Introduction

Lecture1

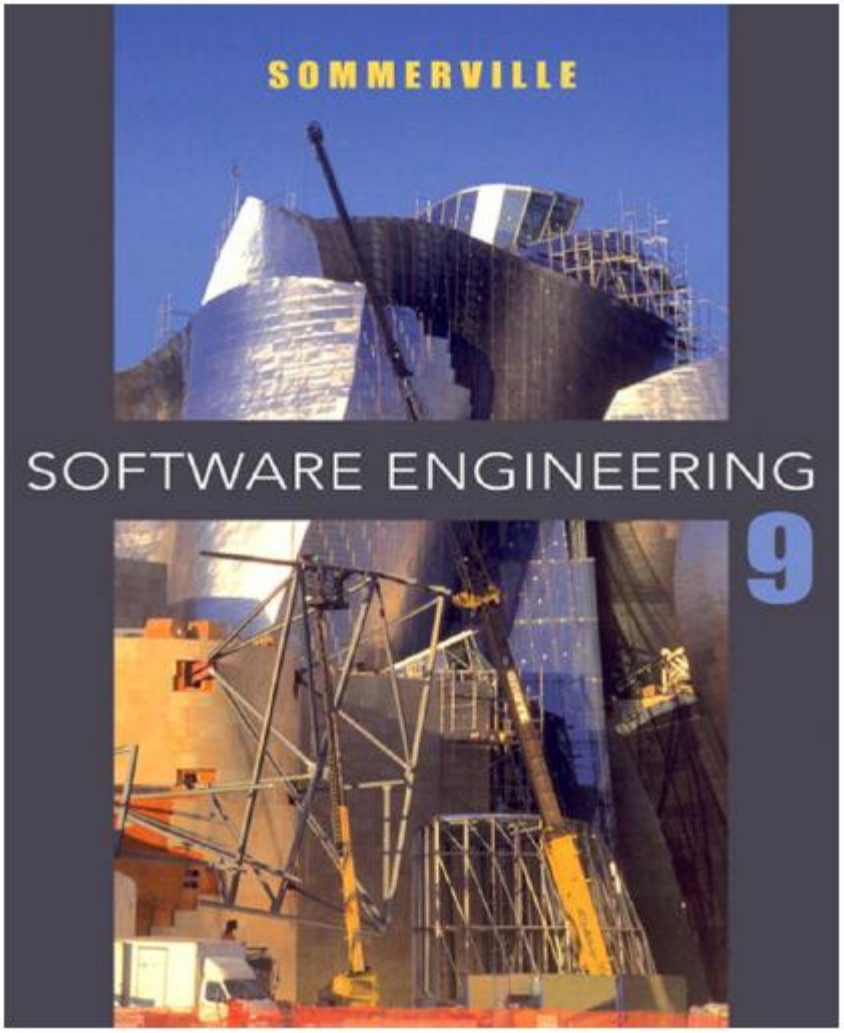
Questions

- What is “software”?
- What is “software modeling”?
- What is “software analysis”?

Course material

- **MODELING FOUNDATION**
- Modeling principles (e.g., decomposition, abstraction, generalization, projection/views, and use of formal approaches)
- Preconditions, postconditions, invariants, and design by contract
- Introduction to mathematical models and formal notation
-
- **TYPE OF MODELS**
- information modeling (e.g., entity-relationship modeling and class diagrams)
- Behavioral modeling (e.g., state diagrams, use case analysis, interaction diagrams, failure modes and effects analysis, and fault tree analysis)
- Architectural modeling (e.g., architectural patterns and component diagrams)
- Domain modeling (e.g., domain engineering approaches)
- Enterprise modeling (e.g., business processes, organizations, goals, and workflow)
- Modeling embedded systems (e.g., real-time schedule analysis, and interface protocols)
-
- **ANALYSIS FUNDAMENTAL**
- Analyzing form (e.g., completeness, consistency, and robustness)
- Analyzing correctness (e.g., static analysis, simulation, and model checking)
- Analyzing dependability (e.g., failure mode analysis and fault trees)
- Formal analysis (e.g., theorem proving)
-

References



Reference [1]

- You need to read the chapters: 1

What is software?

- Computer programs and associated documentation such as requirements, design models and user manuals.
- Software products may be developed for a particular customer or may be developed for a general market.
- Software products may be
  - Generic developed to be sold to a range of different customers e.g. PC software such as Excel or Word.
  - Bespoke (custom) developed for a single customer according to their specification.
- New software can be created by developing new programs, configuring generic software systems or reusing existing software.

Software engineering

- The economies of ALL developed nations are dependent on software.
- More and more systems are software controlled
- Software engineering is concerned with theories, methods and tools for professional software development.
- Expenditure on software represents a significant fraction of GNP (Gross National Product) in all developed countries.

Software Costs

- Software costs often dominate computer system costs. The costs of software on a PC are often greater than the hardware cost.
- Software costs more to maintain than it does to develop. For systems with a long life, maintenance costs may be several times development costs.

Software products

- Generic products
  - Stand-alone systems that are marketed and sold to any customer who wishes to buy them.
  - Examples – PC software such as graphics programs, project management tools; CAD software; software for specific markets such as appointments systems for dentists.
- Customized products
  - Software that is commissioned by a specific customer to meet their own needs.
  - Examples – embedded control systems, air traffic control software, traffic monitoring systems.

Product specification

- Generic products
  - The specification of what the software should do is owned by the software developer and decisions on software change are made by the developer.
- Customized products
  - The specification of what the software should do is owned by the customer for the software and they make decisions on software changes that are required.

Question	Answer
What is software?	Computer programs and associated documentation. Software products may be developed for a particular customer or may be developed for a general market.
What are the attributes of good software?	Good software should deliver the required functionality and performance to the user and should be maintainable, dependable and usable.
What is software engineering?	Software engineering is an engineering discipline that is concerned with all aspects of software production.
What are the fundamental software engineering activities?	Software specification, software development, software validation and software evolution.
What is the difference between software engineering and computer science?	Computer science focuses on theory and fundamentals; software engineering is concerned with the practicalities of developing and delivering useful software.
What is the difference between software engineering and system engineering?	System engineering is concerned with all aspects of computer-based systems development including hardware, software and process engineering. Software engineering is part of this more general process.

Question	Answer
What are the key challenges facing software engineering?	Coping with increasing diversity, demands for reduced delivery times and developing trustworthy software.
What are the costs of software engineering?	Roughly 60% of software costs are development costs, 40% are testing costs. For custom software, evolution costs often exceed development costs.
What are the best software engineering techniques and methods?	While all software projects have to be professionally managed and developed, different techniques are appropriate for different types of system. For example, games should always be developed using a series of prototypes whereas safety critical control systems require a complete and analyzable specification to be developed. You can't, therefore, say that one method is better than another.
What differences has the web made to software engineering?	The web has led to the availability of software services and the possibility of developing highly distributed service-based systems. Web-based systems development has led to important advances in programming languages and software reuse.

Product characteristic	Description
Maintainability	Software should be written in such a way so that it can evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable requirement of a changing business environment.
Dependability and security	Software dependability includes a range of characteristics including reliability, security and safety. Dependable software should not cause physical or economic damage in the event of system failure. Malicious users should not be able to access or damage the system.
Efficiency	Software should not make wasteful use of system resources such as memory and processor cycles. Efficiency therefore includes responsiveness, processing time, memory utilisation, etc.
Acceptability	Software must be acceptable to the type of users for which it is designed. This means that it must be understandable, usable and compatible with other systems that they use.

Importance of software engineering

- More and more, individuals and society rely on advanced software systems. We need to be able to produce reliable and trustworthy systems economically and quickly.
- It is usually cheaper, in the long run, to use software engineering methods and techniques for software systems rather than just write the programs as if it was a personal programming project. For most types of system, the majority of costs are the costs of changing the software after it has gone into use.

Software process activities

- Software specification, where customers and engineers define the software that is to be produced and the constraints on its operation.
- Software development, where the software is designed and programmed.
- Software validation, where the software is checked to ensure that it is what the customer requires.
- Software evolution, where the software is modified to reflect changing customer and market requirements.

Application types

- Stand-alone applications
  - These are application systems that run on a local computer, such as a PC. They include all necessary functionality and do not need to be connected to a network.
- Interactive transaction-based applications
  - Applications that execute on a remote computer and are accessed by users from their own PCs or terminals. These include web applications such as e-commerce applications.
- Embedded control systems
  - These are software control systems that control and manage hardware devices. Numerically, there are probably more embedded systems than any other type of system.

Application types

- Batch processing systems
  - These are business systems that are designed to process data in large batches. They process large numbers of individual inputs to create corresponding outputs.
- Entertainment systems
  - These are systems that are primarily for personal use and which are intended to entertain the user.
- Systems for modeling and simulation
  - These are systems that are developed by scientists and engineers to model physical processes or situations, which include many, separate, interacting objects.

Application types

- Data collection systems
  - These are systems that collect data from their environment using a set of sensors and send that data to other systems for processing.
- Systems of systems
  - These are systems that are composed of a number of other software systems.

Software engineering fundamentals

- Some fundamental principles apply to all types of software system, irrespective of the development techniques used:
  - Systems should be developed using a managed and understood development process. Of course, different processes are used for different types of software.
  - Dependability and performance are important for all types of system.
  - Understanding and managing the software specification and requirements (what the software should do) are important.
  - Where appropriate, you should reuse software that has already been developed rather than write new software.

Software engineering and the web

- The Web is now a platform for running application and organizations are increasingly developing web-based systems rather than local systems.
- Web services allow application functionality to be accessed over the web.
- Cloud computing is an approach to the provision of computer services where applications run remotely on the ‘cloud’.
  - Users do not buy software buy pay according to use.

Web software engineering

- Software reuse is the dominant approach for constructing web-based systems.
  - When building these systems, you think about how you can assemble them from pre-existing software components and systems.
- Web-based systems should be developed and delivered incrementally.
  - It is now generally recognized that it is impractical to specify all the requirements for such systems in advance.
- User interfaces are constrained by the capabilities of web browsers.
  - Technologies such as AJAX allow rich interfaces to be created within a web browser but are still difficult to use. Web forms with local scripting are more commonly used.

Web-based software engineering

- Web-based systems are complex distributed systems but the fundamental principles of software engineering discussed previously are as applicable to them as they are to any other types of system.
- The fundamental ideas of software engineering, discussed in the previous section, apply to web-based software in the same way that they apply to other types of software system.

Modeling Principles

Lecture2

System modeling

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

Existing and planned system models

- Models of the existing system are used during requirements engineering. They help clarify what the existing system does and can be used as a basis for discussing its strengths and weaknesses. These then lead to requirements for the new system.
- Models of the new system are used during requirements engineering to help explain the proposed requirements to other system stakeholders. Engineers use these models to discuss design proposals and to document the system for implementation.
- In a model-driven engineering process, it is possible to generate a complete or partial system implementation from the system model.

System perspectives

- An external perspective, where you model the context or environment of the system.
- An interaction perspective, where you model the interactions between a system and its environment, or between the components of a system.
- A structural perspective, where you model the organization of a system or the structure of the data that is processed by the system.
- A behavioral perspective, where you model the dynamic behavior of the system and how it responds to events.

UML diagram types

- Activity diagrams, which show the activities involved in a process or in data processing .
- Use case diagrams, which show the interactions between a system and its environment.
- Sequence diagrams, which show interactions between actors and the system and between system components.
- Class diagrams, which show the object classes in the system and the associations between these classes.
- State diagrams, which show how the system reacts to internal and external events.

Use of graphical models

- As a means of facilitating discussion about an existing or proposed system
  - Incomplete and incorrect models are OK as their role is to support discussion.
- As a way of documenting an existing system
  - Models should be an accurate representation of the system but need not be complete.
- As a detailed system description that can be used to generate a system implementation
  - Models have to be both correct and complete.

Context models

- Context models are used to illustrate the operational context of a system - they show what lies outside the system boundaries.
- Social and organisational concerns may affect the decision on where to position system boundaries.
- Architectural models show the system and its relationship with other systems.

System boundaries

- System boundaries are established to define what is inside and what is outside the system.
  - They show other systems that are used or depend on the system being developed.
- The position of the system boundary has a profound effect on the system requirements.
- Defining a system boundary is a political judgment
  - There may be pressures to develop system boundaries that increase / decrease the influence or workload of different parts of an organization.

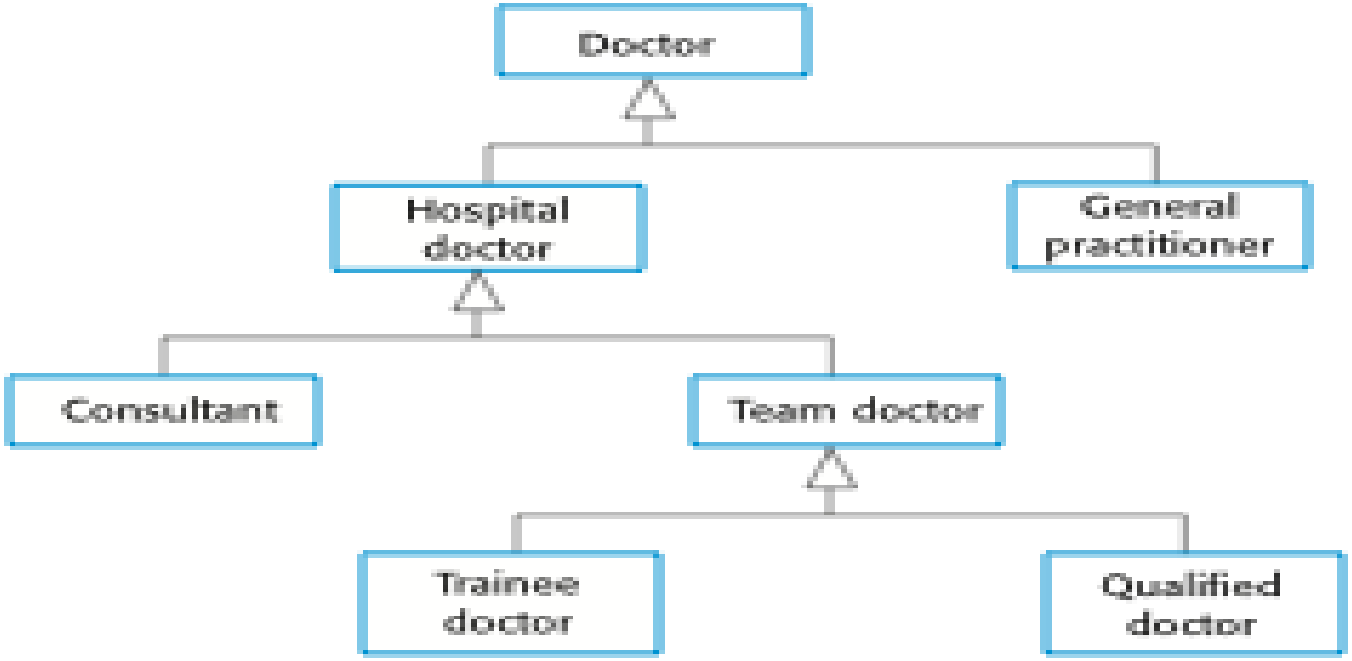
Generalization

- Generalization is an everyday technique that we use to manage complexity.
- Rather than learn the detailed characteristics of every entity that we experience, we place these entities in more general classes (animals, cars, houses, etc.) and learn the characteristics of these classes.
- This allows us to infer that different members of these classes have some common characteristics e.g. squirrels and rats are rodents.

