interaction with the application, the information that is presented, and the user tasks that are conducted.
- During architectural design, WebApp content and functionality are combined to create a functional architecture.
- A **functional architecture** is a representation of the functional domain of the WebApp and describes the key functional components in the WebApp and how these components interact with each other.