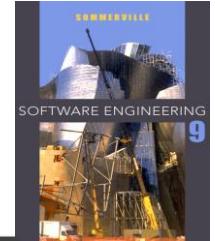


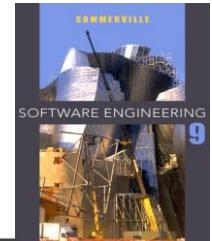
Chapter 1- Introduction

Lecture 1



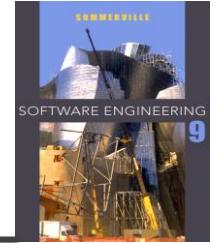
Topics covered

- ✧ Professional software development
 - What is meant by software engineering.
- ✧ Software engineering ethics
 - A brief introduction to ethical issues that affect software engineering.
- ✧ Case studies
 - An introduction to three examples that are used in later chapters in the book.



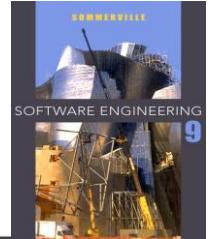
Software engineering

- ✧ 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 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 engineering is concerned with cost-effective software development.



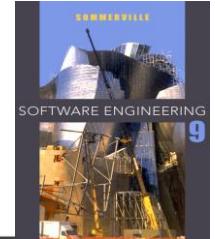
Software products

✧ 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

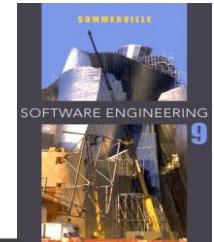
✧ Generic products

- 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

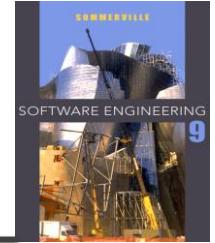
- The specification of what the software should do is owned by the customer for the software and they make decisions on software changes that are required.

Frequently asked questions about software engineering



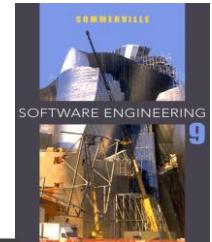
Question	Answer
What is software?	Computer programs and associated documentation. Software products may be developed for a particular customer or may be developed for a general market.
What are the attributes of good software?	Good software should deliver the required functionality and performance to 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.

Frequently asked questions about software engineering

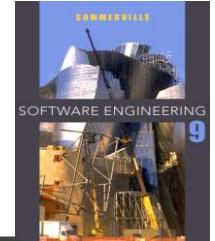


Question	Answer
What are the key challenges facing software engineering?	Coping with increasing diversity, demands for reduced delivery times and developing trustworthy software.
What are the costs of software engineering?	Roughly 60% of software costs are development costs, 40% are testing costs. For 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.

Essential attributes of good software

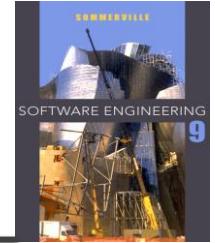


Product characteristic	Description
Maintainability	Software should be written in such a way so that it can evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable requirement of a changing business environment.
Dependability and security	Software 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.



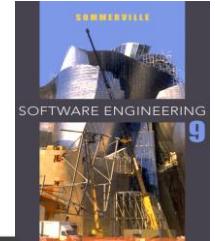
Software engineering

- ✧ Software engineering is an engineering discipline that is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use.
- ✧ Engineering discipline
 - Using appropriate theories and methods to solve problems bearing in mind organizational and financial constraints.
- ✧ All aspects of software production
 - Not just technical process of development. Also project management and the development of tools, methods etc. to support software production.



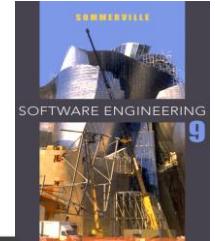
Importance of software engineering

- ✧ 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.



General issues that affect most software

✧ Heterogeneity

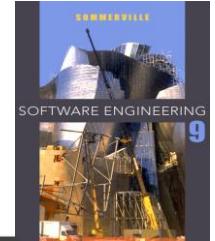
- Increasingly, systems are required to operate as distributed systems across networks that include different types of computer and mobile devices.

✧ Business and social change

- Business and society are changing incredibly quickly as emerging economies develop and new technologies become available. They need to be able to change their existing software and to rapidly develop new software.

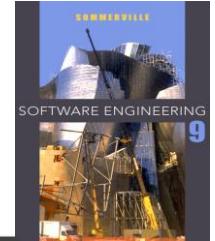
✧ Security and trust

- As software is intertwined with all aspects of our lives, it is essential that we can trust that software.



Software engineering diversity

- ✧ There are many different types of software system and there is no universal set of software techniques that is applicable to all of these.
- ✧ The software engineering methods and tools used depend on the type of application being developed, the requirements of the customer and the background of the development team.



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.

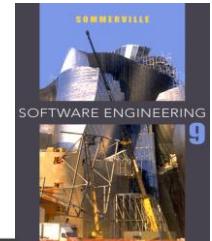
✧ Interactive transaction-based applications

ACID

- 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 → IoT

- 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

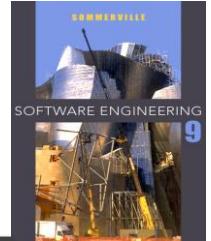
- These are business systems that are designed to process data in large batches. They process large numbers of individual inputs to create corresponding outputs.

✧ 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.



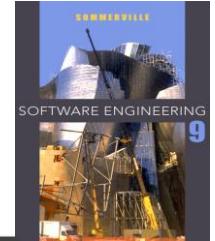
Application types

✧ 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.

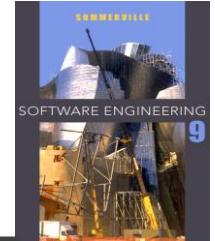
✧ Systems of systems

- These are systems that are composed of a number of other software systems.



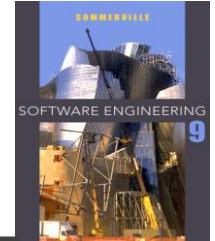
Software engineering fundamentals

- ✧ 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.



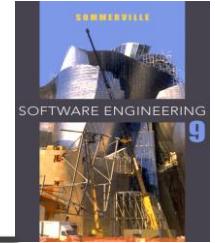
Software engineering and the web

- ✧ The Web is now a platform for running application and organizations are increasingly developing web-based systems rather than local systems.
- ✧ Web services (discussed in Chapter 19) 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 but pay according to use.



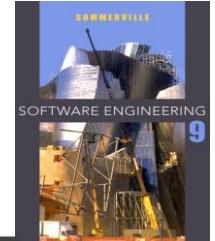
Web software engineering

- ✧ 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

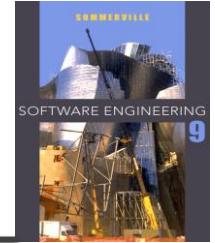
- ✧ Web-based systems are complex distributed systems but the fundamental principles of software engineering discussed previously are as applicable to them as they are to any other types of system.
- ✧ The fundamental ideas of software engineering, discussed in the previous section, apply to web-based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



Key points

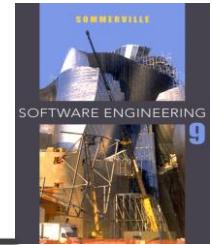
Summary

- ✧ Software engineering is an engineering discipline that is concerned with all aspects of software production.
- ✧ Essential software product attributes are maintainability, dependability and security, efficiency and acceptability.
- ✧ The high-level activities of specification, development, validation and evolution are part of all software processes.
- ✧ The fundamental notions of software engineering are universally applicable to all types of system development.



Key points

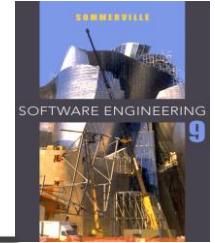
- ✧ There are many different types of system and each requires appropriate software engineering tools and techniques for their development.
- ✧ The fundamental ideas of software engineering are applicable to all types of software system.



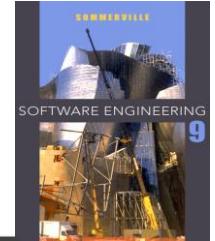
Chapter 1- Introduction

Lecture 2

Software engineering ethics



- ✧ Software engineering involves wider responsibilities than simply the application of technical skills.
- ✧ Software engineers must behave in an honest and ethically responsible way if they are to be respected as professionals.
- ✧ Ethical behaviour is more than simply upholding the law but involves following a set of principles that are morally correct.



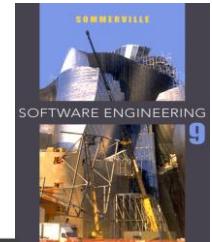
Issues of professional responsibility

✧ Confidentiality

- Engineers should normally respect the confidentiality of their employers or clients irrespective of whether or not a formal confidentiality agreement has been signed.

✧ Competence

- Engineers should not misrepresent their level of competence. They should not knowingly accept work which is outwith their competence.



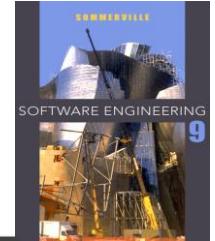
Issues of professional responsibility

✧ Intellectual property rights

- Engineers should be aware of local laws governing the use of intellectual property such as patents, copyright, etc. They should be careful to ensure that the intellectual property of employers and clients is protected.

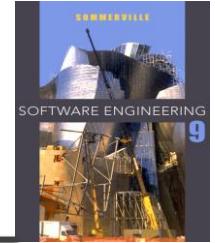
✧ Computer misuse

- Software engineers should not use their technical skills to misuse other people's computers. Computer misuse ranges from relatively trivial (game playing on an employer's machine, say) to extremely serious (dissemination of viruses).



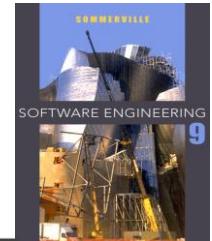
ACM/IEEE Code of Ethics

- ✧ The professional societies in the US have cooperated to produce a code of ethical practice.
- ✧ Members of these organisations sign up to the code of practice when they join.
- ✧ The Code contains eight Principles related to the behaviour of and decisions made by professional software engineers, including practitioners, educators, managers, supervisors and policy makers, as well as trainees and students of the profession.



Rationale for the code of ethics

- *Computers have a central and growing role in commerce, industry, government, medicine, education, entertainment and society at large. Software engineers are those who contribute by direct participation or by teaching, to the analysis, specification, design, development, certification, maintenance and testing of software systems.*
- *Because of their roles in developing software systems, software engineers have significant opportunities to do good or cause harm, to enable others to do good or cause harm, or to influence others to do good or cause harm. To ensure, as much as possible, that their efforts will be used for good, software engineers must commit themselves to making software engineering a beneficial and respected profession.*



The ACM/IEEE Code of Ethics

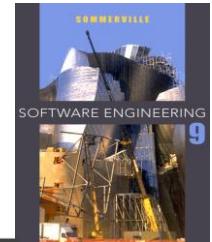
Software Engineering Code of Ethics and Professional Practice

ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices

PREAMBLE

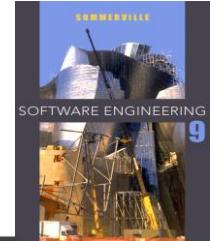
The short version of the code summarizes aspirations at a high level of the abstraction; the clauses that are included in the full version give examples and details of how these aspirations change the way we act as software engineering professionals. Without the aspirations, the details can become legalistic and tedious; without the details, the aspirations can become high sounding but empty; together, the aspirations and the details form a cohesive code.

Software engineers shall commit themselves to making the analysis, specification, design, development, testing and maintenance of software a beneficial and respected profession. In accordance with their commitment to the health, safety and welfare of the public, software engineers shall adhere to the following Eight Principles:



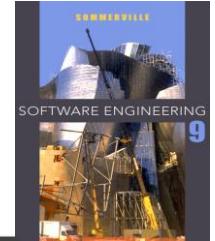
Ethical principles

1. PUBLIC - Software engineers shall act consistently with the public interest.
2. CLIENT AND EMPLOYER - Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.
3. PRODUCT - Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
4. JUDGMENT - Software engineers shall maintain integrity and independence in their professional judgment.
5. MANAGEMENT - Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
6. PROFESSION - Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
7. COLLEAGUES - Software engineers shall be fair to and supportive of their colleagues.
8. SELF - Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.



Ethical dilemmas

- ✧ Disagreement in principle with the policies of senior management.
- ✧ Your employer acts in an unethical way and releases a safety-critical system without finishing the testing of the system.
- ✧ Participation in the development of military weapons systems or nuclear systems.



Case studies

✧ A personal insulin pump

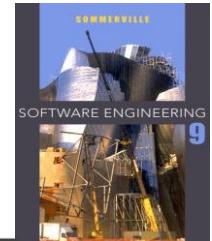
- An embedded system in an insulin pump used by diabetics to maintain blood glucose control.

✧ A mental health case patient management system

- A system used to maintain records of people receiving care for mental health problems.

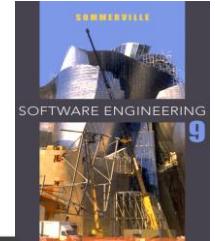
✧ A wilderness weather station

- A data collection system that collects data about weather conditions in remote areas.