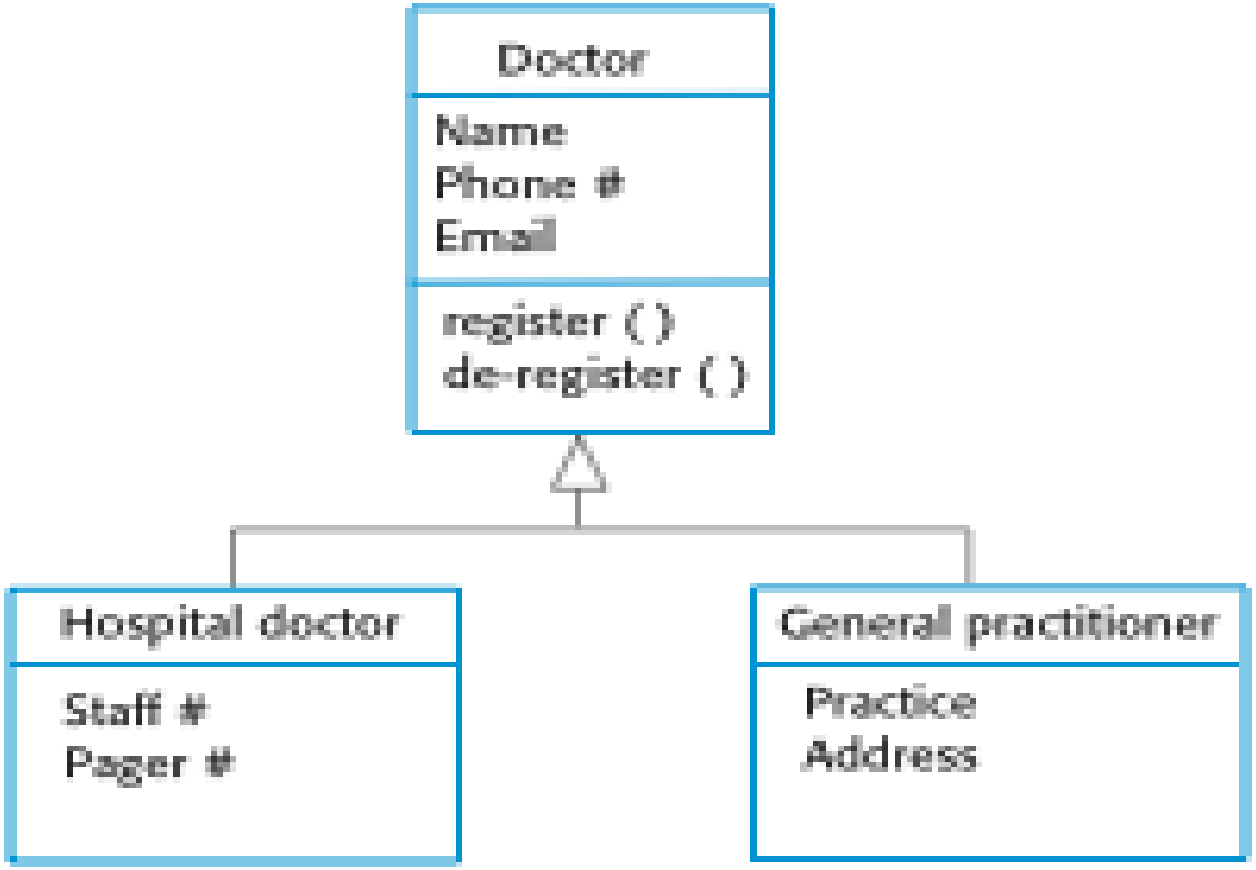
Generalization

- In modeling systems, it is often useful to examine the classes in a system to see if there is scope for generalization. If changes are proposed, then you do not have to look at all classes in the system to see if they are affected by the change.
- In object-oriented languages, such as Java, generalization is implemented using the class inheritance mechanisms built into the language.
- In a generalization, the attributes and operations associated with higher-level classes are also associated with the lower-level classes.
- The lower-level classes are subclasses inherit the attributes and operations from their superclasses. These lower-level classes then add more specific attributes and operations.

A generalization hierarchy



A generalization hierarchy with added detail



Abstraction

- Many levels of abstraction
- At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment.
- at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented.
- The psychological notion of "abstraction" permits one to concentrate on a problem at some level of generalization without regard to irrelevant low level details; use of abstraction also permits one to work with concepts and terms that are familiar in the problem environment without having to transform them to an unfamiliar structure . . .

Component composition

- The process of assembling components to create a system.
- Composition involves integrating components with each other and with the component infrastructure.
- Normally you have to write ‘glue code’ to integrate components.

Component composition

- Component composition is the process of ‘wiring’ components together to create a system.
- When composing reusable components, you normally have to write adaptors to reconcile different component interfaces.
- When choosing compositions, you have to consider required functionality, non-functional requirements and system evolution.

Types of composition

- Sequential composition where the composed components are executed in sequence. This involves composing the provides interfaces of each component.
- Hierarchical composition where one component calls on the services of another. The provides interface of one component is composed with the requires interface of another.
- Additive composition where the interfaces of two components are put together to create a new component. Provides and requires interfaces of integrated component is a combination of interfaces of constituent components.

Composition trade-offs

- When composing components, you may find conflicts between functional and non-functional requirements, and conflicts between the need for rapid delivery and system evolution.
- You need to make decisions such as:
  - What composition of components is effective for delivering the functional requirements?
  - What composition of components allows for future change?
  - What will be the emergent properties of the composed system?

Preconditions, Post conditions, design by contract

Lecture3

Photo Library documentation

“This method adds a photograph to the library and associates the photograph identifier and catalogue descriptor with the photograph.”

“what happens if the photograph identifier is already associated with a photograph in the library?”

“is the photograph descriptor associated with the catalogue entry as well as the photograph i.e. if I delete the photograph, do I also delete the catalogue information?”

The Object Constraint Language

- The Object Constraint Language (OCL) has been designed to define constraints that are associated with UML models.
- It is based around the notion of pre and post condition specification – common to many formal methods.

The OCL description of the Photo Library interface

•-- The context keyword names the component to which the conditions apply

•**context** addItem

•-- The preconditions specify what must be true before execution of addItem
•**pre**: PhotoLibrary.libSize() > 0
•PhotoLibrary.retrieve(pid) = null

•-- The postconditions specify what is true after execution
•**post**:libSize () = libSize()@pre + 1
•PhotoLibrary.retrieve(pid) = p
•PhotoLibrary.catEntry(pid) = photodesc

•**context** delete

•**pre**: PhotoLibrary.retrieve(pid) <> null ;

•**post**: PhotoLibrary.retrieve(pid) = null
•PhotoLibrary.catEntry(pid) = PhotoLibrary.catEntry(pid)@pre
•PhotoLibrary.libSize() = libSize()@pre—1

Photo library conditions

- As specified, the OCL associated with the Photo Library component states that:
  - There must not be a photograph in the library with the same identifier as the photograph to be entered;
  - The library must exist - assume that creating a library adds a single item to it;
  - Each new entry increases the size of the library by 1;
  - If you retrieve using the same identifier then you get back the photo that you added;
  - If you look up the catalogue using that identifier, then you get back the catalogue entry that you made.

Mathematical models and formal notation

Lecture4

Formal specification

- Formal specification is part of a more general collection of techniques that are known as ‘formal methods’.
- These are all based on mathematical representation and analysis of software.
- Formal methods include
  - Formal specification;
  - Specification analysis and proof;
  - Transformational development;
  - Program verification.

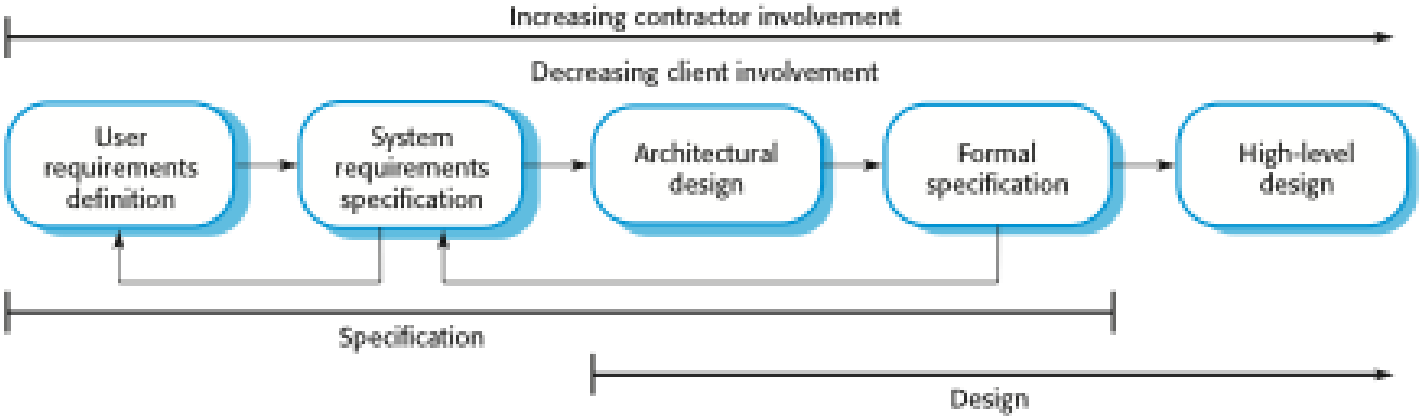
Use of formal methods

- The principal benefits of formal methods are in reducing the number of faults in systems.
- Consequently, their main area of applicability is in critical systems engineering. There have been several successful projects where formal methods have been used in this area.
- In this area, the use of formal methods is most likely to be cost-effective because high system failure costs must be avoided.

Specification in the software process

- Specification and design are inextricably intermingled.
- Architectural design is essential to structure a specification and the specification process.
- Formal specifications are expressed in a mathematical notation with precisely defined vocabulary, syntax and semantics.

Formal specification in a plan-based software process



Benefits of formal specification

- Developing a formal specification requires the system requirements to be analyzed in detail. This helps to detect problems, inconsistencies and incompleteness in the requirements.
- As the specification is expressed in a formal language, it can be automatically analyzed to discover inconsistencies and incompleteness.
- If you use a formal method such as the B method, you can transform the formal specification into a ‘correct’ program.
- Program testing costs may be reduced if the program is formally verified against its specification.

Acceptance of formal methods

- Formal methods have had limited impact on practical software development:
  - Problem owners cannot understand a formal specification and so cannot assess if it is an accurate representation of their requirements.
  - It is easy to assess the costs of developing a formal specification but harder to assess the benefits. Managers may therefore be unwilling to invest in formal methods.
  - Software engineers are unfamiliar with this approach and are therefore reluctant to propose the use of FM.
  - Formal methods are still hard to scale up to large systems.
  - Formal specification is not really compatible with agile development methods.

Information Modeling

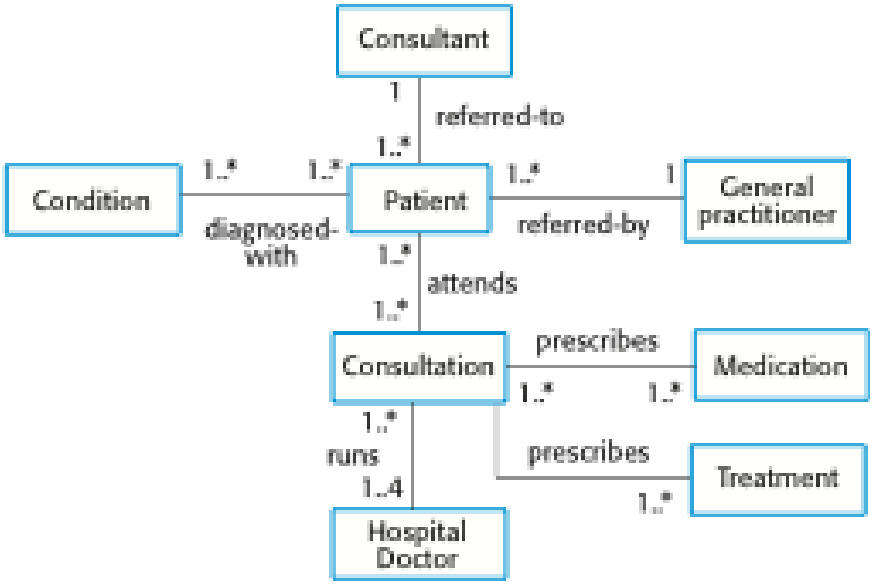
Lecture5

Class diagrams

- Class diagrams are used when developing an object-oriented system model to show the classes in a system and the associations between these classes.
- An object class can be thought of as a general definition of one kind of system object.
- An association is a link between classes that indicates that there is some relationship between these classes.
- When you are developing models during the early stages of the software engineering process, objects represent something in the real world, such as a patient, a prescription, doctor, etc.

UML classes and association

Classes and associations in the MHC-PMS



The Consultation Class

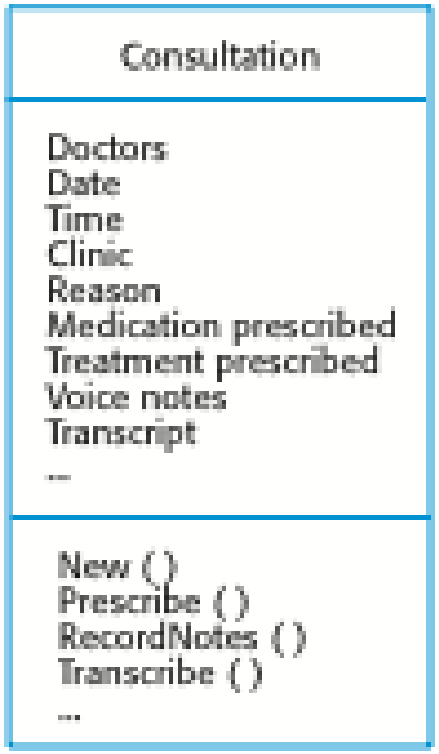


Figure 15.1 The simplest possible class diagram.

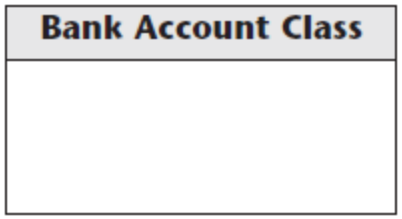
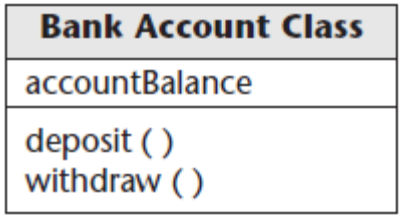


Figure 15.2 The class diagram of Figure 15.1 with an attribute and two operations added.



Behavioral Modeling

Lecture6

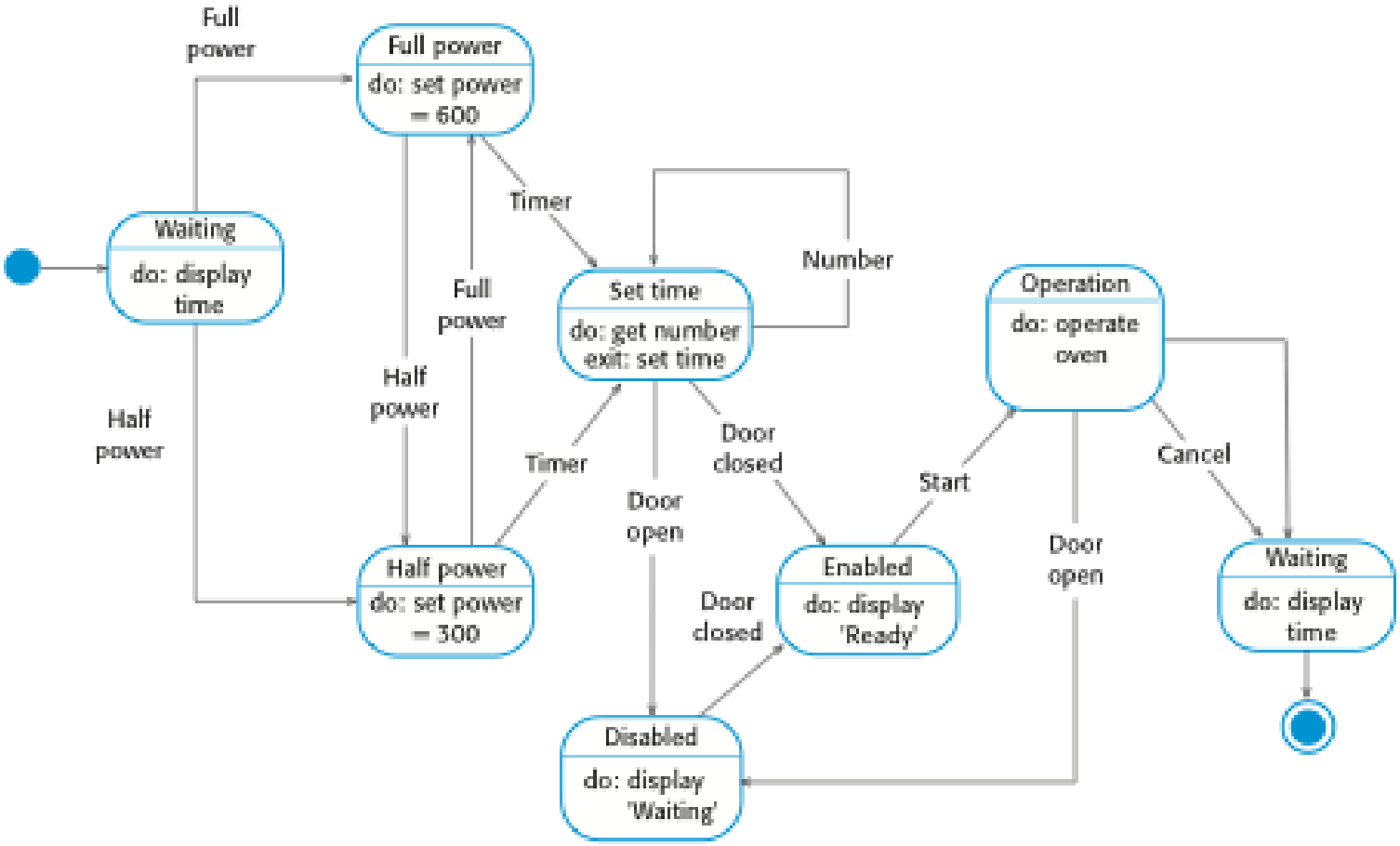
Behavioral models

- Behavioral models are models of the dynamic behavior of a system as it is executing. They show what happens or what is supposed to happen when a system responds to a stimulus from its environment.
- You can think of these stimuli as being of two types:
  - Data Some data arrives that has to be processed by the system.
  - Events Some event happens that triggers system processing. Events may have associated data, although this is not always the case.

State diagrams

- State diagrams show how the system reacts to internal and external events.

State diagrams



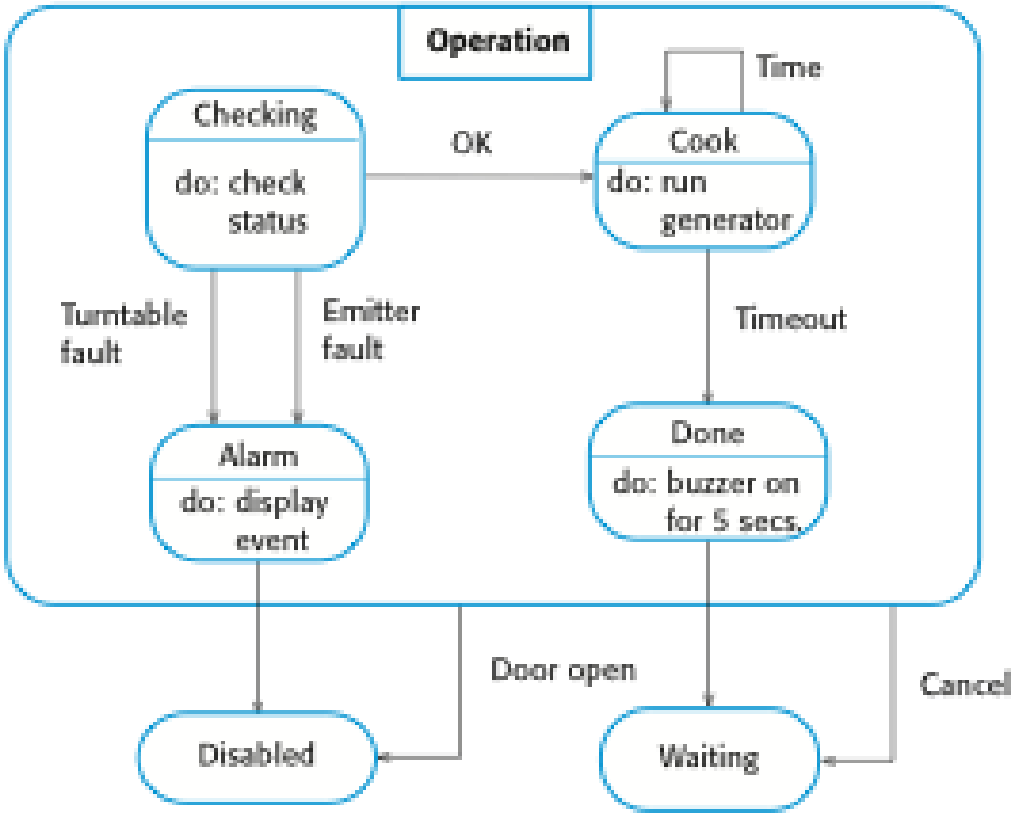
States and stimuli for the microwave oven (a)

- **State Description**
- Waiting The oven is waiting for input. The display shows the current time.
- Half power The oven power is set to 300 watts. The display shows ‘Half power’.
- Full power The oven power is set to 600 watts. The display shows ‘Full power’.
- Set time The cooking time is set to the user’s input value. The display shows the cooking time selected and is updated as the time is set.
- Disabled Oven operation is disabled for safety. Interior oven light is on. Display shows ‘Not ready’.
- Enabled Oven operation is enabled. Interior oven light is off. Display shows ‘Ready to cook’.
- Operation Oven in operation. Interior oven light is on. Display shows the timer countdown. On completion of cooking, the buzzer is sounded for five seconds. Oven light is on. Display shows ‘Cooking complete’ while buzzer is sounding.

States and stimuli for the microwave oven (b)

- **Stimulus Description**
- Half power The user has pressed the half-power button.
- Full power The user has pressed the full-power button.
- Timer The user has pressed one of the timer buttons.
- Number The user has pressed a numeric key.
- Door open The oven door switch is not closed.
- Door closed The oven door switch is closed.
- Start The user has pressed the Start button.
- Cancel The user has pressed the Cancel button.

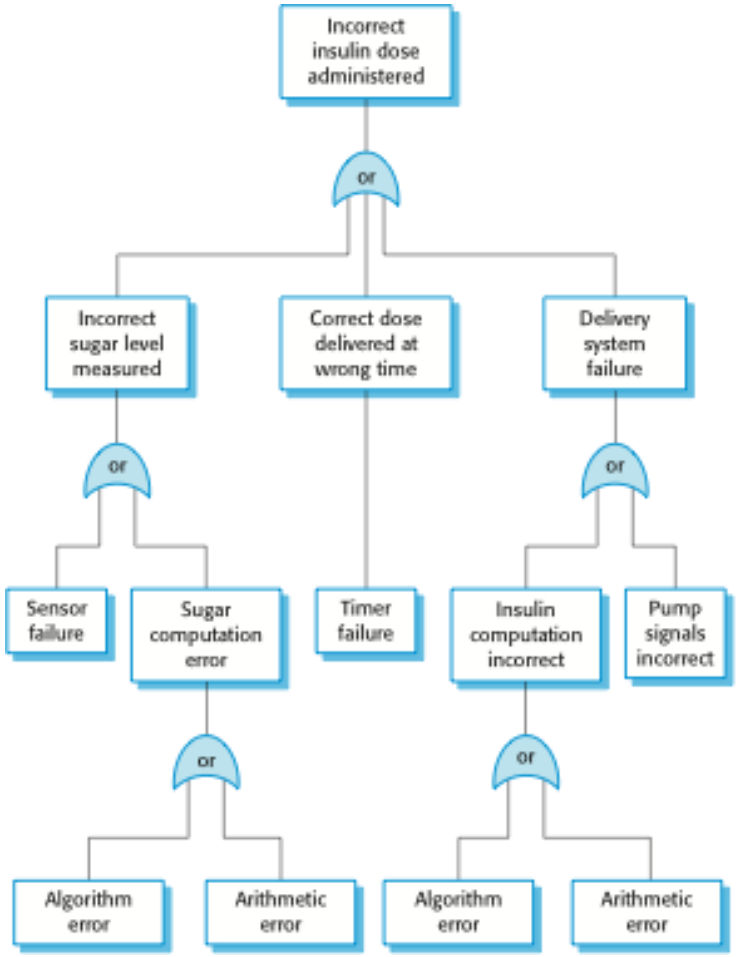
Microwave oven operation



Fault-tree analysis

- A deductive top-down technique.
- Put the risk or hazard at the root of the tree and identify the system states that could lead to that hazard.
- Where appropriate, link these with ‘and’ or ‘or’ conditions.
- A goal should be to minimise the number of single causes of system failure.

An example of a software fault tree



Fault tree analysis

- Three possible conditions that can lead to delivery of incorrect dose of insulin
  - Incorrect measurement of blood sugar level
  - Failure of delivery system
  - Dose delivered at wrong time
- By analysis of the fault tree, root causes of these hazards related to software are:
  - Algorithm error
  - Arithmetic error

Architectural Modeling

Lecture7

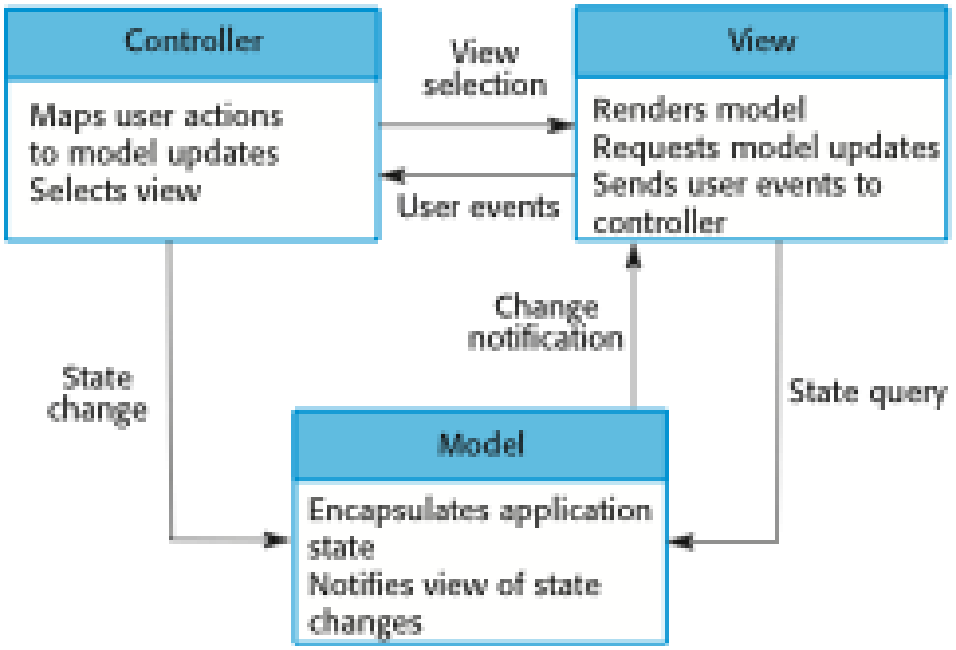
Architectural patterns

- Patterns are a means of representing, sharing and reusing knowledge.
- An architectural pattern is a stylized description of good design practice, which has been tried and tested in different environments.
- Patterns should include information about when they are and when the are not useful.
- Patterns may be represented using tabular and graphical descriptions.

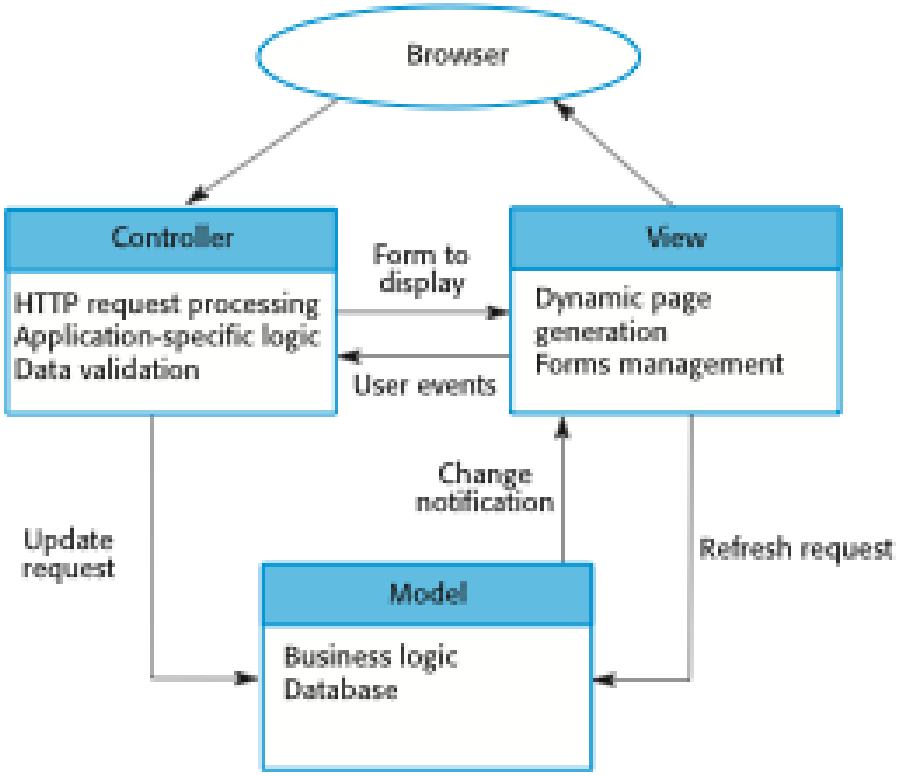
The Model-View-Controller (MVC) pattern

- **Name MVC (Model-View-Controller)**
- **Description** Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines and manages how the data is presented to the user. The Controller component manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model. See Figure 6.3.
- **Example** Figure 6.4 shows the architecture of a web-based application system organized using the MVC pattern.
- **When used** Used when there are multiple ways to view and interact with data. Also used when the future requirements for interaction and presentation of data are unknown.
- **Advantages** Allows the data to change independently of its representation and vice versa. Supports presentation of the same data in different ways with changes made in one representation shown in all of them.
- **Disadvantages** Can involve additional code and code complexity when the data model and interactions are simple.

The organization of the Model-View-Controller



Web application architecture using the MVC pattern



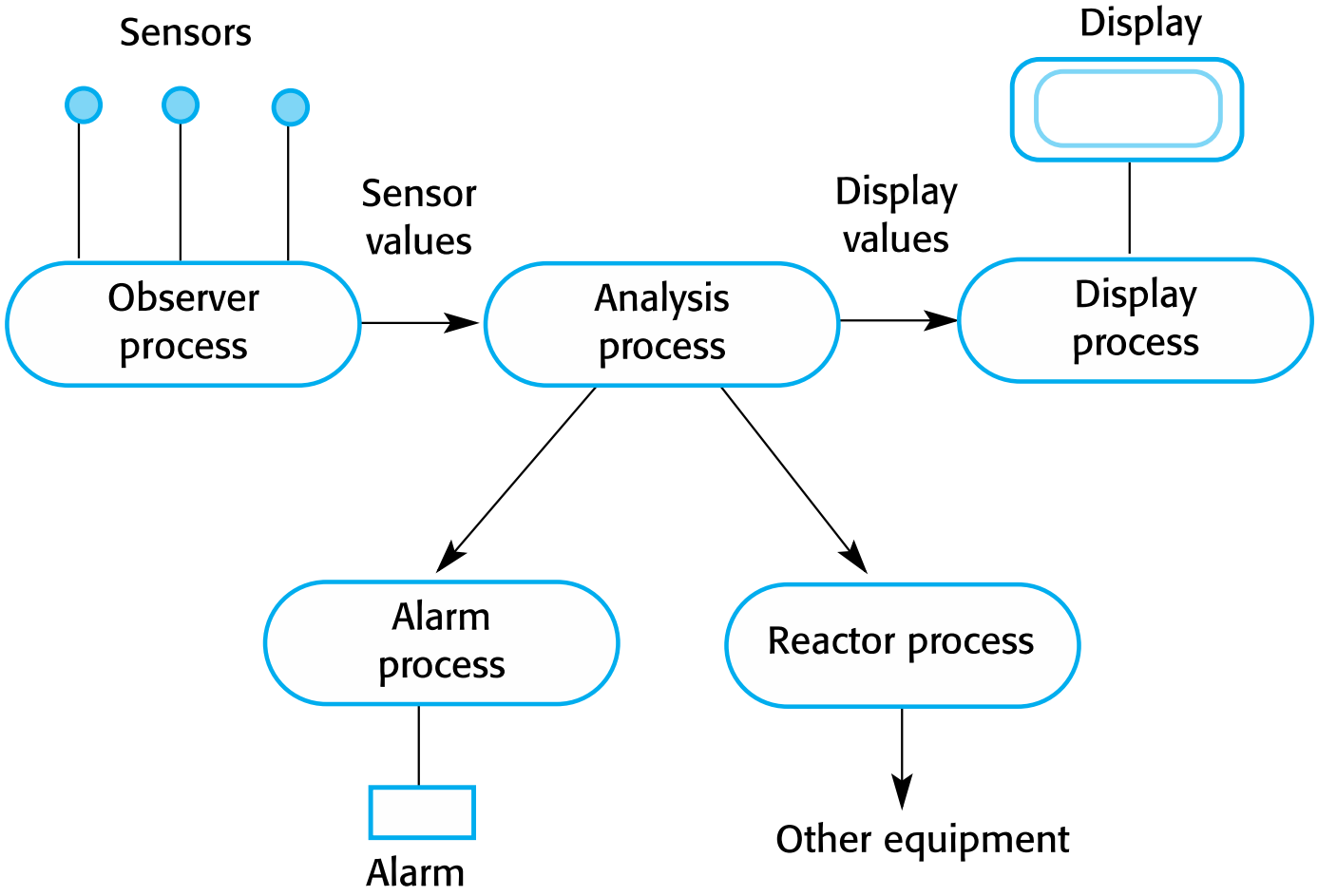
Architectural patterns for embedded systems

- Characteristic system architectures for embedded systems
  - *Observe and React* This pattern is used when a set of sensors are routinely monitored and displayed.
  - *Environmental Control* This pattern is used when a system includes sensors, which provide information about the environment and actuators that can change the environment
  - *Process Pipeline* This pattern is used when data has to be transformed from one representation to another before it can be processed.