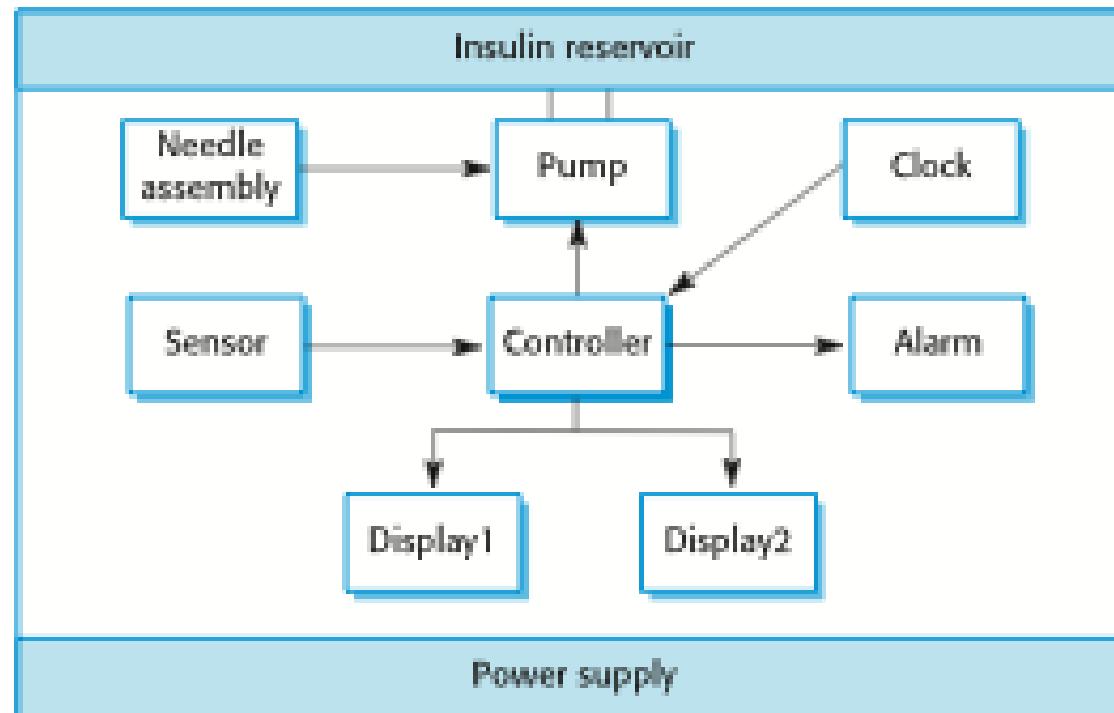


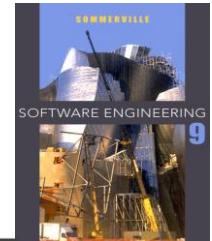
Insulin pump control system

- ✧ Collects data from a blood sugar sensor and calculates the amount of insulin required to be injected.
- ✧ Calculation based on the rate of change of blood sugar levels.
- ✧ Sends signals to a micro-pump to deliver the correct dose of insulin.
- ✧ Safety-critical system as low blood sugars can lead to brain malfunctioning, coma and death; high-blood sugar levels have long-term consequences such as eye and kidney damage.

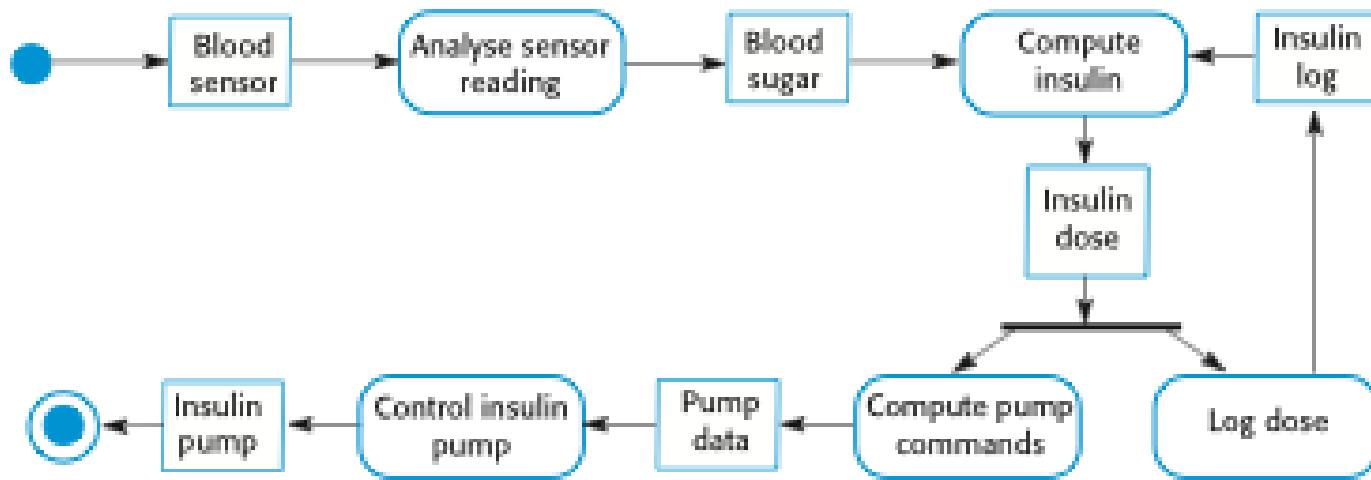


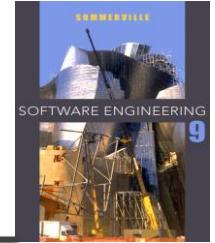
Insulin pump hardware architecture





Activity model of the insulin pump

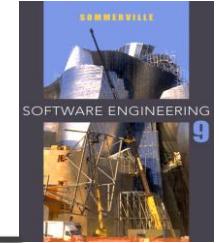




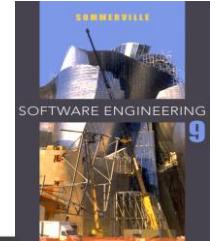
Essential high-level requirements

- ✧ The system shall be available to deliver insulin when required.
- ✧ The system shall perform reliably and deliver the correct amount of insulin to counteract the current level of blood sugar.
- ✧ The system must therefore be designed and implemented to ensure that the system always meets these requirements.

A patient information system for mental health care

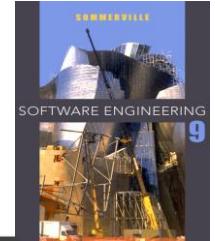


- ✧ A patient information system to support mental health care is a medical information system that maintains information about patients suffering from mental health problems and the treatments that they have received.
- ✧ Most mental health patients do not require dedicated hospital treatment but need to attend specialist clinics regularly where they can meet a doctor who has detailed knowledge of their problems.
- ✧ To make it easier for patients to attend, these clinics are not just run in hospitals. They may also be held in local medical practices or community centres.



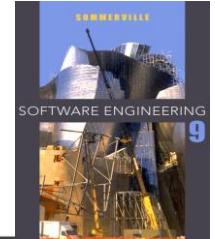
MHC-PMS

- ✧ The MHC-PMS (Mental Health Care-Patient Management System) is an information system that is intended for use in clinics.
- ✧ It makes use of a centralized database of patient information but has also been designed to run on a PC, so that it may be accessed and used from sites that do not have secure network connectivity.
- ✧ When the local systems have secure network access, they use patient information in the database but they can download and use local copies of patient records when they are disconnected.

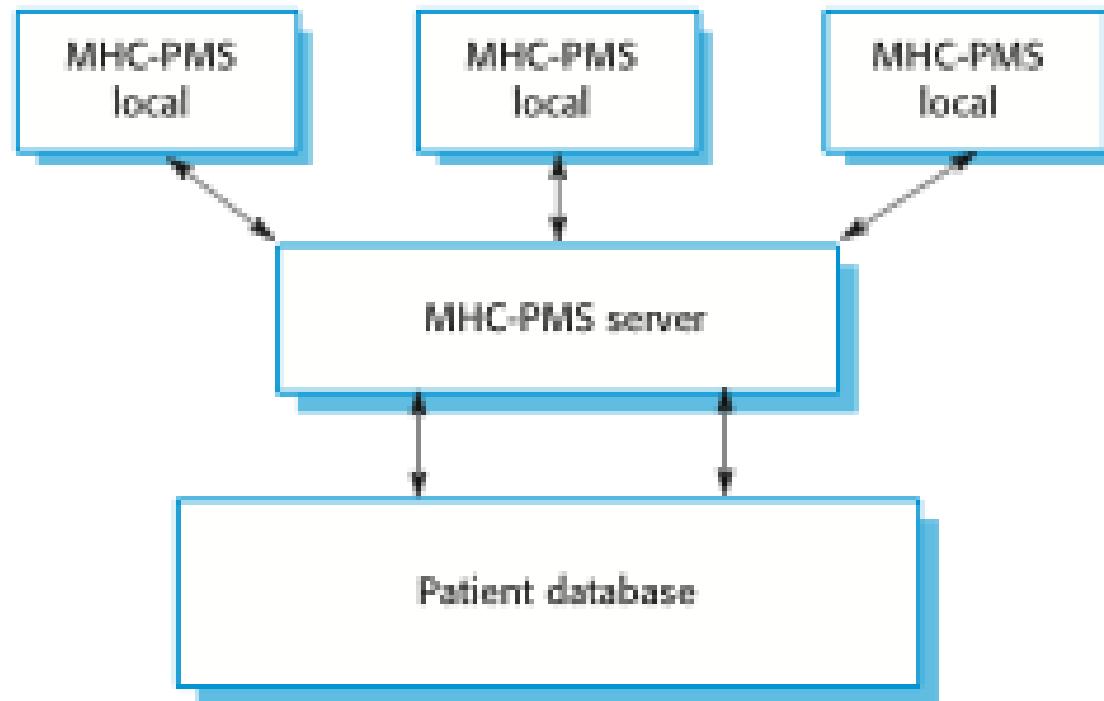


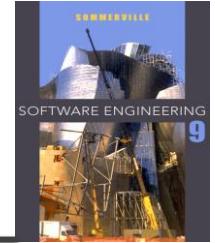
MHC-PMS goals

- ✧ To generate management information that allows health service managers to assess performance against local and government targets.
- ✧ To provide medical staff with timely information to support the treatment of patients.



The organization of the MHC-PMS





MHC-PMS key features

✧ Individual care management

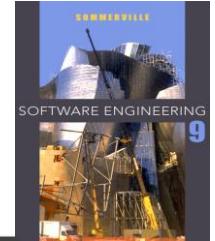
- Clinicians can create records for patients, edit the information in the system, view patient history, etc. The system supports data summaries so that doctors can quickly learn about the key problems and treatments that have been prescribed.

✧ Patient monitoring

- The system monitors the records of patients that are involved in treatment and issues warnings if possible problems are detected.

✧ Administrative reporting

- The system generates monthly management reports showing the number of patients treated at each clinic, the number of patients who have entered and left the care system, number of patients sectioned, the drugs prescribed and their costs, etc.



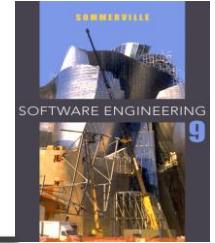
MHC-PMS concerns

✧ Privacy

- It is essential that patient information is confidential and is never disclosed to anyone apart from authorised medical staff and the patient themselves.

✧ Safety

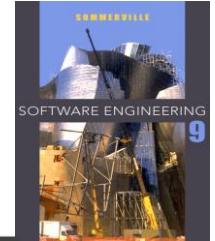
- Some mental illnesses cause patients to become suicidal or a danger to other people. Wherever possible, the system should warn medical staff about potentially suicidal or dangerous patients.
- The system must be available when needed otherwise safety may be compromised and it may be impossible to prescribe the correct medication to patients.



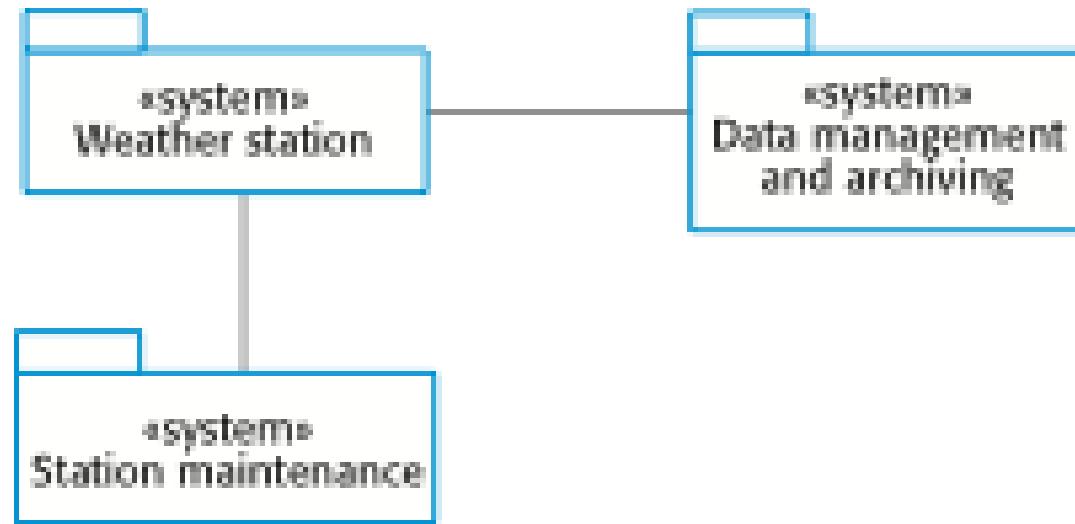
Wilderness weather station

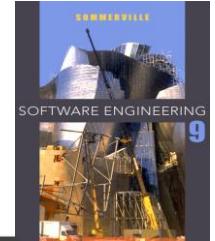
- ✧ The government of a country with large areas of wilderness decides to deploy several hundred weather stations in remote areas.
- ✧ Weather stations collect data from a set of instruments that measure temperature and pressure, sunshine, rainfall, wind speed and wind direction.
 - The weather station includes a number of instruments that measure weather parameters such as the wind speed and direction, the ground and air temperatures, the barometric pressure and the rainfall over a 24-hour period. Each of these instruments is controlled by a software system that takes parameter readings periodically and manages the data collected from the instruments.





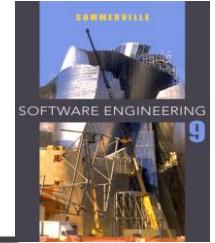
The weather station's environment





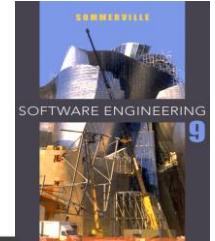
Weather information system

- ✧ The weather station system
 - This is responsible for collecting weather data, carrying out some initial data processing and transmitting it to the data management system.
- ✧ The data management and archiving system
 - This system collects the data from all of the wilderness weather stations, carries out data processing and analysis and archives the data.
- ✧ The station maintenance system
 - This system can communicate by satellite with all wilderness weather stations to monitor the health of these systems and provide reports of problems.