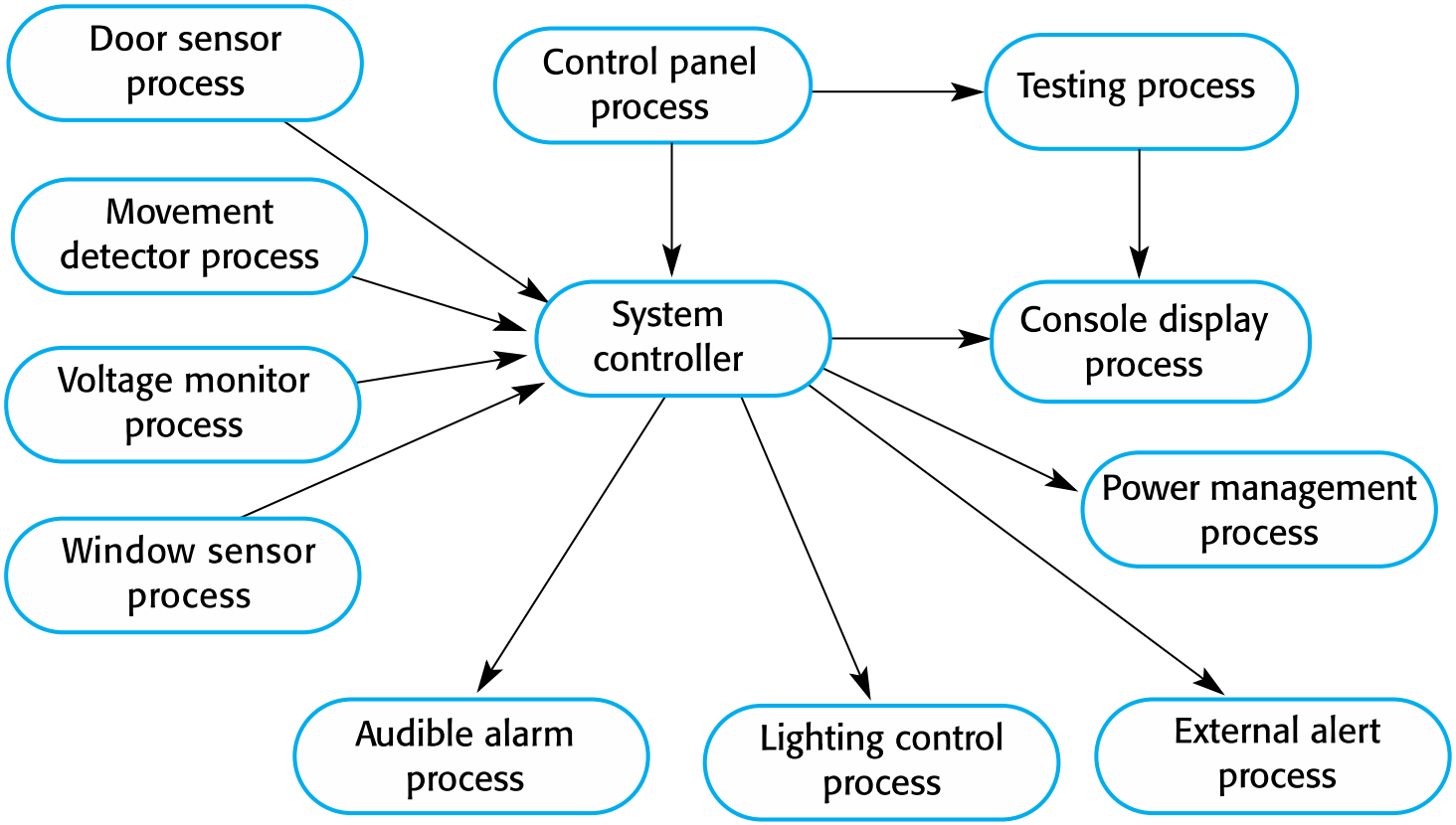
The Observe and React pattern

Name	Observe and React
Description	The input values of a set of sensors of the same types are collected and analyzed. These values are displayed in some way. If the sensor values indicate that some exceptional condition has arisen, then actions are initiated to draw the operator's attention to that value and, in certain cases, to take actions in response to the exceptional value.
Stimuli	Values from sensors attached to the system.
Responses	Outputs to display, alarm triggers, signals to reacting systems.
Processes	Observer, Analysis, Display, Alarm, Reactor.
Used in	Monitoring systems, alarm systems.

Observe and React process structure



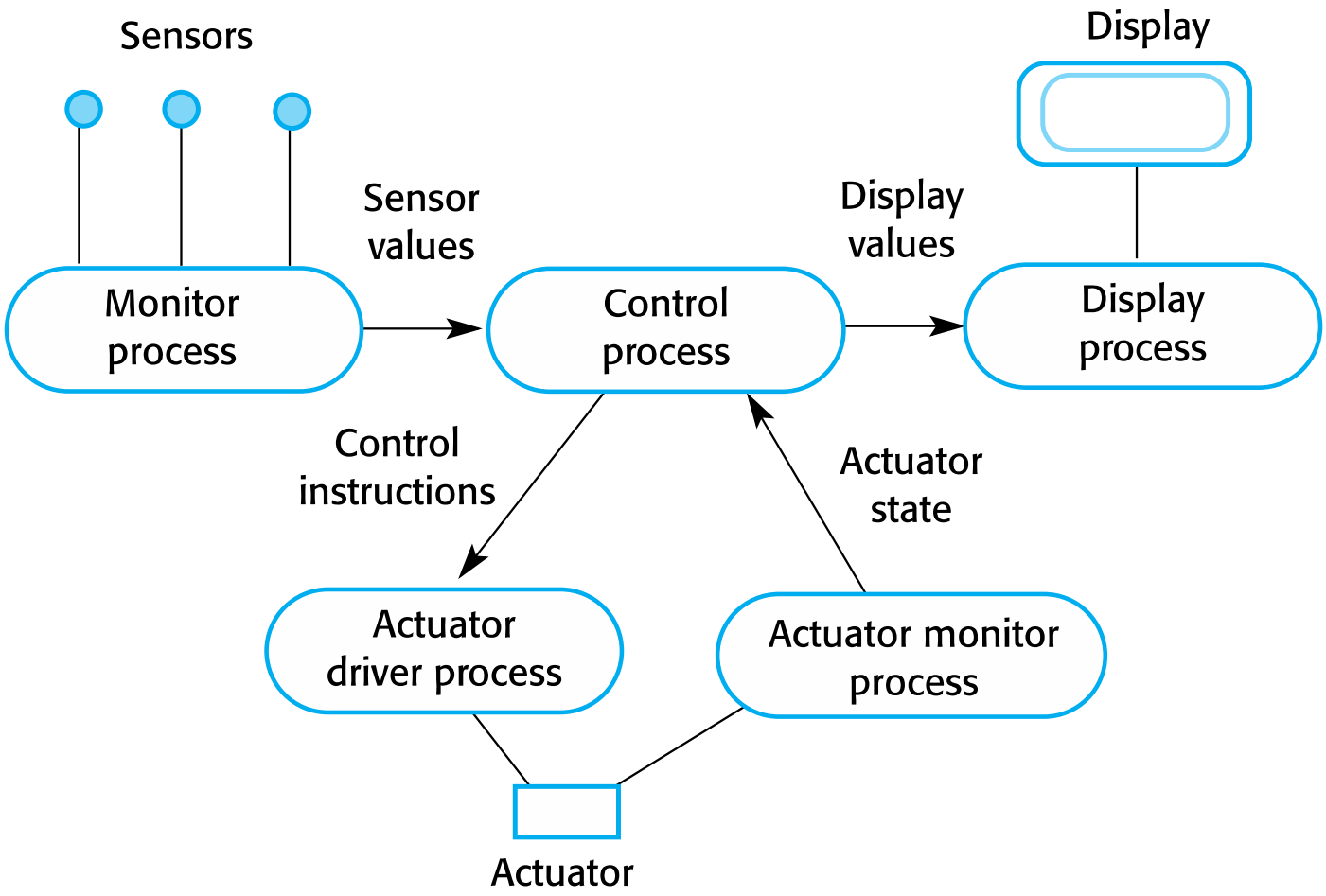
Process structure for a burglar alarm system



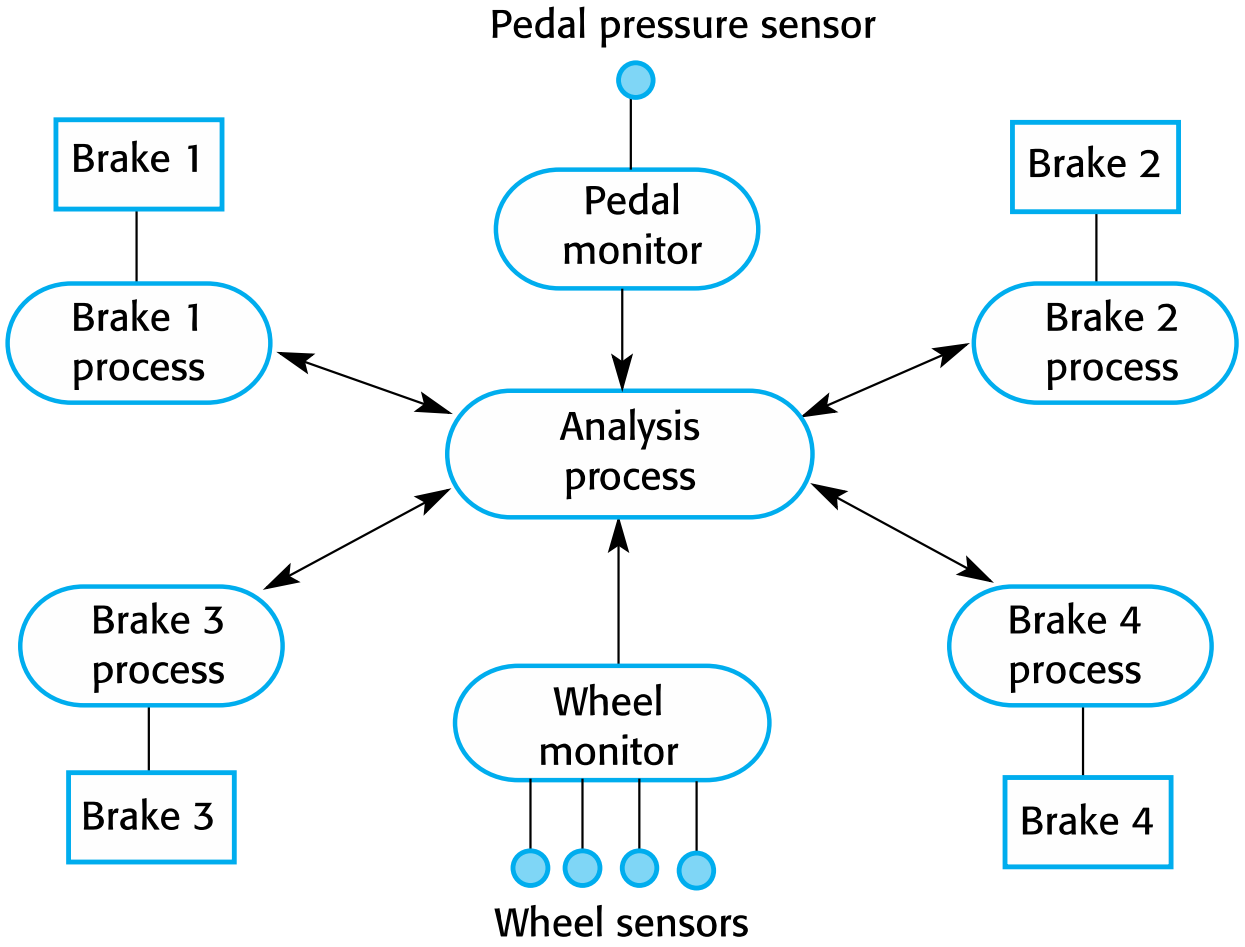
The Environmental Control pattern

Name	Environmental Control
Description	The system analyzes information from a set of sensors that collect data from the system's environment. Further information may also be collected on the state of the actuators that are connected to the system. Based on the data from the sensors and actuators, control signals are sent to the actuators that then cause changes to the system's environment. Information about the sensor values and the state of the actuators may be displayed.
Stimuli	Values from sensors attached to the system and the state of the system actuators.
Responses	Control signals to actuators, display information.
Processes	Monitor, Control, Display, Actuator Driver, Actuator monitor.
Used in	Control systems.

Environmental Control process structure



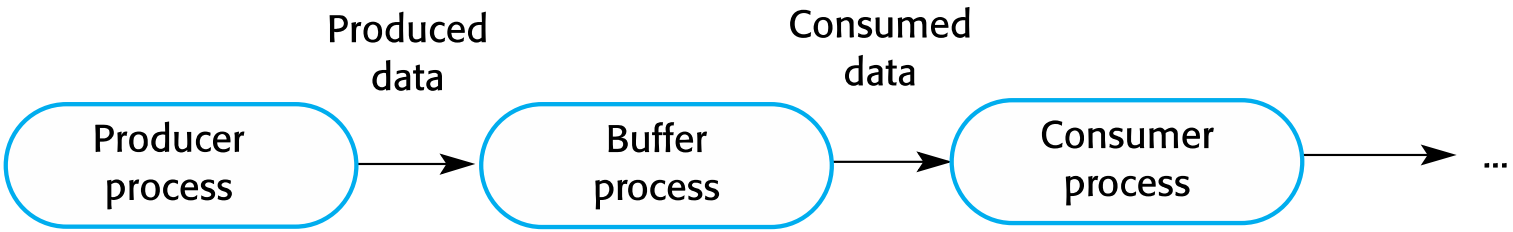
Control system architecture for an anti-skid braking system



The Process Pipeline pattern

Name	Process Pipeline
Description	A pipeline of processes is set up with data moving in sequence from one end of the pipeline to another. The processes are often linked by synchronized buffers to allow the producer and consumer processes to run at different speeds. The culmination of a pipeline may be display or data storage or the pipeline may terminate in an actuator.
Stimuli	Input values from the environment or some other process
Responses	Output values to the environment or a shared buffer
Processes	Producer, Buffer, Consumer
Used in	Data acquisition systems, multimedia systems

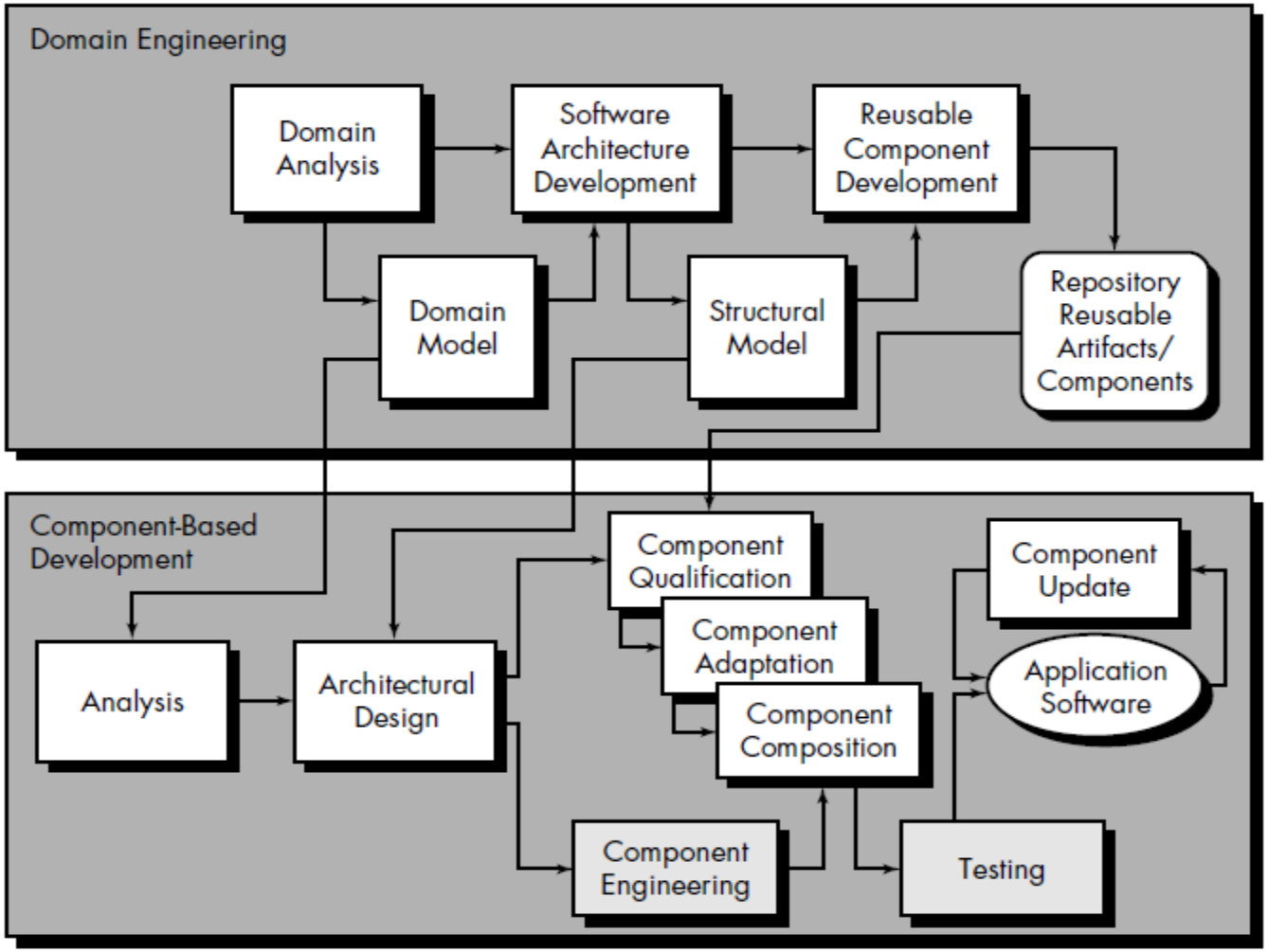
Process Pipeline process structure



Domain Modeling

Lecture8

domain engineering



domain engineering

- *Domain engineering performs the work required to* establish a set of software components that can be reused by the software engineer.
- The intent of domain engineering is to dentify, construct, catalog, and disseminate a set of software components that have applicability to existing and future software in a particular application domain.

domain engineering

- The overall goal is to establish mechanisms that enable software engineers to share these components—to reuse them—during work on new and existing systems.
- Domain engineering includes three major activities—analysis, construction, and dissemination.

The Domain Analysis Process

- 1. Define the domain to be investigated.
- **2. Categorize the items extracted from the domain.**
- **3. Collect a representative sample of applications in the domain.**
- **4. Analyze each application in the sample.**
- **5. Develop an analysis model for the objects.**

Characterization Functions

- 1: not relevant to whether reuse is appropriate
- 2: relevant only under unusual circumstances
- 3: relevant—the component can be modified so that it can be used, despite differences
- 4: clearly relevant, and if the new software does not have this characteristic, reuse will be inefficient but may still be possible
- 5: clearly relevant, and if the new software does not have this characteristic, reuse will be ineffective and reuse without the characteristic is not recommended

Structural Modeling and Structure Points

- **1. A structure point is an abstraction that should have a limited number of**
- instances. Restating this in object-oriented jargon, the size of the class hierarchy should be small. In addition, the abstraction should recur throughout applications in the domain. Otherwise, the cost to verify, document, and disseminate the structure point cannot be justified.
- **2. The rules that govern the use of the structure point should be easily understood.**
- In addition, the interface to the structure point should be relatively
- simple.
- **3. The structure point should implement information hiding by isolating all**
- complexity contained within the structure point itself. This reduces the perceived complexity of the overall system.

Enterprise Modeling

Lecture9

Business process

- A *business process* is “a set of logically related tasks performed to achieve a defined business outcome”
- Within the business process, people, equipment, material resources, and business procedures are combined to produce a specified result.
- Examples of business processes include designing a new product, purchasing services and supplies, hiring a new employee, and paying suppliers. Each demands a set of tasks and each draws on diverse resources within the business.

Business process

- Every business process has a defined customer—a person or group that receives

the outcome (e.g., an idea, a report, a design, a product). In addition, business

- processes cross organizational boundaries. They require that different organizational

groups participate in the “logically related tasks” that define the process.

Static workflows in the Rational Unified Process

Workflow	Description
Business modelling	The business processes are modelled using business use cases.
Requirements	Actors who interact with the system are identified and use cases are developed to model the system requirements.
Analysis and design	A design model is created and documented using architectural models, component models, object models and sequence models.
Implementation	The components in the system are implemented and structured into implementation sub-systems. Automatic code generation from design models helps accelerate this process.

Static workflows in the Rational Unified Process

Workflow	Description
Testing	Testing is an iterative process that is carried out in conjunction with implementation. System testing follows the completion of the implementation.
Deployment	A product release is created, distributed to users and installed in their workplace.
Configuration and change management	This supporting workflow managed changes to the system .
Project management	This supporting workflow manages the system development.
Environment	This workflow is concerned with making appropriate software tools available to the software development team.

RUP good practice

- Develop software iteratively
  - Plan increments based on customer priorities and deliver highest priority increments first.
- Manage requirements
  - Explicitly document customer requirements and keep track of changes to these requirements.
- Use component-based architectures
  - Organize the system architecture as a set of reusable components.

RUP good practice

- Visually model software
  - Use graphical UML models to present static and dynamic views of the software.
- Verify software quality
  - Ensure that the software meet's organizational quality standards.
- Control changes to software
  - Manage software changes using a change management system and configuration management tools.

Modeling Embedded Software

Lecture10

Embedded software

- Computers are used to control a wide range of systems from simple domestic machines, through games controllers, to entire manufacturing plants.
- Their software must react to events generated by the hardware and, often, issue control signals in response to these events.
- The software in these systems is embedded in system hardware, often in read-only memory, and usually responds, in real time, to events from the system's environment.

Responsiveness

- Responsiveness in real-time is the critical difference between embedded systems and other software systems, such as information systems, web-based systems or personal software systems.
- For non-real-time systems, correctness can be defined by specifying how system inputs map to corresponding outputs that should be produced by the system.
- In a real-time system, the correctness depends both on the response to an input and the time taken to generate that response. If the system takes too long to respond, then the required response may be ineffective.

Definition

- A real-time system is a software system where the correct functioning of the system depends on the results produced by the system and the time at which these results are produced.
- A soft real-time system is a system whose operation is degraded if results are not produced according to the specified timing requirements.
- A hard real-time system is a system whose operation is incorrect if results are not produced according to the timing specification.

Embedded system characteristics

- Embedded systems generally run continuously and do not terminate.
- Interactions with the system's environment are uncontrollable and unpredictable.
- There may be physical limitations (e.g. power) that affect the design of a system.
- Direct hardware interaction may be necessary.
- Issues of safety and reliability may dominate the system design.

Embedded system design

- The design process for embedded systems is a systems engineering process that has to consider, in detail, the design and performance of the system hardware.
- Part of the design process may involve deciding which system capabilities are to be implemented in software and which in hardware.
- Low-level decisions on hardware, support software and system timing must be considered early in the process.
- These may mean that additional software functionality, such as battery and power management, has to be included in the system.

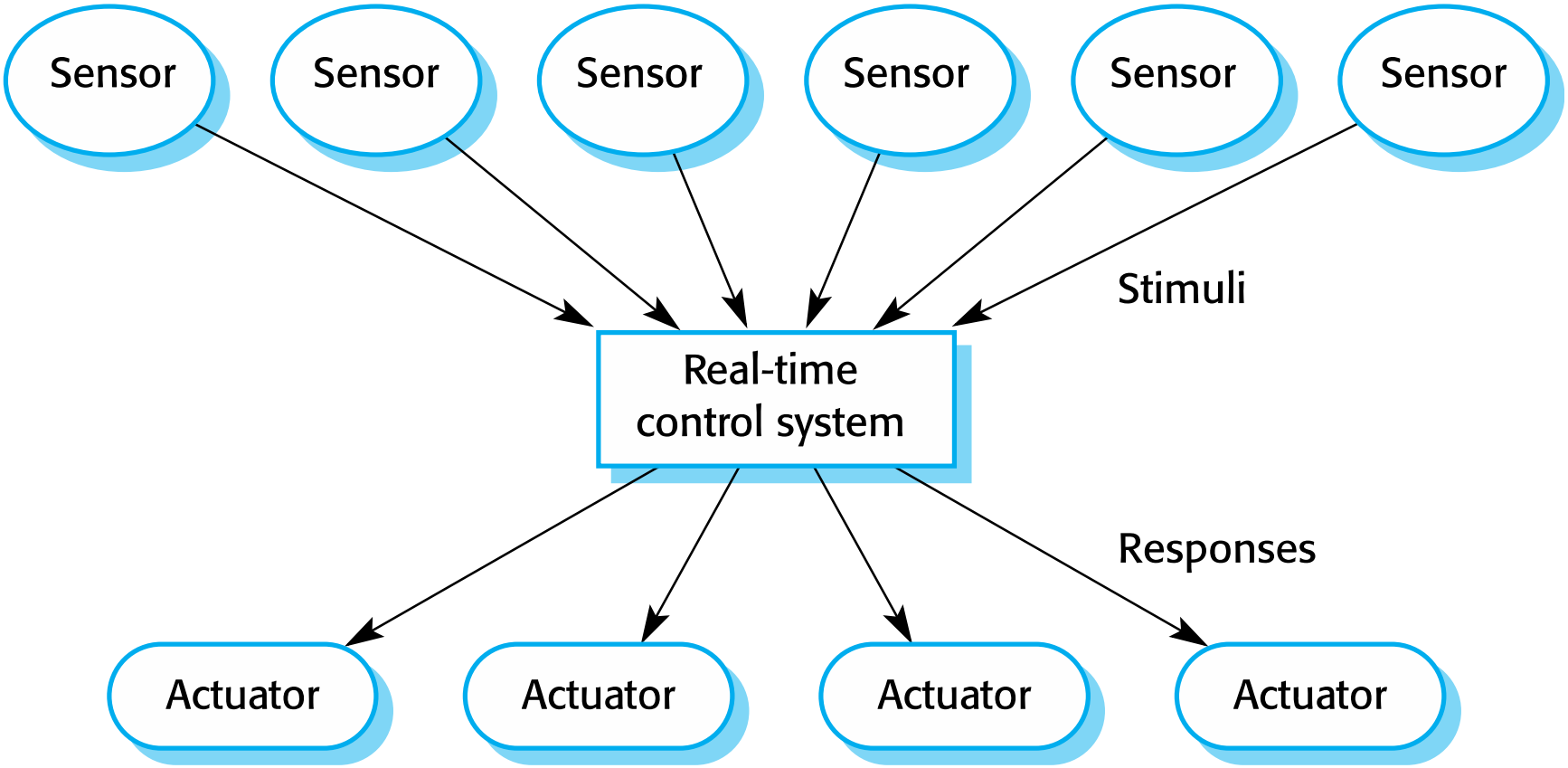
Reactive systems

- Given a stimulus, the system must produce a reaction or response within a specified time.
- Periodic stimuli. Stimuli which occur at predictable time intervals
  - For example, a temperature sensor may be polled 10 times per second.
- Aperiodic stimuli. Stimuli which occur at unpredictable times
  - For example, a system power failure may trigger an interrupt which must be processed by the system.

Stimuli and responses for a burglar alarm system

Stimulus	Response
Single sensor positive	Initiate alarm; turn on lights around site of positive sensor.
Two or more sensors positive	Initiate alarm; turn on lights around sites of positive sensors; call police with location of suspected break-in.
Voltage drop of between 10% and 20%	Switch to battery backup; run power supply test.
Voltage drop of more than 20%	Switch to battery backup; initiate alarm; call police; run power supply test.
Power supply failure	Call service technician.
Sensor failure	Call service technician.
Console panic button positive	Initiate alarm; turn on lights around console; call police.
Clear alarms	Switch off all active alarms; switch off all lights that have been switched on.

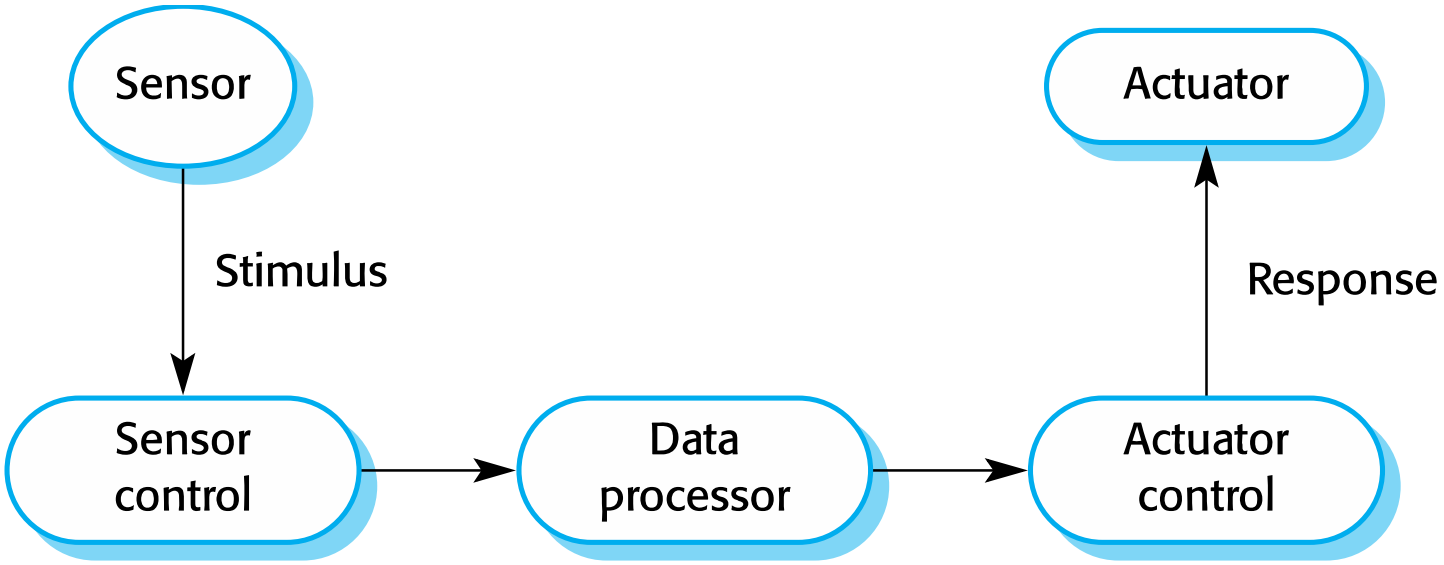
A general model of an embedded real-time system



Architectural considerations

- Because of the need to respond to timing demands made by different stimuli/responses, the system architecture must allow for fast switching between stimulus handlers.
- Timing demands of different stimuli are different so a simple sequential loop is not usually adequate.
- Real-time systems are therefore usually designed as cooperating processes with a real-time executive controlling these processes.

Sensor and actuator processes



System elements

- Sensor control processes
  - Collect information from sensors. May buffer information collected in response to a sensor stimulus.
- Data processor
  - Carries out processing of collected information and computes the system response.
- Actuator control processes
  - Generates control signals for the actuators.

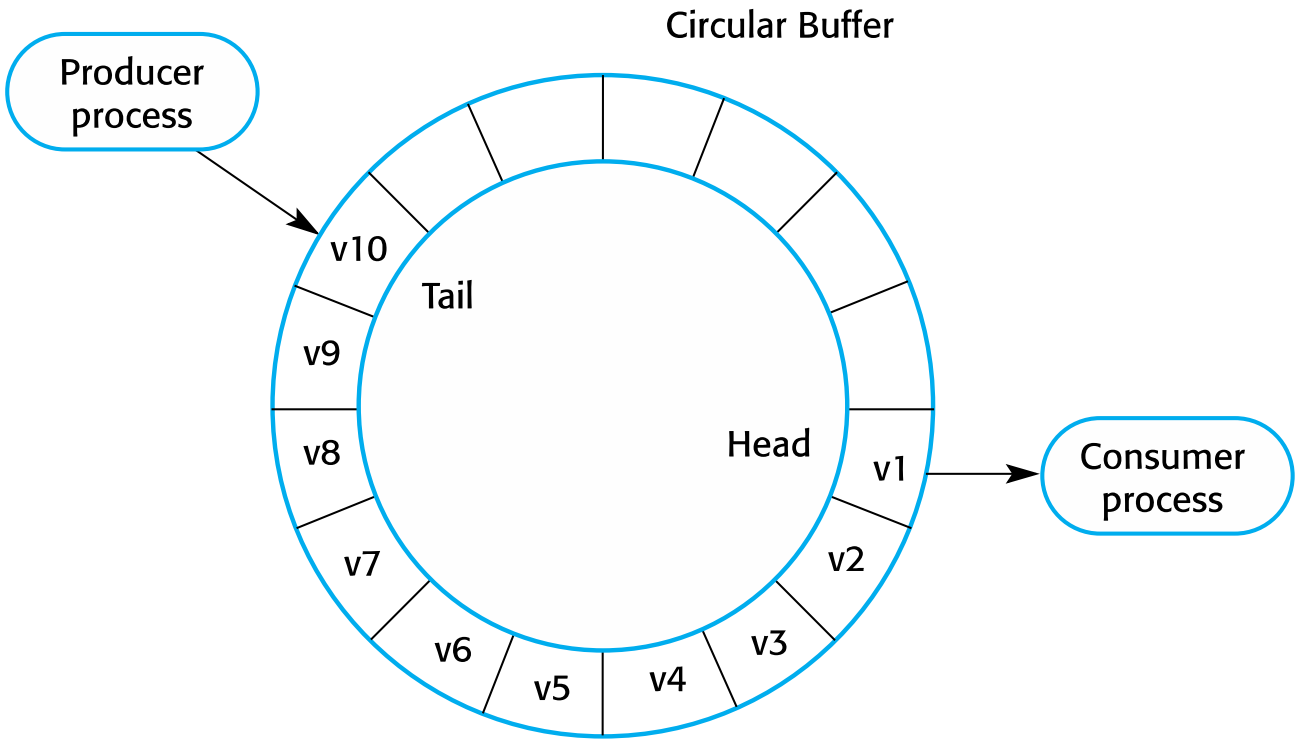
Design process activities

- Platform selection
- Stimuli/response identification
- Timing analysis
- Process design
- Algorithm design
- Data design
- Process scheduling

Process coordination

- Processes in a real-time system have to be coordinated and share information.
- Process coordination mechanisms ensure mutual exclusion to shared resources.
- When one process is modifying a shared resource, other processes should not be able to change that resource.
- When designing the information exchange between processes, you have to take into account the fact that these processes may be running at different speeds.