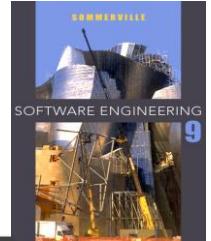
Additional software functionality

- ✧ Monitor the instruments, power and communication hardware and report faults to the management system.
- ✧ Manage the system power, ensuring that batteries are charged whenever the environmental conditions permit but also that generators are shut down in potentially damaging weather conditions, such as high wind.
- ✧ Support dynamic reconfiguration where parts of the software are replaced with new versions and where backup instruments are switched into the system in the event of system failure.



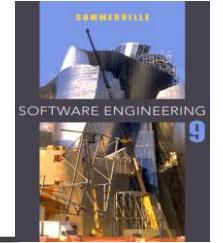
Key points

- ✧ Software engineers have responsibilities to the engineering profession and society. They should not simply be concerned with technical issues.
- ✧ Professional societies publish codes of conduct which set out the standards of behaviour expected of their members.
- ✧ Three case studies are used in the book:
 - An embedded insulin pump control system
 - A system for mental health care patient management
 - A wilderness weather station



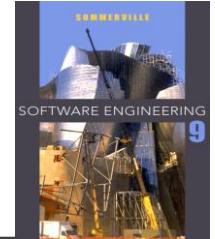
Course structure and organization

- ✧ *Add your own material here about how you will be running the course*



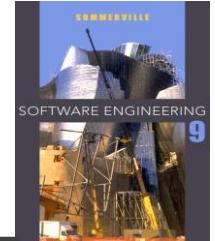
Chapter 2 – Software Processes

Lecture 1



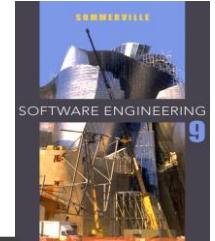
Topics covered

- ✧ Software process models
- ✧ Process activities
- ✧ Coping with change
- ✧ The Rational Unified Process
 - An example of a modern software process.



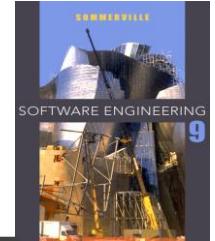
The software process

- ✧ A structured set of activities required to develop a software system.
- ✧ Many different software processes but all involve:
 - Specification – defining what the system should do;
 - Design and implementation – defining the organization of the system and implementing the system;
 - Validation – checking that it does what the customer wants;
 - Evolution – changing the system in response to changing customer needs.
- ✧ A software process model is an abstract representation of a process. It presents a description of a process from some particular perspective.



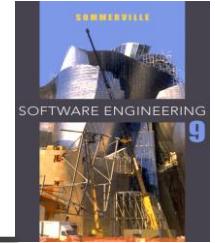
Software process descriptions

- ✧ When we describe and discuss processes, we usually talk about the activities in these processes such as specifying a data model, designing a user interface, etc. and the ordering of these activities.
- ✧ Process descriptions may also include:
 - Products, which are the outcomes of a process activity;
 - Roles, which reflect the responsibilities of the people involved in the process;
 - Pre- and post-conditions, which are statements that are true before and after a process activity has been enacted or a product produced.



Plan-driven and agile processes

- ✧ Plan-driven processes are processes where all of the process activities are planned in advance and progress is measured against this plan.
- ✧ In agile processes, planning is incremental and it is easier to change the process to reflect changing customer requirements.
- ✧ In practice, most practical processes include elements of both plan-driven and agile approaches.
- ✧ There are no right or wrong software processes.



Software process models

✧ The waterfall model

- Plan-driven model. Separate and distinct phases of specification and development.

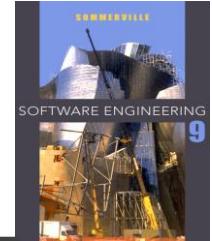
✧ Incremental development

- Specification, development and validation are interleaved. May be plan-driven or agile.

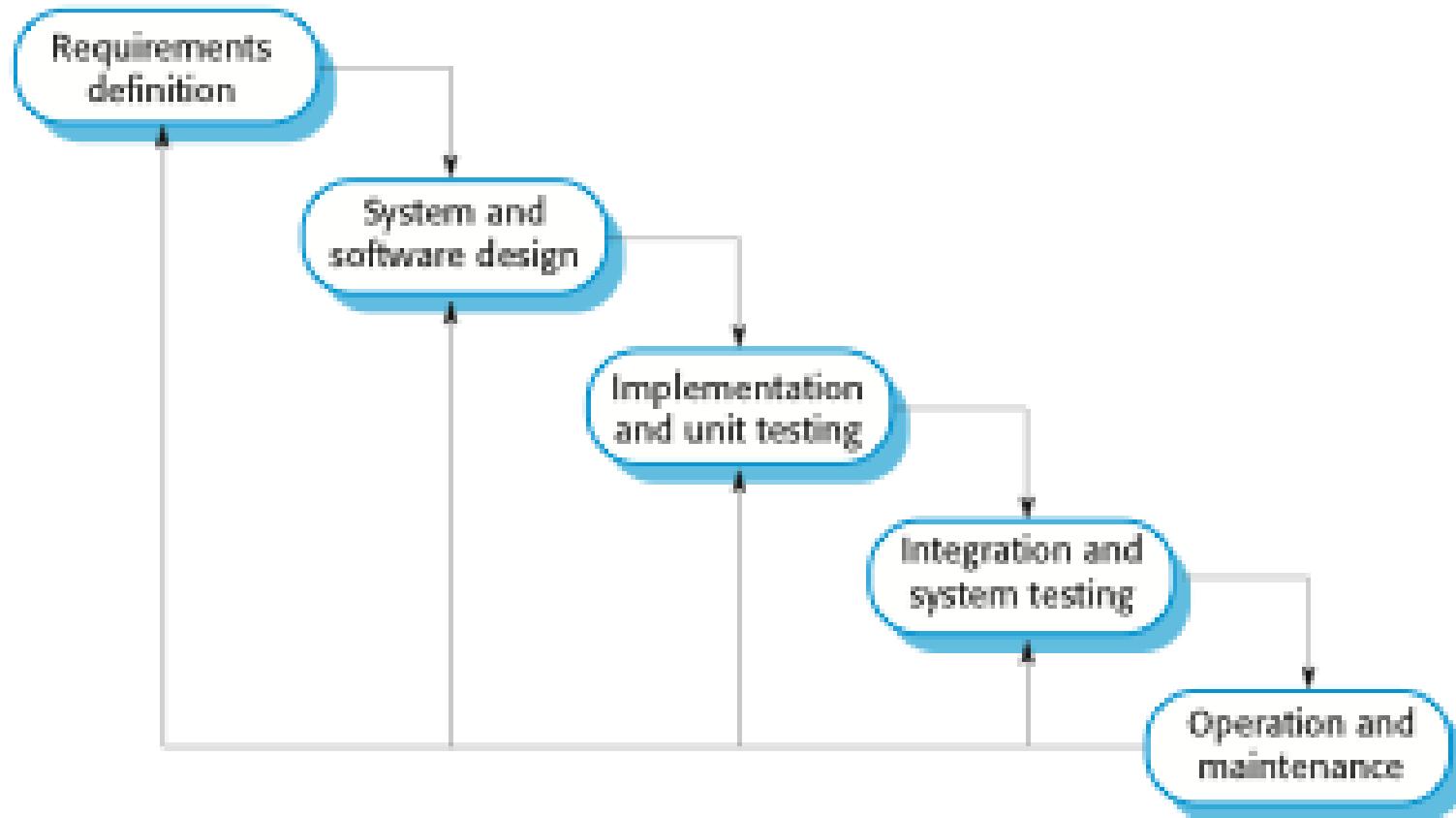
✧ Reuse-oriented software engineering

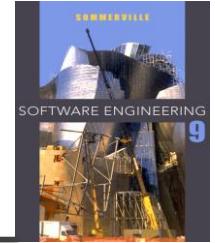
- The system is assembled from existing components. May be plan-driven or agile.

✧ In practice, most large systems are developed using a process that incorporates elements from all of these models.



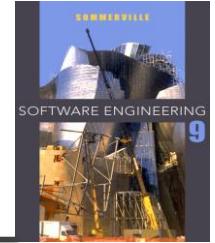
The waterfall model





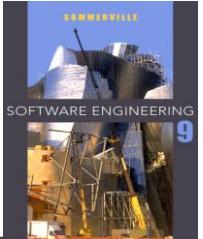
Waterfall model phases

- ✧ There are separate identified phases in the waterfall model:
 - Requirements analysis and definition
 - System and software design
 - Implementation and unit testing
 - Integration and system testing
 - Operation and maintenance
- ✧ The main drawback of the waterfall model is the difficulty of accommodating change after the process is underway. In principle, a phase has to be complete before moving onto the next phase.

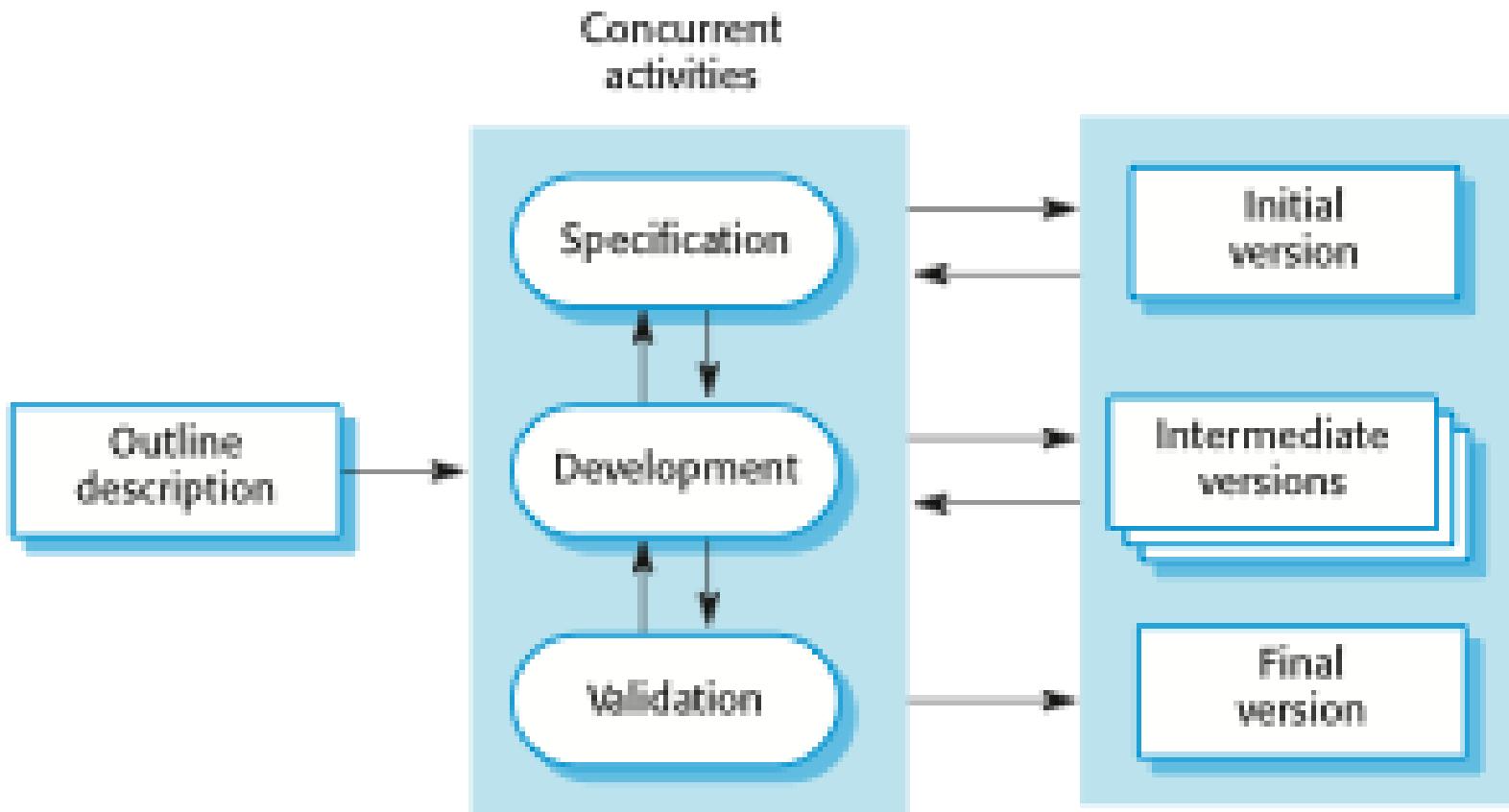


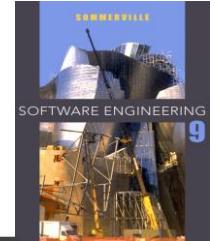
Waterfall model problems

- ✧ Inflexible partitioning of the project into distinct stages makes it difficult to respond to changing customer requirements.
 - Therefore, this model is only appropriate when the requirements are well-understood and changes will be fairly limited during the design process.
 - Few business systems have stable requirements.
- ✧ The waterfall model is mostly used for large systems engineering projects where a system is developed at several sites.
 - In those circumstances, the plan-driven nature of the waterfall model helps coordinate the work.



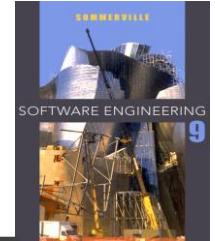
Incremental development





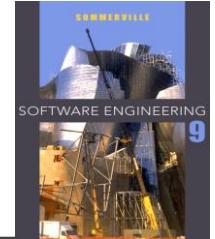
Incremental development benefits

- ✧ The cost of accommodating changing customer requirements is reduced.
 - The amount of analysis and documentation that has to be redone is much less than is required with the waterfall model.
- ✧ It is easier to get customer feedback on the development work that has been done.
 - Customers can comment on demonstrations of the software and see how much has been implemented.
- ✧ More rapid delivery and deployment of useful software to the customer is possible.
 - Customers are able to use and gain value from the software earlier than is possible with a waterfall process.



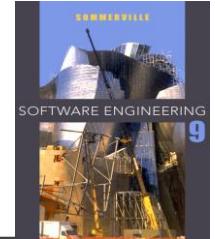
Incremental development problems

- ✧ The process is not visible.
 - Managers need regular deliverables to measure progress. If systems are developed quickly, it is not cost-effective to produce documents that reflect every version of the system.
- ✧ System structure tends to degrade as new increments are added.
 - Unless time and money is spent on refactoring to improve the software, regular change tends to corrupt its structure. Incorporating further software changes becomes increasingly difficult and costly.