Producer/consumer processes sharing a circular buffer



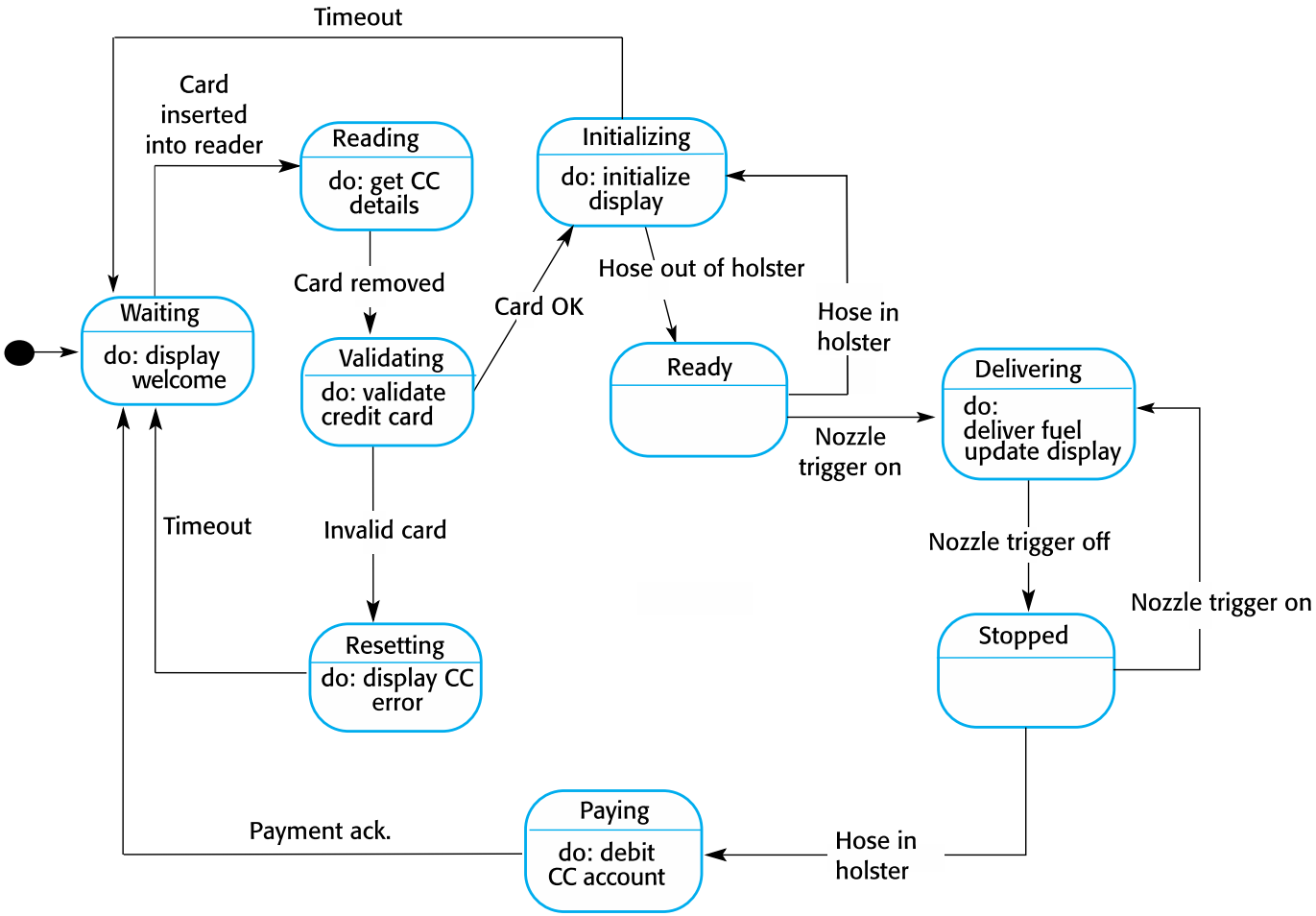
Mutual exclusion

- Producer processes collect data and add it to the buffer. Consumer processes take data from the buffer and make elements available.
- Producer and consumer processes must be mutually excluded from accessing the same element.
- The buffer must stop producer processes adding information to a full buffer and consumer processes trying to take information from an empty buffer.

Real-time system modelling

- The effect of a stimulus in a real-time system may trigger a transition from one state to another.
- State models are therefore often used to describe embedded real-time systems.
- UML state diagrams may be used to show the states and state transitions in a real-time system.

State machine model of a petrol (gas) pump



Real-time programming

- Programming languages for real-time systems development have to include facilities to access system hardware, and it should be possible to predict the timing of particular operations in these languages.
- Systems-level languages, such as C, which allow efficient code to be generated are widely used in preference to languages such as Java.
- There is a performance overhead in object-oriented systems because extra code is required to mediate access to attributes and handle calls to operations. The loss of performance may make it impossible to meet real-time deadlines.

Analyzing Form

Lecture11

Requirements completeness and consistency

- In principle, requirements should be both complete and consistent.
- Complete
  - They should include descriptions of all facilities required.
- Consistent
  - There should be no conflicts or contradictions in the descriptions of the system facilities.
- In practice, it is impossible to produce a complete and consistent requirements document.

Requirements checking

- Validity. Does the system provide the functions which best support the customer's needs?
- Consistency. Are there any requirements conflicts?
- Completeness. Are all functions required by the customer included?
- Realism. Can the requirements be implemented given available budget and technology
- Verifiability. Can the requirements be checked?

Definitions

- *Consistency.* The use of uniform design and documentation techniques throughout the software development project.
- *Completeness.* The degree to which full implementation of required function has been achieved.

Analyzing Correctness

Lecture 12

Validation of critical systems

- The verification and validation costs for critical systems involves additional validation processes and analysis than for non-critical systems:
  - The costs and consequences of failure are high so it is cheaper to find and remove faults than to pay for system failure;
  - You may have to make a formal case to customers or to a regulator that the system meets its dependability requirements. This dependability case may require specific V & V activities to be carried out.

Validation costs

- Because of the additional activities involved, the validation costs for critical systems are usually significantly higher than for non-critical systems.
- Normally, V & V costs take up more than 50% of the total system development costs.
- The outcome of the validation process is a tangible body of evidence that demonstrates the level of dependability of a software system.

Static analysis

- Static analysis techniques are system verification techniques that don't involve executing a program.
- The work on a source representation of the software – either a model or the program code itself.
- Inspections and reviews are a form of static analysis
- Techniques covered here:
  - Formal verification
  - Model checking
  - Automated program analysis

Verification and formal methods

- Formal methods can be used when a mathematical specification of the system is produced.
- They are the ultimate static verification technique that may be used at different stages in the development process:
  - A formal specification may be developed and mathematically analyzed for consistency. This helps discover specification errors and omissions.
  - Formal arguments that a program conforms to its mathematical specification may be developed. This is effective in discovering programming and design errors.

Arguments for formal methods

- Producing a mathematical specification requires a detailed analysis of the requirements and this is likely to uncover errors.
- Concurrent systems can be analysed to discover race conditions that might lead to deadlock. Testing for such problems is very difficult.
- They can detect implementation errors before testing when the program is analyzed alongside the specification.

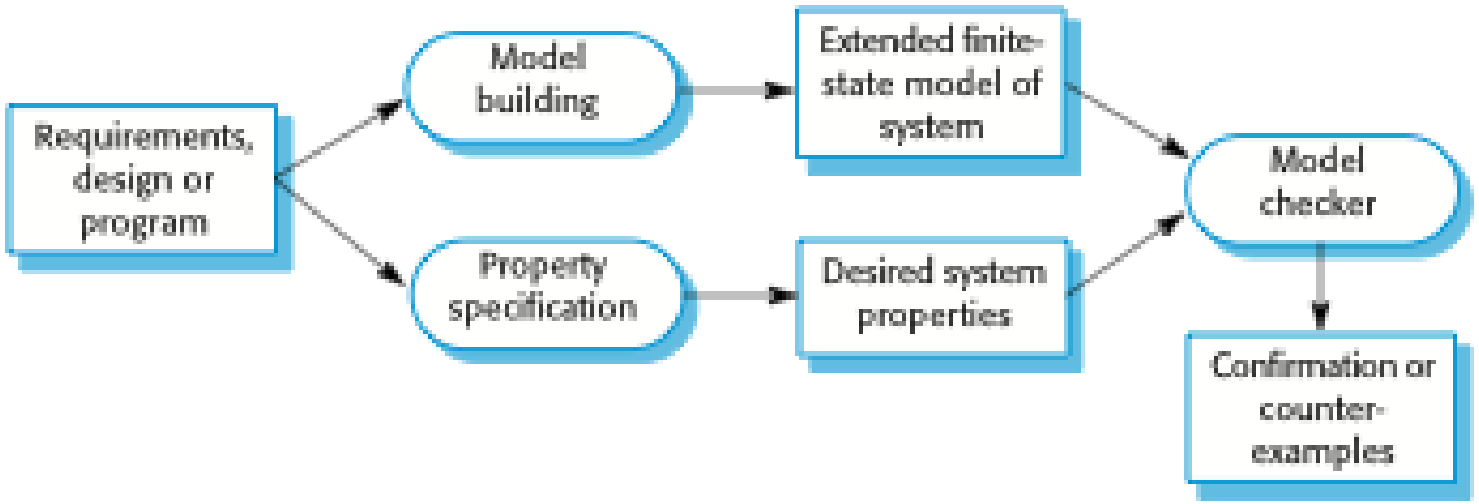
Arguments against formal methods

- Require specialised notations that cannot be understood by domain experts.
- Very expensive to develop a specification and even more expensive to show that a program meets that specification.
- Proofs may contain errors.
- It may be possible to reach the same level of confidence in a program more cheaply using other V & V techniques.

Model checking

- Involves creating an extended finite state model of a system and, using a specialized system (a model checker), checking that model for errors.
- The model checker explores all possible paths through the model and checks that a user-specified property is valid for each path.
- Model checking is particularly valuable for verifying concurrent systems, which are hard to test.
- Although model checking is computationally very expensive, it is now practical to use it in the verification of small to medium sized critical systems.

Model checking



Automated static analysis

- Static analysers are software tools for source text processing.
- They parse the program text and try to discover potentially erroneous conditions and bring these to the attention of the V & V team.
- They are very effective as an aid to inspections - they are a supplement to but not a replacement for inspections.

Automated static analysis checks

Fault class	Static analysis check
Data faults	Variables used before initialization Variables declared but never used Variables assigned twice but never used between assignments Possible array bound violations Undeclared variables
Control faults	Unreachable code Unconditional branches into loops
Input/output faults	Variables output twice with no intervening assignment
Interface faults	Parameter-type mismatches Parameter number mismatches Non-usage of the results of functions Uncalled functions and procedures
Storage management faults	Unassigned pointers Pointer arithmetic Memory leaks

Levels of static analysis

- Characteristic error checking
  - The static analyzer can check for patterns in the code that are characteristic of errors made by programmers using a particular language.
- User-defined error checking
  - Users of a programming language define error patterns, thus extending the types of error that can be detected. This allows specific rules that apply to a program to be checked.
- Assertion checking
  - Developers include formal assertions in their program and relationships that must hold. The static analyzer symbolically executes the code and highlights potential problems.

Use of static analysis

- Particularly valuable when a language such as C is used which has weak typing and hence many errors are undetected by the compiler.
- Particularly valuable for security checking – the static analyzer can discover areas of vulnerability such as buffer overflows or unchecked inputs.
- Static analysis is now routinely used in the development of many safety and security critical systems.

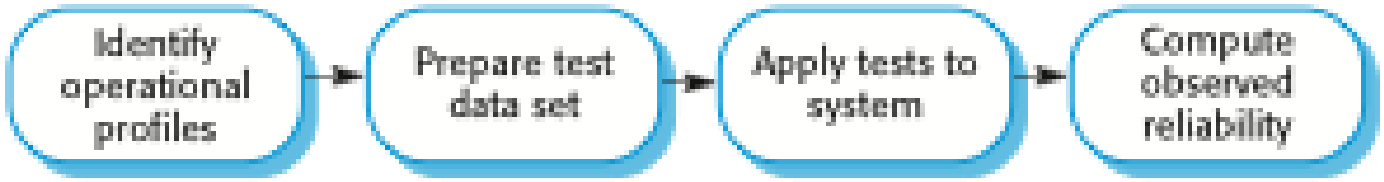
Reliability testing

- Reliability validation involves exercising the program to assess whether or not it has reached the required level of reliability.
- This cannot normally be included as part of a normal defect testing process because data for defect testing is (usually) atypical of actual usage data.
- Reliability measurement therefore requires a specially designed data set that replicates the pattern of inputs to be processed by the system.

Reliability validation activities

- Establish the operational profile for the system.
- Construct test data reflecting the operational profile.
- Test the system and observe the number of failures and the times of these failures.
- Compute the reliability after a statistically significant number of failures have been observed.

Reliability measurement



Statistical testing

- Testing software for reliability rather than fault detection.
- Measuring the number of errors allows the reliability of the software to be predicted. Note that, for statistical reasons, more errors than are allowed for in the reliability specification must be induced.
- An acceptable level of reliability should be specified and the software tested and amended until that level of reliability is reached.

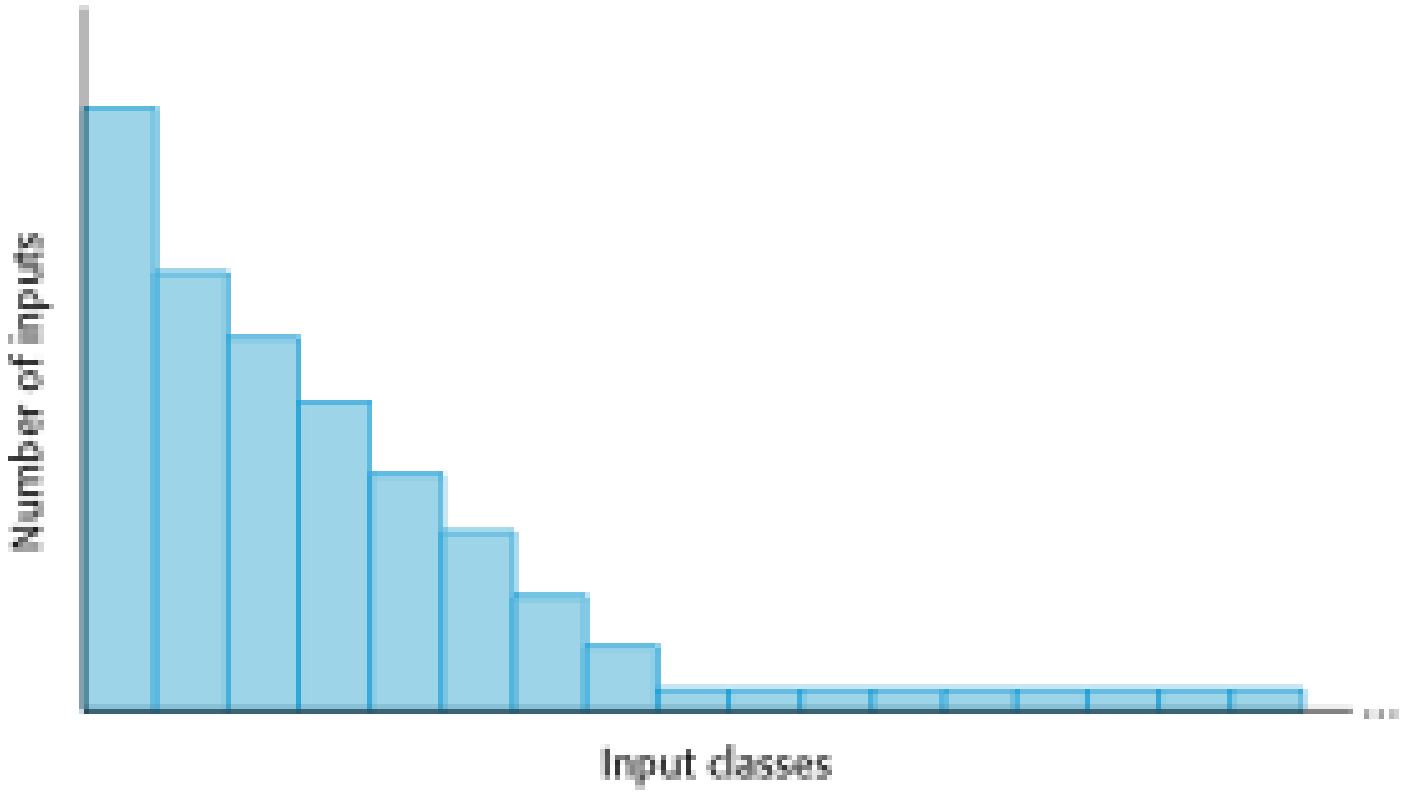
Reliability measurement problems

- Operational profile uncertainty
  - The operational profile may not be an accurate reflection of the real use of the system.
- High costs of test data generation
  - Costs can be very high if the test data for the system cannot be generated automatically.
- Statistical uncertainty
  - You need a statistically significant number of failures to compute the reliability but highly reliable systems will rarely fail.
- Recognizing failure
  - It is not always obvious when a failure has occurred as there may be conflicting interpretations of a specification.

Operational profiles

- An operational profile is a set of test data whose frequency matches the actual frequency of these inputs from ‘normal’ usage of the system. A close match with actual usage is necessary otherwise the measured reliability will not be reflected in the actual usage of the system.
- It can be generated from real data collected from an existing system or (more often) depends on assumptions made about the pattern of usage of a system.

An operational profile



Operational profile generation

- Should be generated automatically whenever possible.
- Automatic profile generation is difficult for interactive systems.
- May be straightforward for ‘normal’ inputs but it is difficult to predict ‘unlikely’ inputs and to create test data for them.
- Pattern of usage of new systems is unknown.
- Operational profiles are not static but change as users learn about a new system and change the way that they use it.

Analyzing Dependability

Lecture13

Topics covered

- Risk-driven specification
- Safety specification
- Security specification
- Software reliability specification

Dependability requirements

- Functional requirements to define error checking and recovery facilities and protection against system failures.
- Non-functional requirements defining the required reliability and availability of the system.
- Excluding requirements that define states and conditions that must not arise.

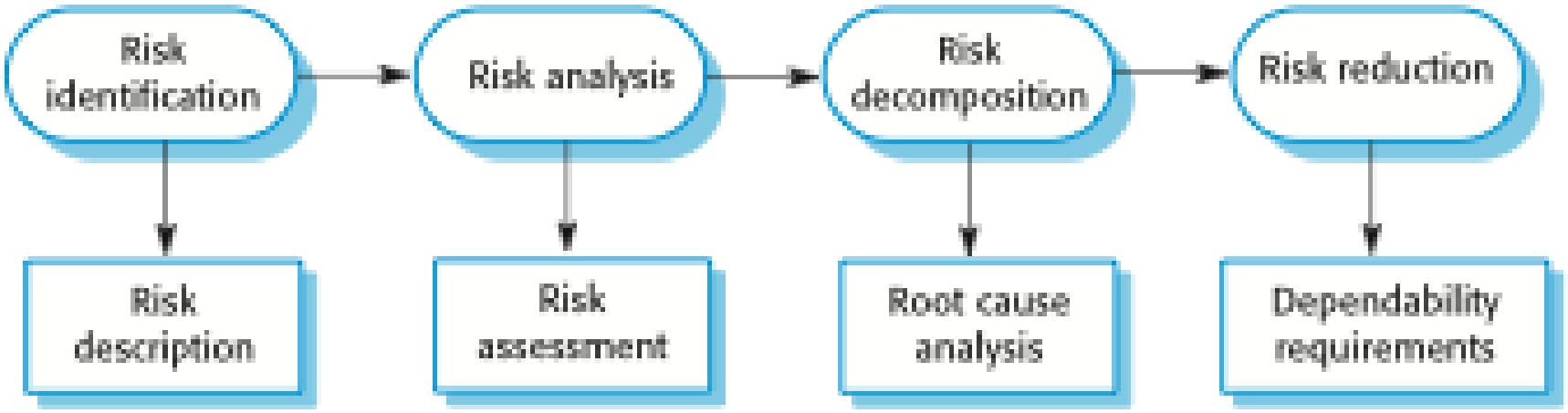
Risk-driven specification

- Critical systems specification should be risk-driven.
- This approach has been widely used in safety and security-critical systems.
- The aim of the specification process should be to understand the risks (safety, security, etc.) faced by the system and to define requirements that reduce these risks.

Stages of risk-based analysis

- Risk identification
  - Identify potential risks that may arise.
- Risk analysis and classification
  - Assess the seriousness of each risk.
- Risk decomposition
  - Decompose risks to discover their potential root causes.
- Risk reduction assessment
  - Define how each risk must be taken into eliminated or reduced when the system is designed.

Risk-driven specification



Phased risk analysis

- Preliminary risk analysis
  - Identifies risks from the systems environment. Aim is to develop an initial set of system security and dependability requirements.
- Life cycle risk analysis
  - Identifies risks that emerge during design and development e.g. risks that are associated with the technologies used for system construction. Requirements are extended to protect against these risks.
- Operational risk analysis
  - Risks associated with the system user interface and operator errors. Further protection requirements may be added to cope with these.

Safety specification

- Goal is to identify protection requirements that ensure that system failures do not cause injury or death or environmental damage.
- Risk identification = Hazard identification
- Risk analysis = Hazard assessment
- Risk decomposition = Hazard analysis
- Risk reduction = safety requirements specification

Hazard identification

- Identify the hazards that may threaten the system.
- Hazard identification may be based on different types of hazard:
  - Physical hazards
  - Electrical hazards
  - Biological hazards
  - Service failure hazards
  - Etc.

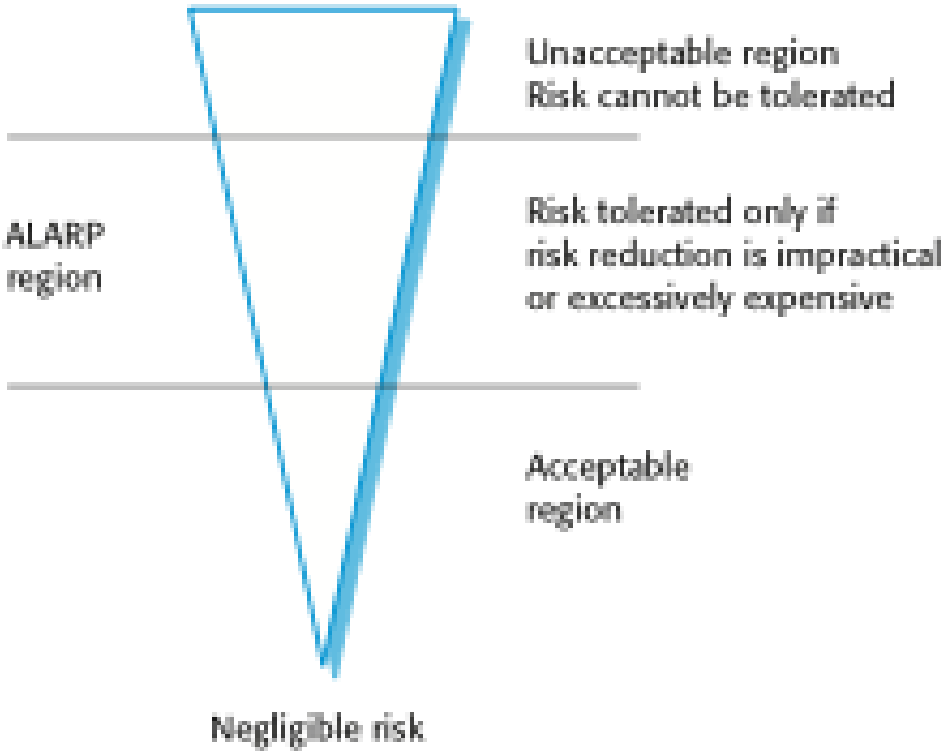
Insulin pump risks

- Insulin overdose (service failure).
- Insulin underdose (service failure).
- Power failure due to exhausted battery (electrical).
- Electrical interference with other medical equipment (electrical).
- Poor sensor and actuator contact (physical).
- Parts of machine break off in body (physical).
- Infection caused by introduction of machine (biological).
- Allergic reaction to materials or insulin (biological).

Hazard assessment

- The process is concerned with understanding the likelihood that a risk will arise and the potential consequences if an accident or incident should occur.
- Risks may be categorised as:
  - Intolerable. Must never arise or result in an accident
  - As low as reasonably practical(ALARP). Must minimise the possibility of risk given cost and schedule constraints
  - Acceptable. The consequences of the risk are acceptable and no extra costs should be incurred to reduce hazard probability

The risk triangle



Social acceptability of risk

- The acceptability of a risk is determined by human, social and political considerations.
- In most societies, the boundaries between the regions are pushed upwards with time i.e. society is less willing to accept risk
  - For example, the costs of cleaning up pollution may be less than the costs of preventing it but this may not be socially acceptable.
- Risk assessment is subjective
  - Risks are identified as probable, unlikely, etc. This depends on who is making the assessment.

Hazard assessment

- Estimate the risk probability and the risk severity.
- It is not normally possible to do this precisely so relative values are used such as ‘unlikely’, ‘rare’, ‘very high’, etc.
- The aim must be to exclude risks that are likely to arise or that have high severity.

Risk classification for the insulin pump

Identified hazard	Hazard probability	Accident severity	Estimated risk	Acceptability
1.Insulin overdose computation	Medium	High	High	Intolerable
2. Insulin underdose computation	Medium	Low	Low	Acceptable
3. Failure of hardware monitoring system	Medium	Medium	Low	ALARP
4. Power failure	High	Low	Low	Acceptable
5. Machine incorrectly fitted	High	High	High	Intolerable
6. Machine breaks in patient	Low	High	Medium	ALARP
7. Machine causes infection	Medium	Medium	Medium	ALARP
8. Electrical interference	Low	High	Medium	ALARP
9. Allergic reaction	Low	Low	Low	Acceptable

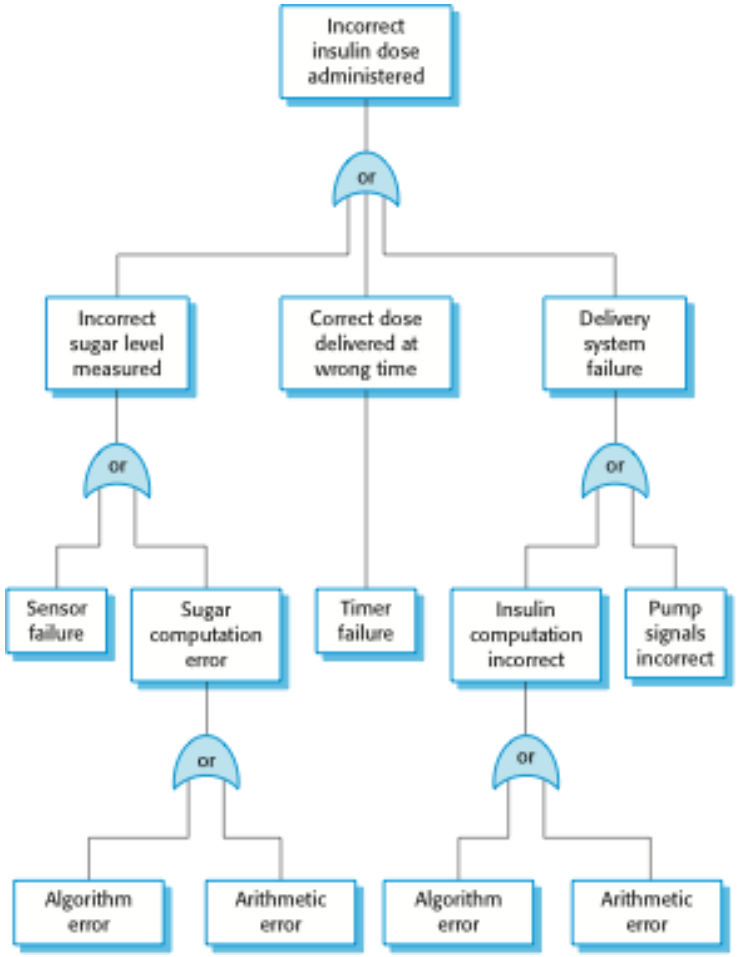
Hazard analysis

- Concerned with discovering the root causes of risks in a particular system.
- Techniques have been mostly derived from safety-critical systems and can be
  - Inductive, bottom-up techniques. Start with a proposed system failure and assess the hazards that could arise from that failure;
  - Deductive, top-down techniques. Start with a hazard and deduce what the causes of this could be.

Fault-tree analysis

- A deductive top-down technique.
- Put the risk or hazard at the root of the tree and identify the system states that could lead to that hazard.
- Where appropriate, link these with ‘and’ or ‘or’ conditions.
- A goal should be to minimise the number of single causes of system failure.

An example of a software fault tree



Fault tree analysis

- Three possible conditions that can lead to delivery of incorrect dose of insulin
  - Incorrect measurement of blood sugar level
  - Failure of delivery system
  - Dose delivered at wrong time
- By analysis of the fault tree, root causes of these hazards related to software are:
  - Algorithm error
  - Arithmetic error

Risk reduction

- The aim of this process is to identify dependability requirements that specify how the risks should be managed and ensure that accidents/incidents do not arise.
- Risk reduction strategies
  - Risk avoidance;
  - Risk detection and removal;
  - Damage limitation.

Strategy use

- Normally, in critical systems, a mix of risk reduction strategies are used.
- In a chemical plant control system, the system will include sensors to detect and correct excess pressure in the reactor.
- However, it will also include an independent protection system that opens a relief valve if dangerously high pressure is detected.

Insulin pump - software risks

- Arithmetic error
  - A computation causes the value of a variable to overflow or underflow;
  - Maybe include an exception handler for each type of arithmetic error.
- Algorithmic error
  - Compare dose to be delivered with previous dose or safe maximum doses. Reduce dose if too high.

Examples of safety requirements

SR1: The system shall not deliver a single dose of insulin that is greater than a specified maximum dose for a system user.

SR2: The system shall not deliver a daily cumulative dose of insulin that is greater than a specified maximum daily dose for a system user.

SR3: The system shall include a hardware diagnostic facility that shall be executed at least four times per hour.

SR4: The system shall include an exception handler for all of the exceptions that are identified in Table 3.

SR5: The audible alarm shall be sounded when any hardware or software anomaly is discovered and a diagnostic message, as defined in Table 4, shall be displayed.

SR6: In the event of an alarm, insulin delivery shall be suspended until the user has reset the system and cleared the alarm.

Topics covered

- Risk-driven specification
- Safety specification
- Security specification
- Software reliability specification

Dependability requirements

- Functional requirements to define error checking and recovery facilities and protection against system failures.
- Non-functional requirements defining the required reliability and availability of the system.
- Excluding requirements that define states and conditions that must not arise.