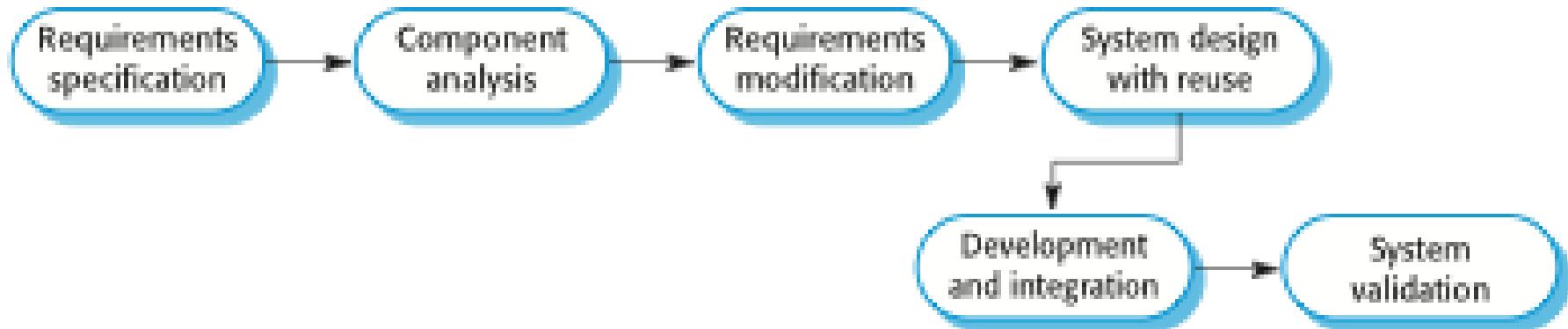


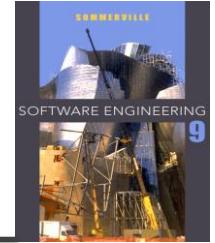
Reuse-oriented software engineering

- ✧ Based on systematic reuse where systems are integrated from existing components or COTS (Commercial-off-the-shelf) systems.
- ✧ Process stages
 - Component analysis;
 - Requirements modification;
 - System design with reuse;
 - Development and integration.
- ✧ Reuse is now the standard approach for building many types of business system
 - Reuse covered in more depth in Chapter 16.



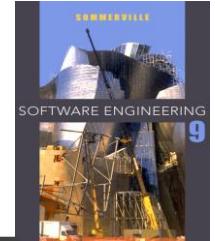
Reuse-oriented software engineering





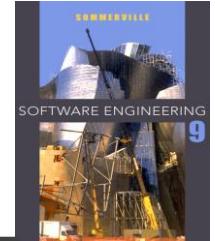
Types of software component

- ✧ Web services that are developed according to service standards and which are available for remote invocation.
- ✧ Collections of objects that are developed as a package to be integrated with a component framework such as .NET or J2EE.
- ✧ Stand-alone software systems (COTS) that are configured for use in a particular environment.



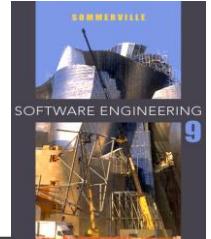
Process activities

- ✧ Real software processes are inter-leaved sequences of technical, collaborative and managerial activities with the overall goal of specifying, designing, implementing and testing a software system.
- ✧ The four basic process activities of specification, development, validation and evolution are organized differently in different development processes. In the waterfall model, they are organized in sequence, whereas in incremental development they are inter-leaved.

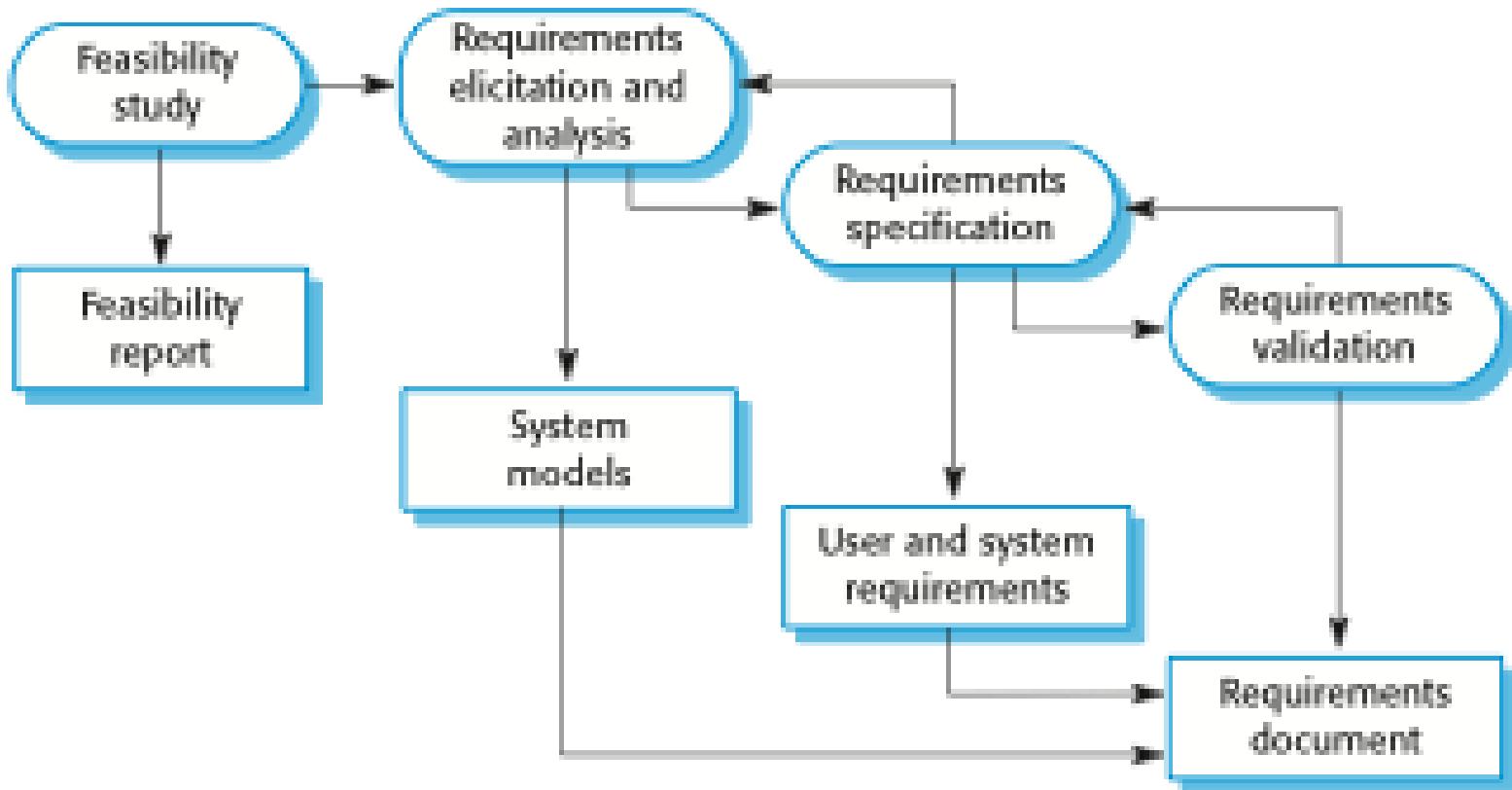


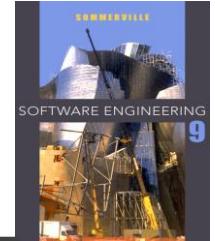
Software specification

- ✧ The process of establishing what services are required and the constraints on the system's operation and development.
- ✧ Requirements engineering process
 - Feasibility study
 - Is it technically and financially feasible to build the system?
 - Requirements elicitation and analysis
 - What do the system stakeholders require or expect from the system?
 - Requirements specification
 - Defining the requirements in detail
 - Requirements validation
 - Checking the validity of the requirements



The requirements engineering process

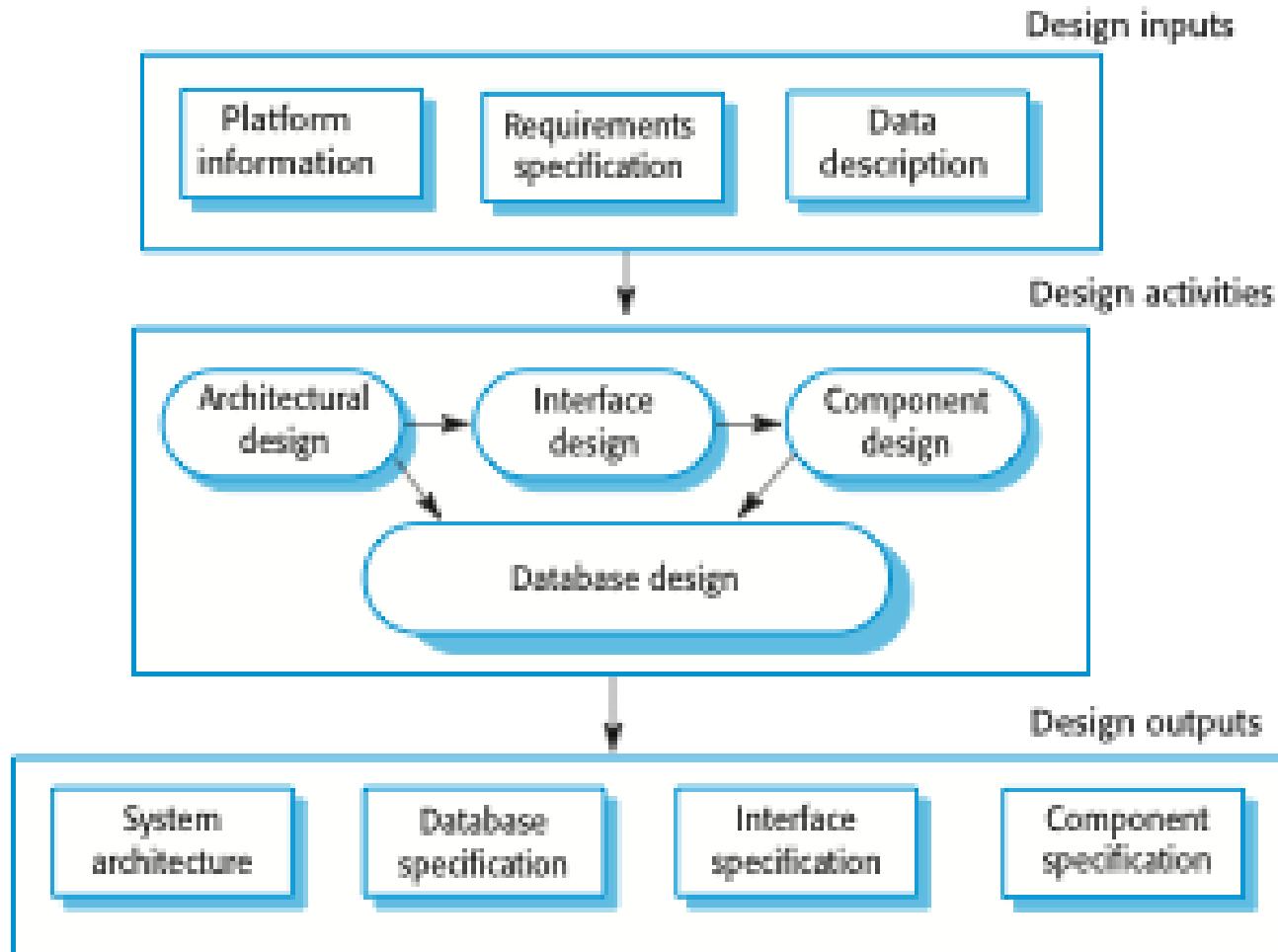
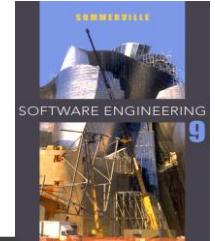


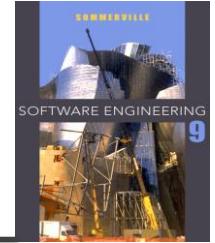


Software design and implementation

- ✧ The process of converting the system specification into an executable system.
- ✧ Software design
 - Design a software structure that realises the specification;
- ✧ Implementation
 - Translate this structure into an executable program;
- ✧ The activities of design and implementation are closely related and may be inter-leaved.

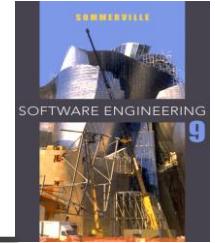
A general model of the design process





Design activities

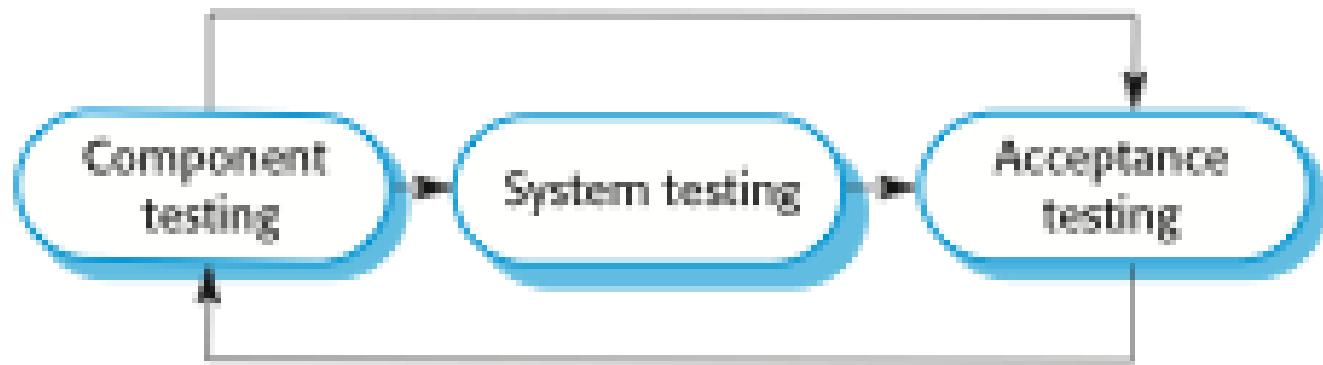
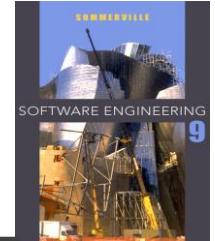
- ✧ *Architectural design*, where you identify the overall structure of the system, the principal components (sometimes called sub-systems or modules), their relationships and how they are distributed.
- ✧ *Interface design*, where you define the interfaces between system components.
- ✧ *Component design*, where you take each system component and design how it will operate.
- ✧ *Database design*, where you design the system data structures and how these are to be represented in a database.

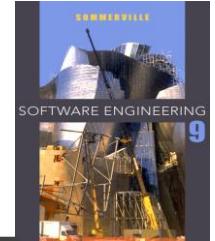


Software validation

- ✧ Verification and validation (V & V) is intended to show that a system conforms to its specification and meets the requirements of the system customer.
- ✧ Involves checking and review processes and system testing.
- ✧ System testing involves executing the system with test cases that are derived from the specification of the real data to be processed by the system.
- ✧ Testing is the most commonly used V & V activity.

Stages of testing

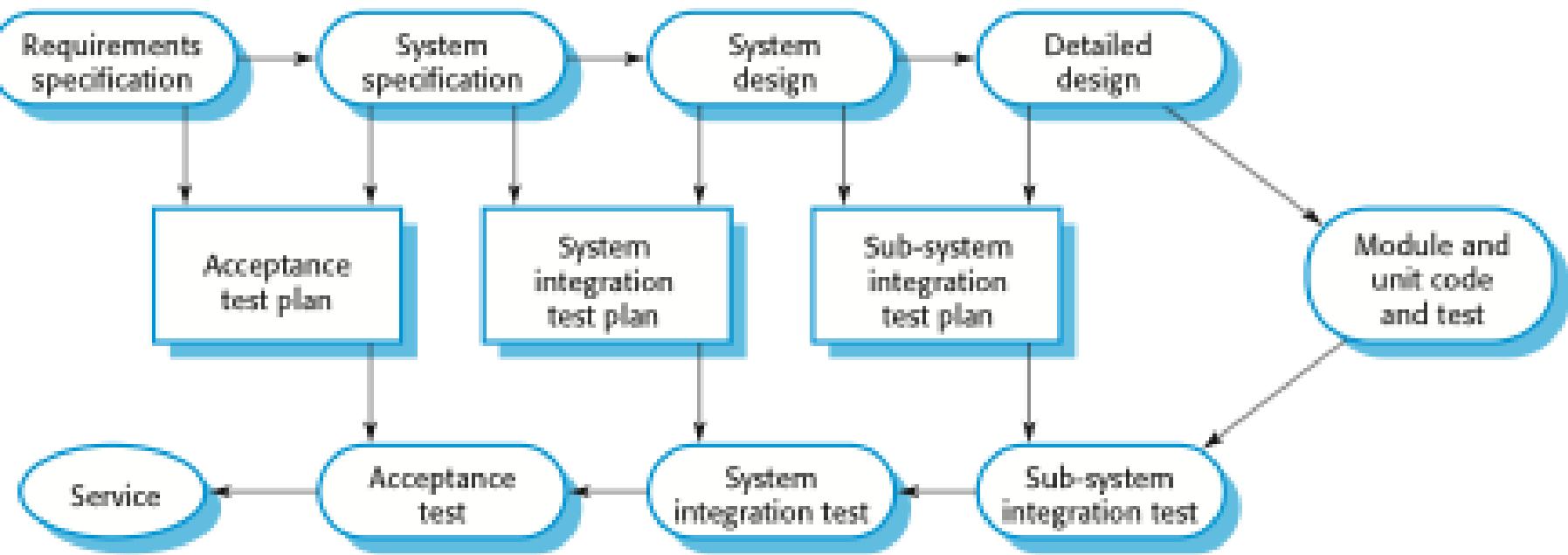
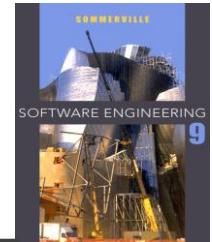


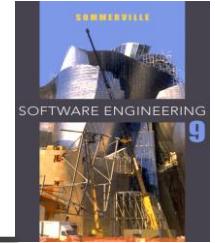


Testing stages

- ✧ Development or component testing
 - Individual components are tested independently;
 - Components may be functions or objects or coherent groupings of these entities.
- ✧ System testing
 - Testing of the system as a whole. Testing of emergent properties is particularly important.
- ✧ Acceptance testing
 - Testing with customer data to check that the system meets the customer's needs.

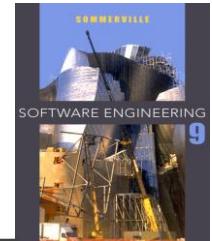
Testing phases in a plan-driven software process



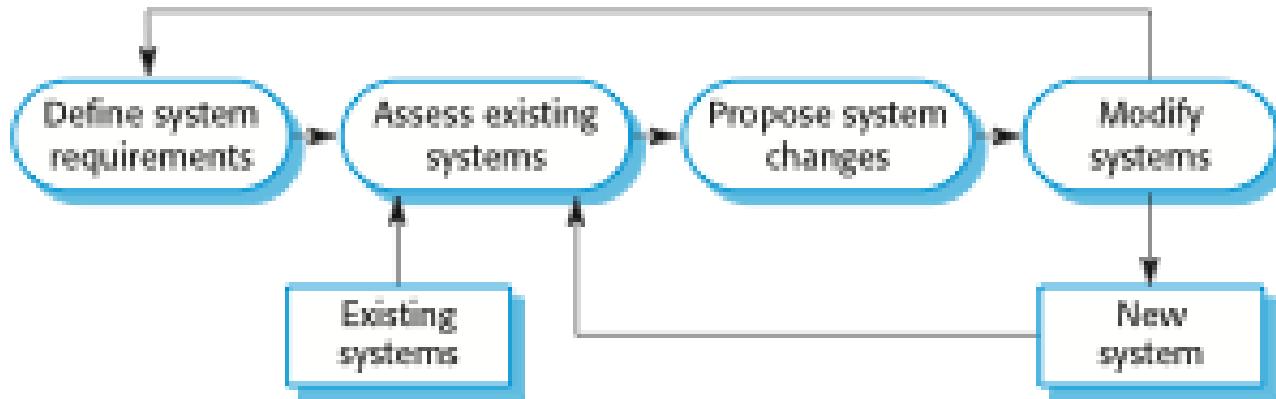


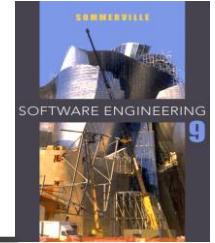
Software evolution

- ✧ Software is inherently flexible and can change.
- ✧ As requirements change through changing business circumstances, the software that supports the business must also evolve and change.
- ✧ Although there has been a demarcation between development and evolution (maintenance) this is increasingly irrelevant as fewer and fewer systems are completely new.



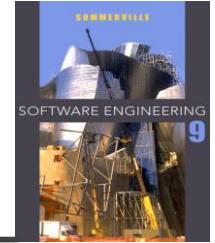
System evolution





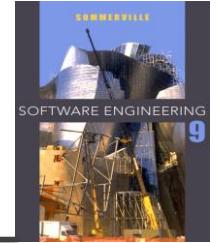
Key points

- ✧ Software processes are the activities involved in producing a software system. Software process models are abstract representations of these processes.
- ✧ General process models describe the organization of software processes. Examples of these general models include the ‘waterfall’ model, incremental development, and reuse-oriented development.



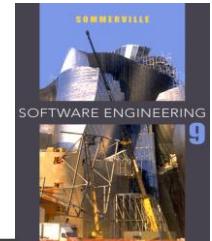
Key points

- ✧ Requirements engineering is the process of developing a software specification.
- ✧ Design and implementation processes are concerned with transforming a requirements specification into an executable software system.
- ✧ Software validation is the process of checking that the system conforms to its specification and that it meets the real needs of the users of the system.
- ✧ Software evolution takes place when you change existing software systems to meet new requirements. The software must evolve to remain useful.



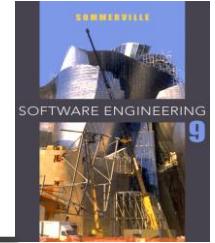
Chapter 2 – Software Processes

Lecture 2



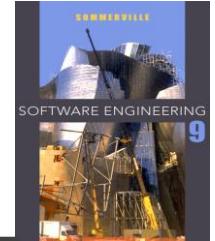
Coping with change

- ✧ Change is inevitable in all large software projects.
 - Business changes lead to new and changed system requirements
 - New technologies open up new possibilities for improving implementations
 - Changing platforms require application changes
- ✧ Change leads to rework so the costs of change include both rework (e.g. re-analysing requirements) as well as the costs of implementing new functionality



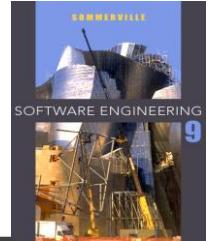
Reducing the costs of rework

- ✧ Change avoidance, where the software process includes activities that can anticipate possible changes before significant rework is required.
 - For example, a prototype system may be developed to show some key features of the system to customers.
- ✧ Change tolerance, where the process is designed so that changes can be accommodated at relatively low cost.
 - This normally involves some form of incremental development. Proposed changes may be implemented in increments that have not yet been developed. If this is impossible, then only a single increment (a small part of the system) may have to be altered to incorporate the change.



Software prototyping

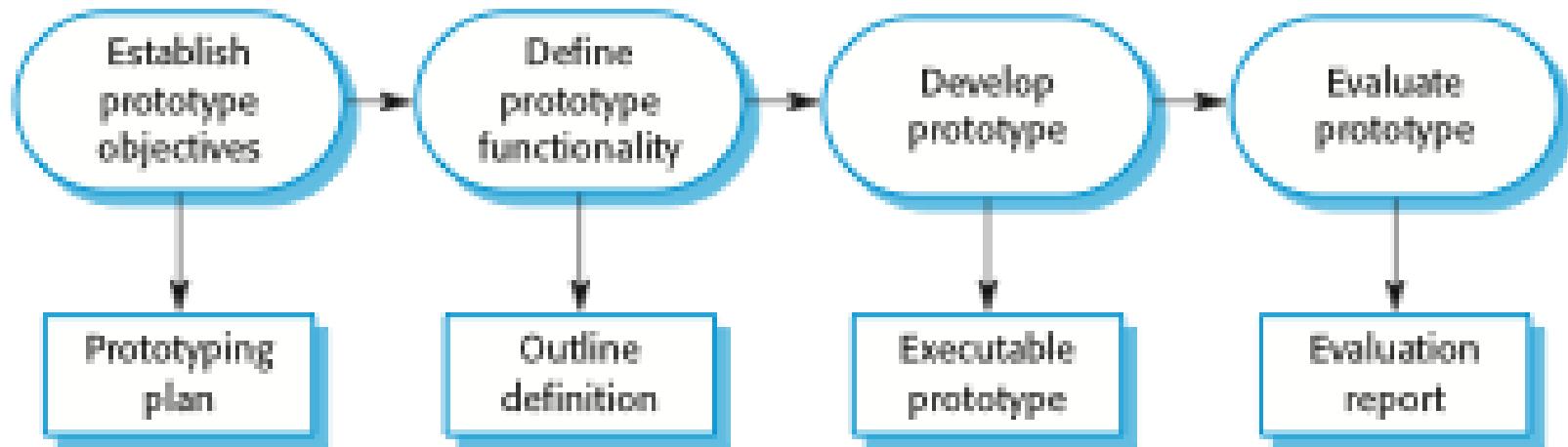
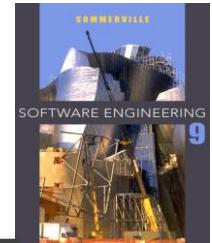
- ✧ A prototype is an initial version of a system used to demonstrate concepts and try out design options.
- ✧ A prototype can be used in:
 - The requirements engineering process to help with requirements elicitation and validation;
 - In design processes to explore options and develop a UI design;
 - In the testing process to run back-to-back tests.

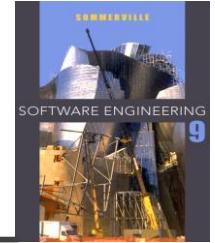


Benefits of prototyping

- ✧ Improved system usability.
- ✧ A closer match to users' real needs.
- ✧ Improved design quality.
- ✧ Improved maintainability.
- ✧ Reduced development effort.

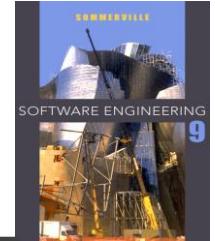
The process of prototype development





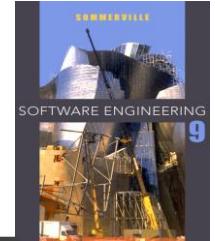
Prototype development

- ✧ May be based on rapid prototyping languages or tools
- ✧ May involve leaving out functionality
 - Prototype should focus on areas of the product that are not well-understood;
 - Error checking and recovery may not be included in the prototype;
 - Focus on functional rather than non-functional requirements such as reliability and security



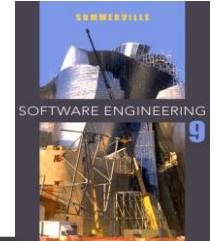
Throw-away prototypes

- ✧ Prototypes should be discarded after development as they are not a good basis for a production system:
 - It may be impossible to tune the system to meet non-functional requirements;
 - Prototypes are normally undocumented;
 - The prototype structure is usually degraded through rapid change;
 - The prototype probably will not meet normal organisational quality standards.



Incremental delivery

- ✧ Rather than deliver the system as a single delivery, the development and delivery is broken down into increments with each increment delivering part of the required functionality.
- ✧ User requirements are prioritised and the highest priority requirements are included in early increments.
- ✧ Once the development of an increment is started, the requirements are frozen though requirements for later increments can continue to evolve.