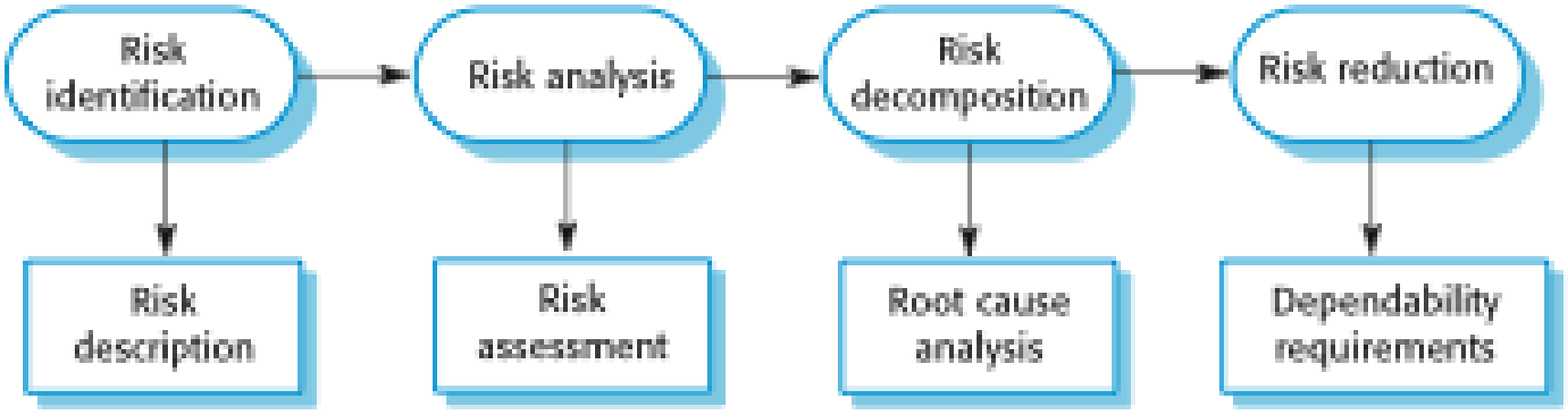
Risk-driven specification

- Critical systems specification should be risk-driven.
- This approach has been widely used in safety and security-critical systems.
- The aim of the specification process should be to understand the risks (safety, security, etc.) faced by the system and to define requirements that reduce these risks.

Stages of risk-based analysis

- Risk identification
  - Identify potential risks that may arise.
- Risk analysis and classification
  - Assess the seriousness of each risk.
- Risk decomposition
  - Decompose risks to discover their potential root causes.
- Risk reduction assessment
  - Define how each risk must be taken into eliminated or reduced when the system is designed.

Risk-driven specification



Phased risk analysis

- Preliminary risk analysis
  - Identifies risks from the systems environment. Aim is to develop an initial set of system security and dependability requirements.
- Life cycle risk analysis
  - Identifies risks that emerge during design and development e.g. risks that are associated with the technologies used for system construction. Requirements are extended to protect against these risks.
- Operational risk analysis
  - Risks associated with the system user interface and operator errors. Further protection requirements may be added to cope with these.

Safety specification

- Goal is to identify protection requirements that ensure that system failures do not cause injury or death or environmental damage.
- Risk identification = Hazard identification
- Risk analysis = Hazard assessment
- Risk decomposition = Hazard analysis
- Risk reduction = safety requirements specification

Hazard identification

- Identify the hazards that may threaten the system.
- Hazard identification may be based on different types of hazard:
  - Physical hazards
  - Electrical hazards
  - Biological hazards
  - Service failure hazards
  - Etc.

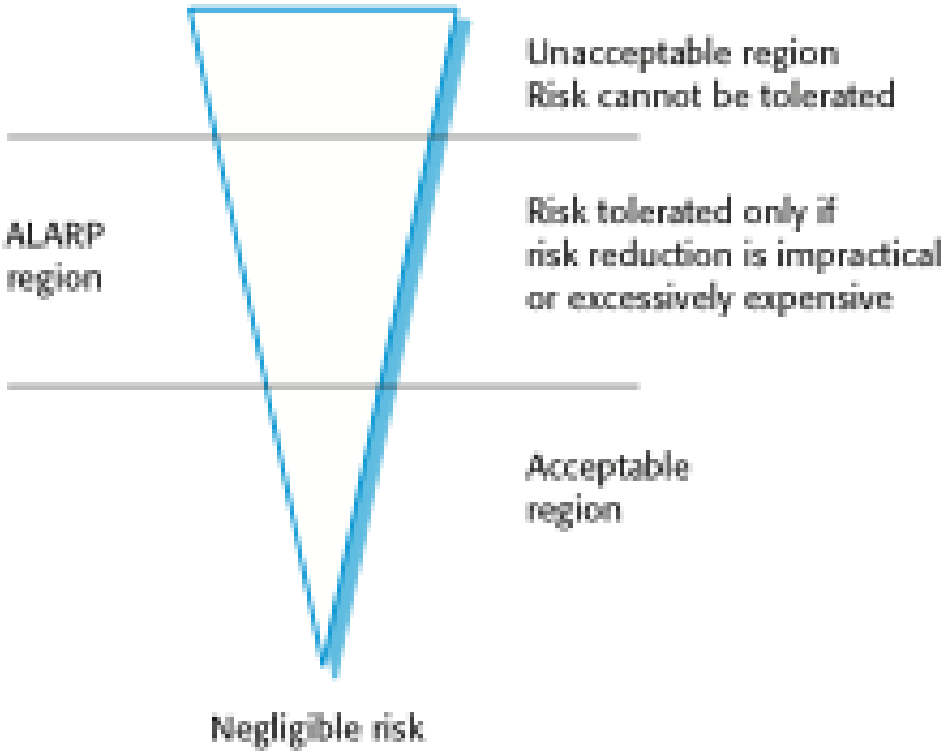
Insulin pump risks

- Insulin overdose (service failure).
- Insulin underdose (service failure).
- Power failure due to exhausted battery (electrical).
- Electrical interference with other medical equipment (electrical).
- Poor sensor and actuator contact (physical).
- Parts of machine break off in body (physical).
- Infection caused by introduction of machine (biological).
- Allergic reaction to materials or insulin (biological).

Hazard assessment

- The process is concerned with understanding the likelihood that a risk will arise and the potential consequences if an accident or incident should occur.
- Risks may be categorised as:
  - Intolerable. Must never arise or result in an accident
  - As low as reasonably practical(ALARP). Must minimise the possibility of risk given cost and schedule constraints
  - Acceptable. The consequences of the risk are acceptable and no extra costs should be incurred to reduce hazard probability

The risk triangle



Social acceptability of risk

- The acceptability of a risk is determined by human, social and political considerations.
- In most societies, the boundaries between the regions are pushed upwards with time i.e. society is less willing to accept risk
  - For example, the costs of cleaning up pollution may be less than the costs of preventing it but this may not be socially acceptable.
- Risk assessment is subjective
  - Risks are identified as probable, unlikely, etc. This depends on who is making the assessment.

Hazard assessment

- Estimate the risk probability and the risk severity.
- It is not normally possible to do this precisely so relative values are used such as ‘unlikely’, ‘rare’, ‘very high’, etc.
- The aim must be to exclude risks that are likely to arise or that have high severity.

Risk classification for the insulin pump

Identified hazard	Hazard probability	Accident severity	Estimated risk	Acceptability
1.Insulin overdose computation	Medium	High	High	Intolerable
2. Insulin underdose computation	Medium	Low	Low	Acceptable
3. Failure of hardware monitoring system	Medium	Medium	Low	ALARP
4. Power failure	High	Low	Low	Acceptable
5. Machine incorrectly fitted	High	High	High	Intolerable
6. Machine breaks in patient	Low	High	Medium	ALARP
7. Machine causes infection	Medium	Medium	Medium	ALARP
8. Electrical interference	Low	High	Medium	ALARP
9. Allergic reaction	Low	Low	Low	Acceptable

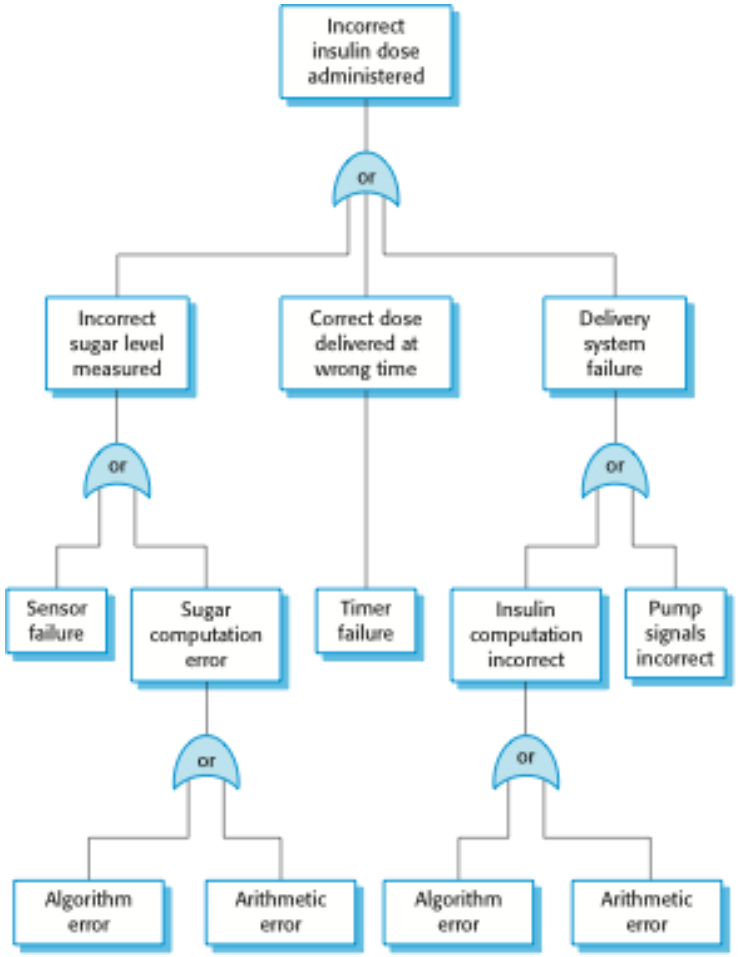
Hazard analysis

- Concerned with discovering the root causes of risks in a particular system.
- Techniques have been mostly derived from safety-critical systems and can be
  - Inductive, bottom-up techniques. Start with a proposed system failure and assess the hazards that could arise from that failure;
  - Deductive, top-down techniques. Start with a hazard and deduce what the causes of this could be.

Fault-tree analysis

- A deductive top-down technique.
- Put the risk or hazard at the root of the tree and identify the system states that could lead to that hazard.
- Where appropriate, link these with ‘and’ or ‘or’ conditions.
- A goal should be to minimise the number of single causes of system failure.

An example of a software fault tree



Fault tree analysis

- Three possible conditions that can lead to delivery of incorrect dose of insulin
  - Incorrect measurement of blood sugar level
  - Failure of delivery system
  - Dose delivered at wrong time
- By analysis of the fault tree, root causes of these hazards related to software are:
  - Algorithm error
  - Arithmetic error

Risk reduction

- The aim of this process is to identify dependability requirements that specify how the risks should be managed and ensure that accidents/incidents do not arise.
- Risk reduction strategies
  - Risk avoidance;
  - Risk detection and removal;
  - Damage limitation.

Strategy use

- Normally, in critical systems, a mix of risk reduction strategies are used.
- In a chemical plant control system, the system will include sensors to detect and correct excess pressure in the reactor.
- However, it will also include an independent protection system that opens a relief valve if dangerously high pressure is detected.

Insulin pump - software risks

- Arithmetic error
  - A computation causes the value of a variable to overflow or underflow;
  - Maybe include an exception handler for each type of arithmetic error.
- Algorithmic error
  - Compare dose to be delivered with previous dose or safe maximum doses. Reduce dose if too high.

Examples of safety requirements

SR1: The system shall not deliver a single dose of insulin that is greater than a specified maximum dose for a system user.

SR2: The system shall not deliver a daily cumulative dose of insulin that is greater than a specified maximum daily dose for a system user.

SR3: The system shall include a hardware diagnostic facility that shall be executed at least four times per hour.

SR4: The system shall include an exception handler for all of the exceptions that are identified in Table 3.

SR5: The audible alarm shall be sounded when any hardware or software anomaly is discovered and a diagnostic message, as defined in Table 4, shall be displayed.

SR6: In the event of an alarm, insulin delivery shall be suspended until the user has reset the system and cleared the alarm.

Formal Methods

Lecture14

Formal specification

- Formal specification is part of a more general collection of techniques that are known as ‘formal methods’.
- These are all based on mathematical representation and analysis of software.
- Formal methods include
  - Formal specification;
  - Specification analysis and proof;
  - Transformational development;
  - Program verification.

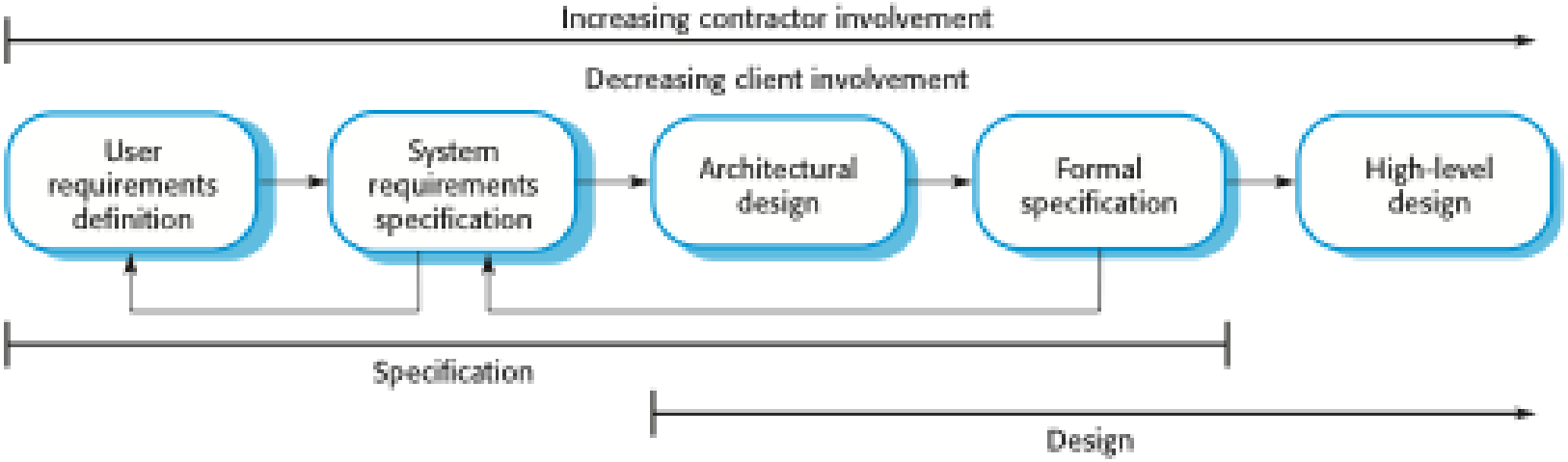
Use of formal methods

- The principal benefits of formal methods are in reducing the number of faults in systems.
- Consequently, their main area of applicability is in critical systems engineering. There have been several successful projects where formal methods have been used in this area.
- In this area, the use of formal methods is most likely to be cost-effective because high system failure costs must be avoided.

Specification in the software process

- Specification and design are inextricably intermingled.
- Architectural design is essential to structure a specification and the specification process.
- Formal specifications are expressed in a mathematical notation with precisely defined vocabulary, syntax and semantics.

Formal specification in a plan-based software process



Benefits of formal specification

- Developing a formal specification requires the system requirements to be analyzed in detail. This helps to detect problems, inconsistencies and incompleteness in the requirements.
- As the specification is expressed in a formal language, it can be automatically analyzed to discover inconsistencies and incompleteness.
- If you use a formal method such as the B method, you can transform the formal specification into a ‘correct’ program.
- Program testing costs may be reduced if the program is formally verified against its specification.

Acceptance of formal methods

- Formal methods have had limited impact on practical software development:
  - Problem owners cannot understand a formal specification and so cannot assess if it is an accurate representation of their requirements.
  - It is easy to assess the costs of developing a formal specification but harder to assess the benefits. Managers may therefore be unwilling to invest in formal methods.
  - Software engineers are unfamiliar with this approach and are therefore reluctant to propose the use of FM.
  - Formal methods are still hard to scale up to large systems.
  - Formal specification is not really compatible with agile development methods.

Verification and formal methods

- Formal methods can be used when a mathematical specification of the system is produced.
- They are the ultimate static verification technique that may be used at different stages in the development process:
  - A formal specification may be developed and mathematically analyzed for consistency. This helps discover specification errors and omissions.
  - Formal arguments that a program conforms to its mathematical specification may be developed. This is effective in discovering programming and design errors.

Arguments for formal methods

- Producing a mathematical specification requires a detailed analysis of the requirements and this is likely to uncover errors.
- Concurrent systems can be analysed to discover race conditions that might lead to deadlock. Testing for such problems is very difficult.
- They can detect implementation errors before testing when the program is analyzed alongside the specification.

Arguments against formal methods

- Require specialized notations that cannot be understood by domain experts.
- Very expensive to develop a specification and even more expensive to show that a program meets that specification.
- Proofs may contain errors.
- It may be possible to reach the same level of confidence in a program more cheaply using other V & V techniques.

Discussions

Lecture15