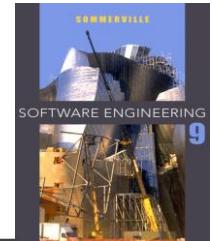
Incremental development and delivery

✧ Incremental development

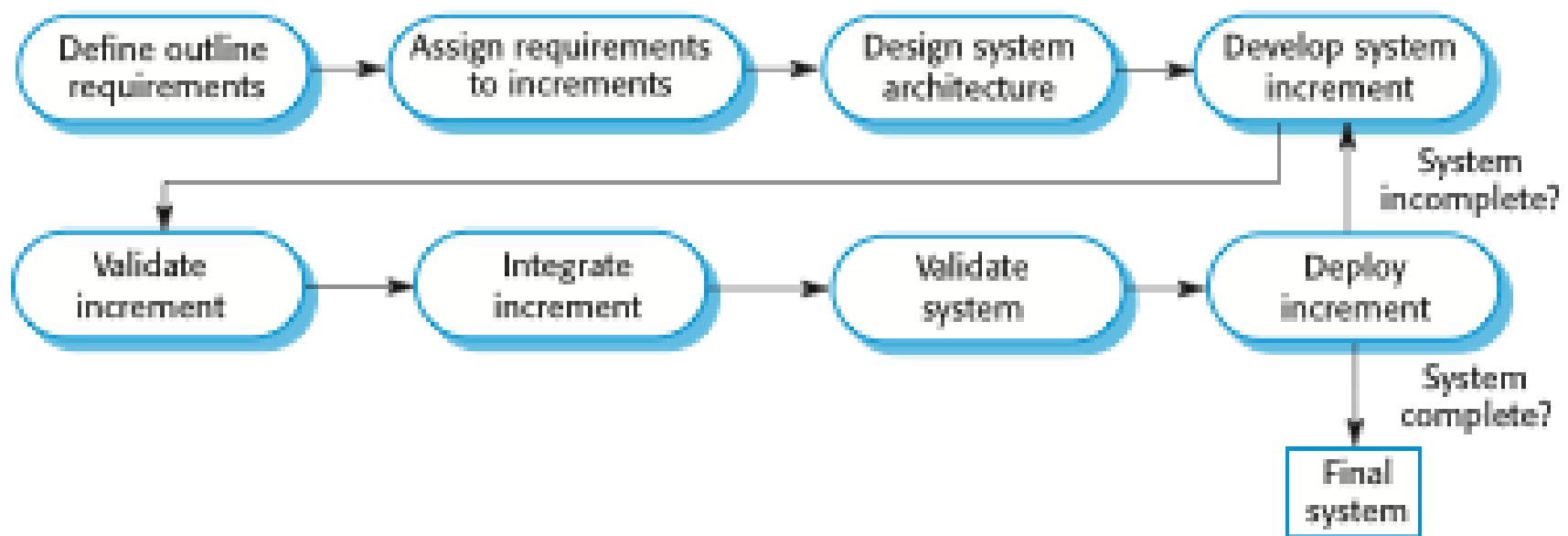
- Develop the system in increments and evaluate each increment before proceeding to the development of the next increment;
- Normal approach used in agile methods;
- Evaluation done by user/customer proxy.

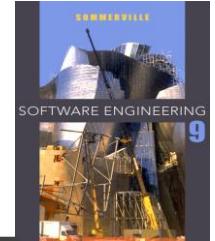
✧ Incremental delivery

- Deploy an increment for use by end-users;
- More realistic evaluation about practical use of software;
- Difficult to implement for replacement systems as increments have less functionality than the system being replaced.



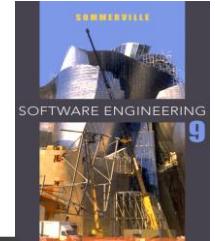
Incremental delivery





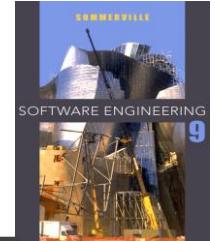
Incremental delivery advantages

- ✧ Customer value can be delivered with each increment so system functionality is available earlier.
- ✧ Early increments act as a prototype to help elicit requirements for later increments.
- ✧ Lower risk of overall project failure.
- ✧ The highest priority system services tend to receive the most testing.



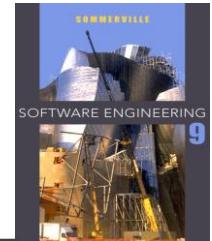
Incremental delivery problems

- ✧ Most systems require a set of basic facilities that are used by different parts of the system.
 - As requirements are not defined in detail until an increment is to be implemented, it can be hard to identify common facilities that are needed by all increments.
- ✧ The essence of iterative processes is that the specification is developed in conjunction with the software.
 - However, this conflicts with the procurement model of many organizations, where the complete system specification is part of the system development contract.

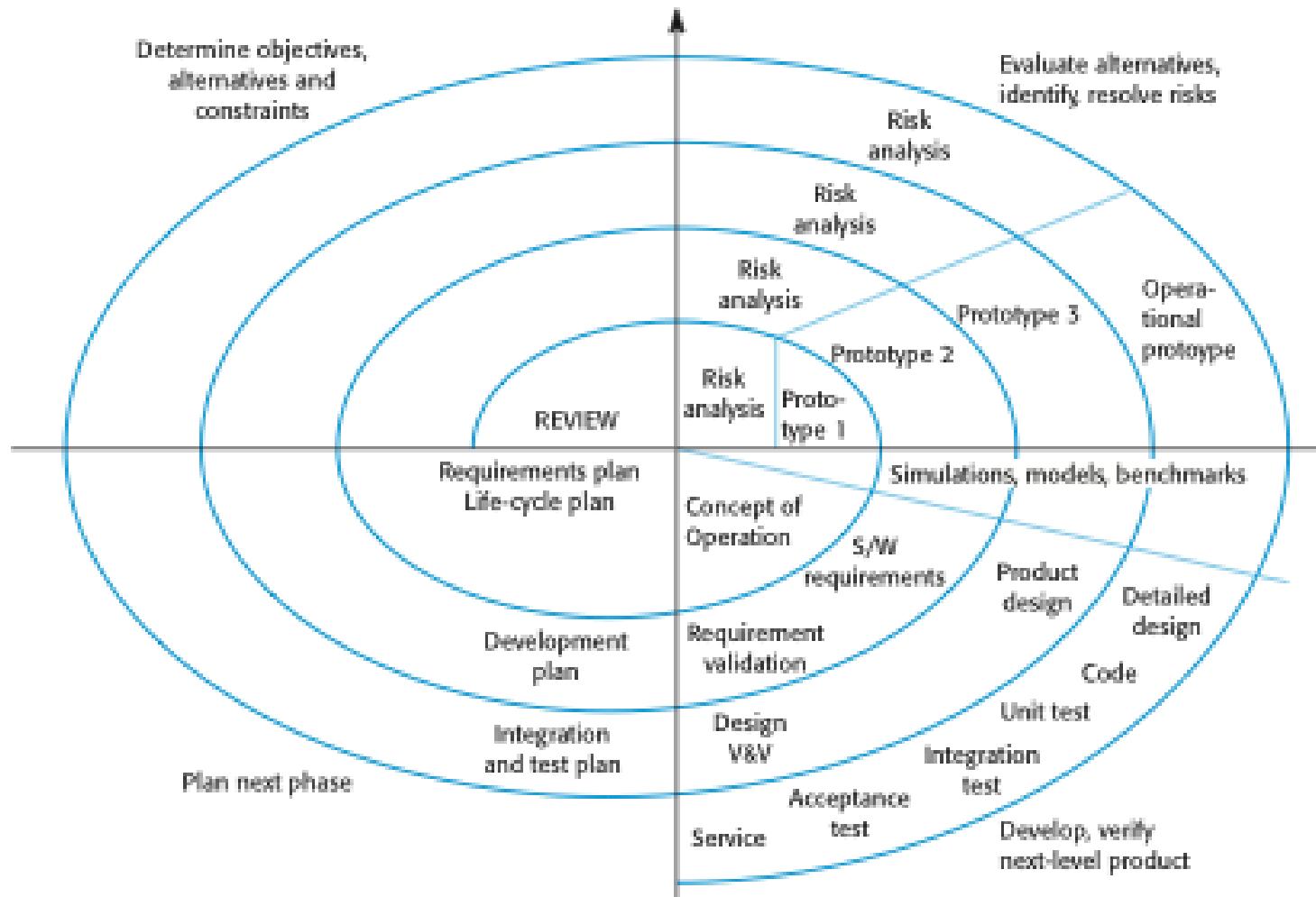


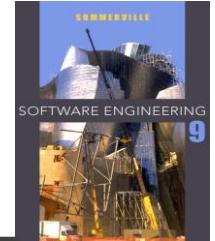
Boehm's spiral model

- ✧ Process is represented as a spiral rather than as a sequence of activities with backtracking.
- ✧ Each loop in the spiral represents a phase in the process.
- ✧ No fixed phases such as specification or design - loops in the spiral are chosen depending on what is required.
- ✧ Risks are explicitly assessed and resolved throughout the process.



Boehm's spiral model of the software process





Spiral model sectors

✧ Objective setting

- Specific objectives for the phase are identified.

✧ Risk assessment and reduction

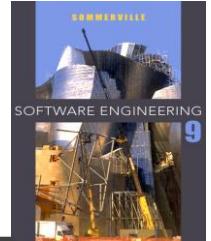
- Risks are assessed and activities put in place to reduce the key risks.

✧ Development and validation

- A development model for the system is chosen which can be any of the generic models.

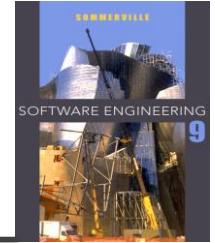
✧ Planning

- The project is reviewed and the next phase of the spiral is planned.



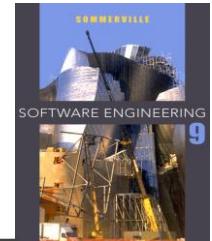
Spiral model usage

- ✧ Spiral model has been very influential in helping people think about iteration in software processes and introducing the risk-driven approach to development.
- ✧ In practice, however, the model is rarely used as published for practical software development.

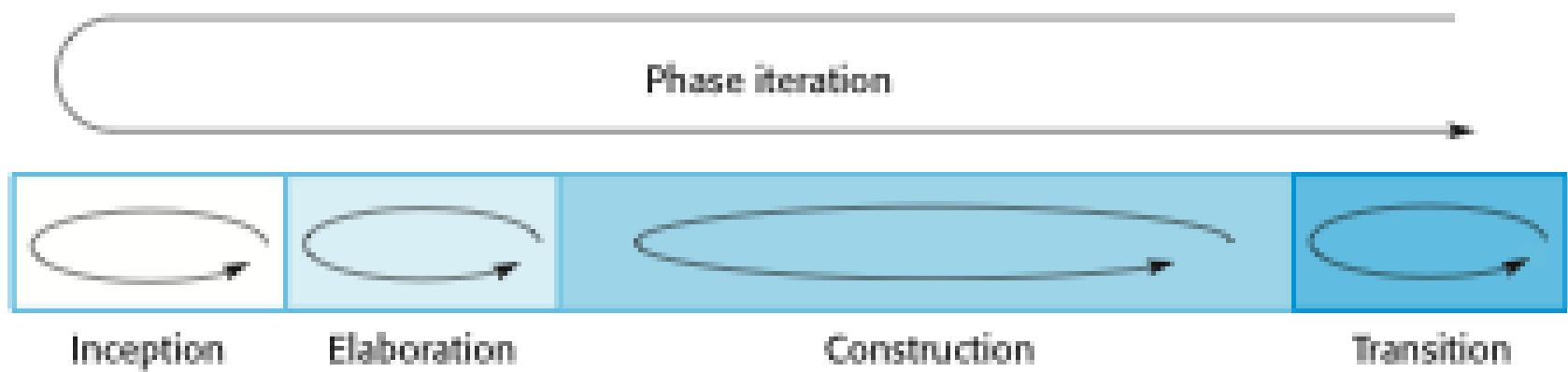


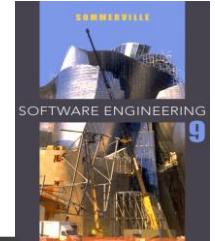
The Rational Unified Process

- ✧ A modern generic process derived from the work on the UML and associated process.
- ✧ Brings together aspects of the 3 generic process models discussed previously.
- ✧ Normally described from 3 perspectives
 - A dynamic perspective that shows phases over time;
 - A static perspective that shows process activities;
 - A practive perspective that suggests good practice.



Phases in the Rational Unified Process





RUP phases

✧ Inception

- Establish the business case for the system.

✧ Elaboration

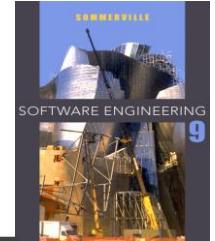
- Develop an understanding of the problem domain and the system architecture.

✧ Construction

- System design, programming and testing.

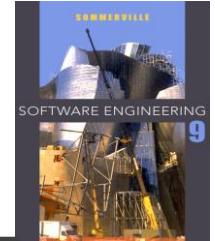
✧ Transition

- Deploy the system in its operating environment.



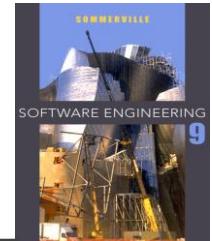
RUP iteration

- ✧ In-phase iteration
 - Each phase is iterative with results developed incrementally.
- ✧ Cross-phase iteration
 - As shown by the loop in the RUP model, the whole set of phases may be enacted incrementally.



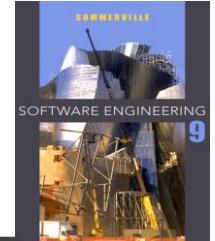
Static workflows in the Rational Unified Process

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.



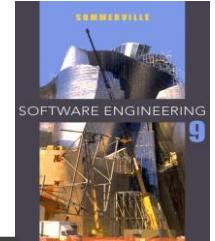
Static workflows in the Rational Unified Process

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 (see Chapter 25).
Project management	This supporting workflow manages the system development (see Chapters 22 and 23).
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

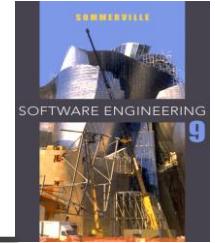
- Use graphical UML models to present static and dynamic views of the software.

✧ Verify software quality

- Ensure that the software meets organizational quality standards.

✧ Control changes to software

- Manage software changes using a change management system and configuration management tools.



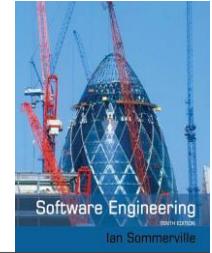
Key points

- ✧ Processes should include activities to cope with change. This may involve a prototyping phase that helps avoid poor decisions on requirements and design.
- ✧ Processes may be structured for iterative development and delivery so that changes may be made without disrupting the system as a whole.
- ✧ The Rational Unified Process is a modern generic process model that is organized into phases (inception, elaboration, construction and transition) but separates activities (requirements, analysis and design, etc.) from these phases.



Chapter 3 – Agile Software Development

Topics covered



- ✧ Agile methods
- ✧ Agile development techniques
- ✧ Agile project management
- ✧ Scaling agile methods

Rapid software development



- ✧ Rapid development and delivery is now often the most important requirement for software systems
 - Businesses operate in a fast –changing requirement and it is practically impossible to produce a set of stable software requirements
 - Software has to evolve quickly to reflect changing business needs.
- ✧ Plan-driven development is essential for some types of system but does not meet these business needs.
- ✧ Agile development methods emerged in the late 1990s whose aim was to radically reduce the delivery time for working software systems

Agile development

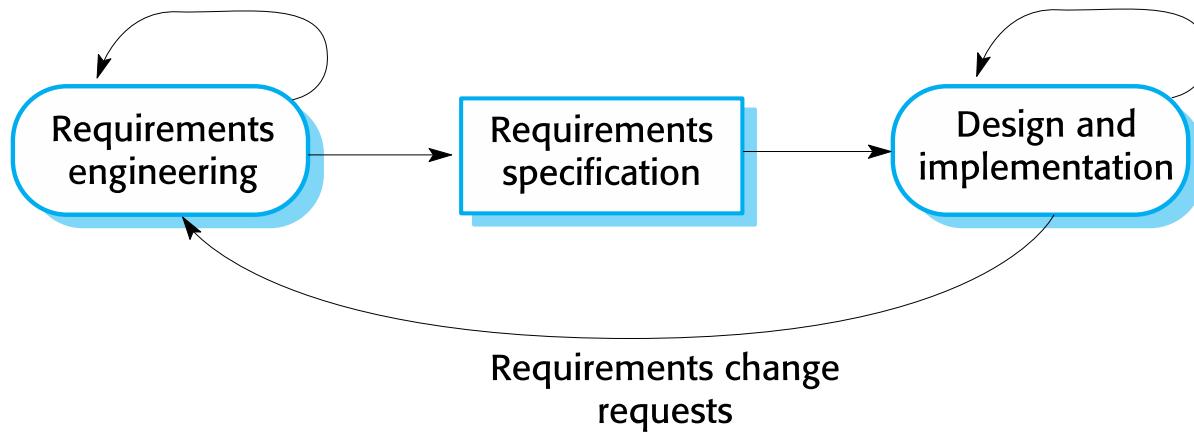


- ✧ Program specification, design and implementation are inter-leaved
- ✧ The system is developed as a series of versions or increments with stakeholders involved in version specification and evaluation
- ✧ Frequent delivery of new versions for evaluation
- ✧ Extensive tool support (e.g. automated testing tools) used to support development.
- ✧ Minimal documentation – focus on working code

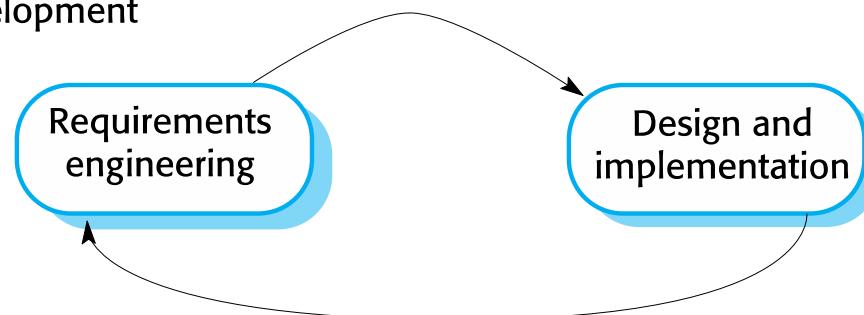
Plan-driven and agile development



Plan-based development



Agile development



Plan-driven and agile development



✧ Plan-driven development

- A plan-driven approach to software engineering is based around separate development stages with the outputs to be produced at each of these stages planned in advance.
- Not necessarily waterfall model – plan-driven, incremental development is possible
- Iteration occurs within activities.

✧ Agile development

- Specification, design, implementation and testing are interleaved and the outputs from the development process are decided through a process of negotiation during the software development process.



Agile methods

Agile methods



- ✧ Dissatisfaction with the overheads involved in software design methods of the 1980s and 1990s led to the creation of agile methods. These methods:
 - Focus on the code rather than the design
 - Are based on an iterative approach to software development
 - Are intended to deliver working software quickly and evolve this quickly to meet changing requirements.
- ✧ The aim of agile methods is to reduce overheads in the software process (e.g. by limiting documentation) and to be able to respond quickly to changing requirements without excessive rework.

Agile manifesto



- ✧ We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
 - Individuals and interactions over processes and tools
 - Working software over comprehensive documentation
 - Customer collaboration over contract negotiation
 - Responding to change over following a plan
- ✧ That is, while there is value in the items on the right, we value the items on the left more.

The principles of agile methods



Principle	Description
Customer involvement	Customers should be closely involved throughout the development process. Their role is provide and prioritize new system requirements and to evaluate the iterations of the system.
Incremental delivery	The software is developed in increments with the customer specifying the requirements to be included in each increment.
People not process	The skills of the development team should be recognized and exploited. Team members should be left to develop their own ways of working without prescriptive processes.
Embrace change	Expect the system requirements to change and so design the system to accommodate these changes.
Maintain simplicity	Focus on simplicity in both the software being developed and in the development process. Wherever possible, actively work to eliminate complexity from the system.

Agile method applicability



- ✧ Product development where a software company is developing a small or medium-sized product for sale.
 - Virtually all software products and apps are now developed using an agile approach
- ✧ Custom system development within an organization, where there is a clear commitment from the customer to become involved in the development process and where there are few external rules and regulations that affect the software.



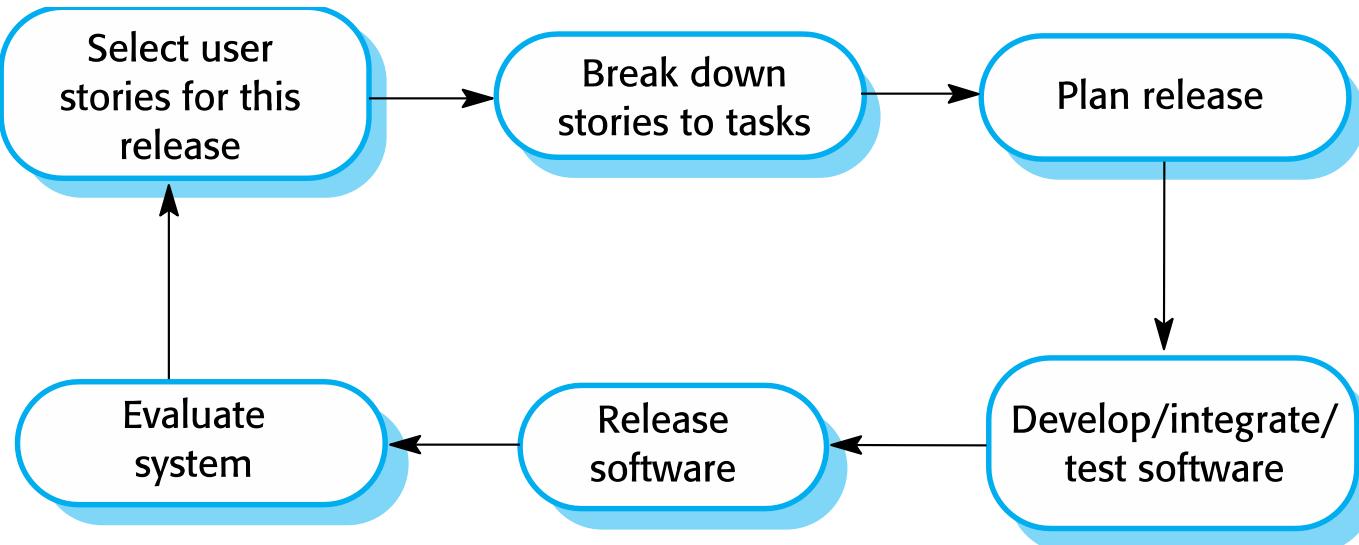
Agile development techniques

Extreme programming



- ✧ A very influential agile method, developed in the late 1990s, that introduced a range of agile development techniques.
- ✧ Extreme Programming (XP) takes an ‘extreme’ approach to iterative development.
 - New versions may be built several times per day;
 - Increments are delivered to customers every 2 weeks;
 - All tests must be run for every build and the build is only accepted if tests run successfully.

The extreme programming release cycle



Extreme programming practices (a)



Principle or practice	Description
Incremental planning	Requirements are recorded on story cards and the stories to be included in a release are determined by the time available and their relative priority. The developers break these stories into development 'Tasks'. See Figures 3.5 and 3.6.
Small releases	The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the first release.
Simple design	Enough design is carried out to meet the current requirements and no more.
Test-first development	An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.
Refactoring	All developers are expected to refactor the code continuously as soon as possible code improvements are found. This keeps the code simple and maintainable.

Extreme programming practices (b)



Pair programming	Developers work in pairs, checking each other's work and providing the support to always do a good job.
Collective ownership	The pairs of developers work on all areas of the system, so that no islands of expertise develop and all the developers take responsibility for all of the code. Anyone can change anything.
Continuous integration	As soon as the work on a task is complete, it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.
Sustainable pace	Large amounts of overtime are not considered acceptable as the net effect is often to reduce code quality and medium term productivity
On-site customer	A representative of the end-user of the system (the customer) should be available full time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.

XP and agile principles



- ✧ Incremental development is supported through small, frequent system releases.
- ✧ Customer involvement means full-time customer engagement with the team.
- ✧ People not process through pair programming, collective ownership and a process that avoids long working hours.
- ✧ Change supported through regular system releases.
- ✧ Maintaining simplicity through constant refactoring of code.

Influential XP practices



- ✧ Extreme programming has a technical focus and is not easy to integrate with management practice in most organizations.
- ✧ Consequently, while agile development uses practices from XP, the method as originally defined is not widely used.
- ✧ Key practices
 - User stories for specification
 - Refactoring
 - Test-first development
 - Pair programming

User stories for requirements



- ✧ In XP, a customer or user is part of the XP team and is responsible for making decisions on requirements.
- ✧ User requirements are expressed as user stories or scenarios.
- ✧ These are written on cards and the development team break them down into implementation tasks. These tasks are the basis of schedule and cost estimates.
- ✧ The customer chooses the stories for inclusion in the next release based on their priorities and the schedule estimates.



A ‘prescribing medication’ story

Software Engineering
Ian Sommerville

Prescribing medication

The record of the patient must be open for input. Click on the medication field and select either ‘current medication’, ‘new medication’ or ‘formulary’.

If you select ‘current medication’, you will be asked to check the dose; If you wish to change the dose, enter the new dose then confirm the prescription.

If you choose, ‘new medication’, the system assumes that you know which medication you wish to prescribe. Type the first few letters of the drug name. You will then see a list of possible drugs starting with these letters. Choose the required medication. You will then be asked to check that the medication you have selected is correct. Enter the dose then confirm the prescription.

If you choose ‘formulary’, you will be presented with a search box for the approved formulary. Search for the drug required then select it. You will then be asked to check that the medication you have selected is correct. Enter the dose then confirm the prescription.

In all cases, the system will check that the dose is within the approved range and will ask you to change it if it is outside the range of recommended doses.

After you have confirmed the prescription, it will be displayed for checking. Either click ‘OK’ or ‘Change’. If you click ‘OK’, your prescription will be recorded on the audit database. If you click ‘Change’, you reenter the ‘Prescribing medication’ process.

Examples of task cards for prescribing medication



Task 1: Change dose of prescribed drug

Task 2: Formulary selection

Task 3: Dose checking

Dose checking is a safety precaution to check that the doctor has not prescribed a dangerously small or large dose.

Using the formulary id for the generic drug name, lookup the formulary and retrieve the recommended maximum and minimum dose.

Check the prescribed dose against the minimum and maximum. If outside the range, issue an error message saying that the dose is too high or too low. If within the range, enable the 'Confirm' button.

Refactoring



- ✧ Conventional wisdom in software engineering is to design for change. It is worth spending time and effort anticipating changes as this reduces costs later in the life cycle.
- ✧ XP, however, maintains that this is not worthwhile as changes cannot be reliably anticipated.
- ✧ Rather, it proposes constant code improvement (refactoring) to make changes easier when they have to be implemented.

Refactoring



- ✧ Programming team look for possible software improvements and make these improvements even where there is no immediate need for them.
- ✧ This improves the understandability of the software and so reduces the need for documentation.
- ✧ Changes are easier to make because the code is well-structured and clear.
- ✧ However, some changes require architecture refactoring and this is much more expensive.

Examples of refactoring



- ✧ Re-organization of a class hierarchy to remove duplicate code.
- ✧ Tidying up and renaming attributes and methods to make them easier to understand.
- ✧ The replacement of inline code with calls to methods that have been included in a program library.

Test-first development



- ✧ Testing is central to XP and XP has developed an approach where the program is tested after every change has been made.
- ✧ XP testing features:
 - Test-first development.
 - Incremental test development from scenarios.
 - User involvement in test development and validation.
 - Automated test harnesses are used to run all component tests each time that a new release is built.

Test-driven development



- ✧ Writing tests before code clarifies the requirements to be implemented.
- ✧ Tests are written as programs rather than data so that they can be executed automatically. The test includes a check that it has executed correctly.
 - Usually relies on a testing framework such as Junit.
- ✧ All previous and new tests are run automatically when new functionality is added, thus checking that the new functionality has not introduced errors.

Customer involvement



- ✧ The role of the customer in the testing process is to help develop acceptance tests for the stories that are to be implemented in the next release of the system.
- ✧ The customer who is part of the team writes tests as development proceeds. All new code is therefore validated to ensure that it is what the customer needs.
- ✧ However, people adopting the customer role have limited time available and so cannot work full-time with the development team. They may feel that providing the requirements was enough of a contribution and so may be reluctant to get involved in the testing process.

Test case description for dose checking



Test 4: Dose checking

Input:

1. A number in mg representing a single dose of the drug.
2. A number representing the number of single doses per day.

Tests:

1. Test for inputs where the single dose is correct but the frequency is too high.
2. Test for inputs where the single dose is too high and too low.
3. Test for inputs where the single dose * frequency is too high and too low.
4. Test for inputs where single dose * frequency is in the permitted range.

Output:

OK or error message indicating that the dose is outside the safe range.

Test automation



- ✧ Test automation means that tests are written as executable components before the task is implemented
 - These testing components should be stand-alone, should simulate the submission of input to be tested and should check that the result meets the output specification. An automated test framework (e.g. Junit) is a system that makes it easy to write executable tests and submit a set of tests for execution.
- ✧ As testing is automated, there is always a set of tests that can be quickly and easily executed
 - Whenever any functionality is added to the system, the tests can be run and problems that the new code has introduced can be caught immediately.

Problems with test-first development



- ✧ Programmers prefer programming to testing and sometimes they take short cuts when writing tests. For example, they may write incomplete tests that do not check for all possible exceptions that may occur.
- ✧ Some tests can be very difficult to write incrementally. For example, in a complex user interface, it is often difficult to write unit tests for the code that implements the 'display logic' and workflow between screens.
- ✧ It difficult to judge the completeness of a set of tests. Although you may have a lot of system tests, your test set may not provide complete coverage.

Pair programming



- ✧ Pair programming involves programmers working in pairs, developing code together.
- ✧ This helps develop common ownership of code and spreads knowledge across the team.
- ✧ It serves as an informal review process as each line of code is looked at by more than 1 person.
- ✧ It encourages refactoring as the whole team can benefit from improving the system code.

Pair programming



- ✧ In pair programming, programmers sit together at the same computer to develop the software.
- ✧ Pairs are created dynamically so that all team members work with each other during the development process.
- ✧ The sharing of knowledge that happens during pair programming is very important as it reduces the overall risks to a project when team members leave.
- ✧ Pair programming is not necessarily inefficient and there is some evidence that suggests that a pair working together is more efficient than 2 programmers working separately.



Agile project management

Agile project management



- ✧ The principal responsibility of software project managers is to manage the project so that the software is delivered on time and within the planned budget for the project.
- ✧ The standard approach to project management is plan-driven. Managers draw up a plan for the project showing what should be delivered, when it should be delivered and who will work on the development of the project deliverables.
- ✧ Agile project management requires a different approach, which is adapted to incremental development and the practices used in agile methods.

Scrum



- ✧ Scrum is an agile method that focuses on managing iterative development rather than specific agile practices.
- ✧ There are three phases in Scrum.
 - The initial phase is an outline planning phase where you establish the general objectives for the project and design the software architecture.
 - This is followed by a series of sprint cycles, where each cycle develops an increment of the system.
 - The project closure phase wraps up the project, completes required documentation such as system help frames and user manuals and assesses the lessons learned from the project.





Scrum terminology (a)

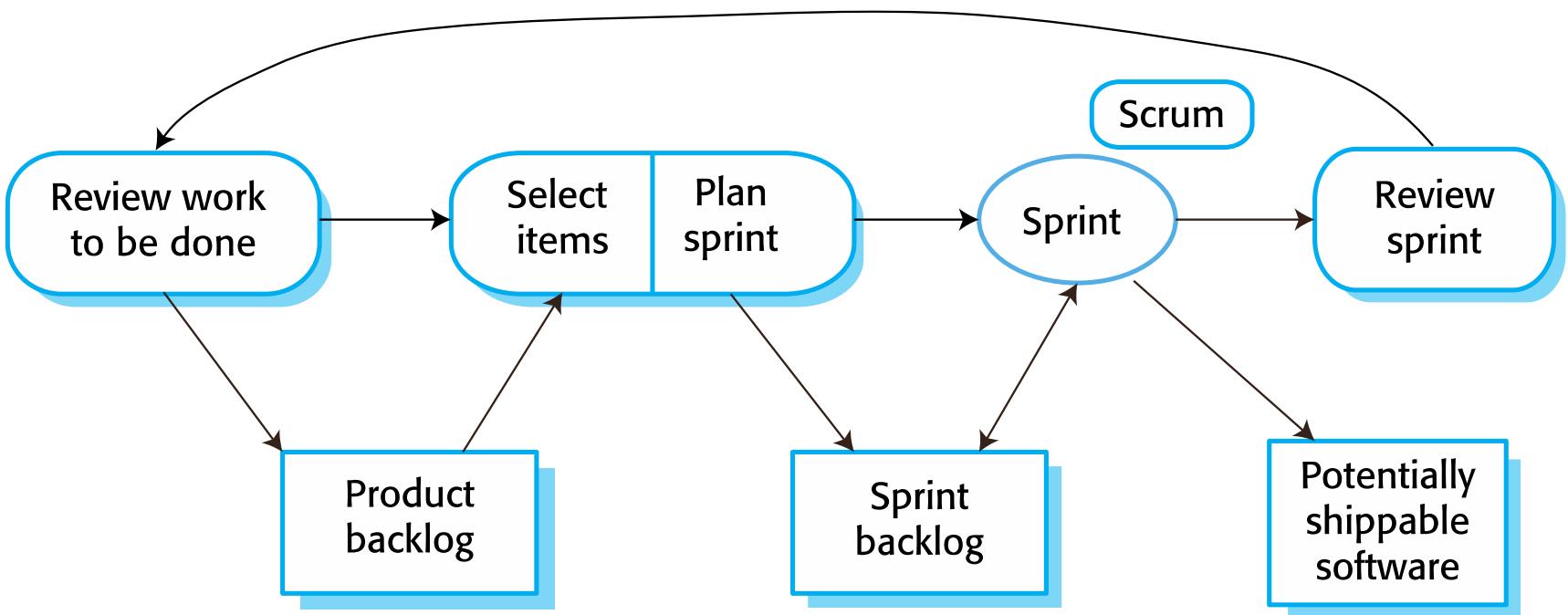
Scrum term	Definition
Development team	A self-organizing group of software developers, which should be no more than 7 people. They are responsible for developing the software and other essential project documents.
Potentially shippable product increment	The software increment that is delivered from a sprint. The idea is that this should be 'potentially shippable' which means that it is in a finished state and no further work, such as testing, is needed to incorporate it into the final product. In practice, this is not always achievable.
Product backlog	This is a list of 'to do' items which the Scrum team must tackle. They may be feature definitions for the software, software requirements, user stories or descriptions of supplementary tasks that are needed, such as architecture definition or user documentation.
Product owner	An individual (or possibly a small group) whose job is to identify product features or requirements, prioritize these for development and continuously review the product backlog to ensure that the project continues to meet critical business needs. The Product Owner can be a customer but might also be a product manager in a software company or other stakeholder representative.



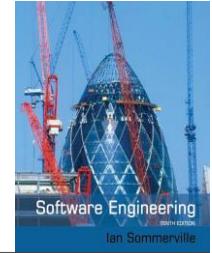
Scrum terminology (b)

Scrum term	Definition
Scrum	A daily meeting of the Scrum team that reviews progress and prioritizes work to be done that day. Ideally, this should be a short face-to-face meeting that includes the whole team.
ScrumMaster	The ScrumMaster is responsible for ensuring that the Scrum process is followed and guides the team in the effective use of Scrum. He or she is responsible for interfacing with the rest of the company and for ensuring that the Scrum team is not diverted by outside interference. The Scrum developers are adamant that the ScrumMaster should not be thought of as a project manager. Others, however, may not always find it easy to see the difference.
Sprint	A development iteration. Sprints are usually 2-4 weeks long.
Velocity	An estimate of how much product backlog effort that a team can cover in a single sprint. Understanding a team's velocity helps them estimate what can be covered in a sprint and provides a basis for measuring improving performance.

Scrum sprint cycle



The Scrum sprint cycle



- ✧ Sprints are fixed length, normally 2–4 weeks.
- ✧ The starting point for planning is the product backlog, which is the list of work to be done on the project.
- ✧ The selection phase involves all of the project team who work with the customer to select the features and functionality from the product backlog to be developed during the sprint.

The Sprint cycle



- ✧ Once these are agreed, the team organize themselves to develop the software.
- ✧ During this stage the team is isolated from the customer and the organization, with all communications channelled through the so-called 'Scrum master'.
- ✧ The role of the Scrum master is to protect the development team from external distractions.
- ✧ At the end of the sprint, the work done is reviewed and presented to stakeholders. The next sprint cycle then begins.

Teamwork in Scrum



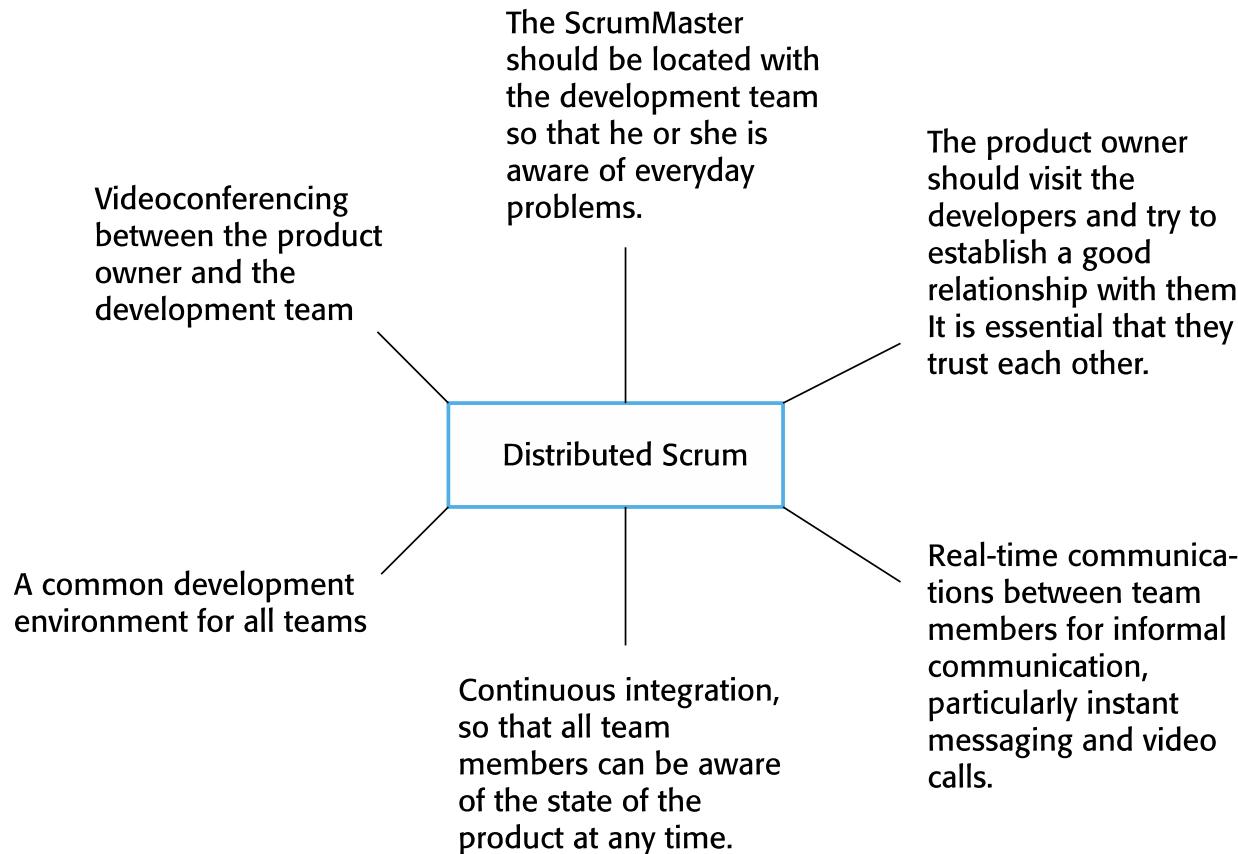
- ✧ The ‘Scrum master’ is a facilitator who arranges daily meetings, tracks the backlog of work to be done, records decisions, measures progress against the backlog and communicates with customers and management outside of the team.
- ✧ The whole team attends short daily meetings (Scrums) where all team members share information, describe their progress since the last meeting, problems that have arisen and what is planned for the following day.
 - This means that everyone on the team knows what is going on and, if problems arise, can re-plan short-term work to cope with them.



Scrum benefits

- ✧ The product is broken down into a set of manageable and understandable chunks.
- ✧ Unstable requirements do not hold up progress.
- ✧ The whole team have visibility of everything and consequently team communication is improved.
- ✧ Customers see on-time delivery of increments and gain feedback on how the product works.
- ✧ Trust between customers and developers is established and a positive culture is created in which everyone expects the project to succeed.

Distributed Scrum





Scaling agile methods

Scaling agile methods



- ✧ Agile methods have proved to be successful for small and medium sized projects that can be developed by a small co-located team.
- ✧ It is sometimes argued that the success of these methods comes because of improved communications which is possible when everyone is working together.
- ✧ Scaling up agile methods involves changing these to cope with larger, longer projects where there are multiple development teams, perhaps working in different locations.

Scaling out and scaling up

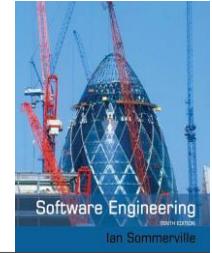


- ✧ ‘Scaling up’ is concerned with using agile methods for developing large software systems that cannot be developed by a small team.
- ✧ ‘Scaling out’ is concerned with how agile methods can be introduced across a large organization with many years of software development experience.
- ✧ When scaling agile methods it is important to maintain agile fundamentals:
 - Flexible planning, frequent system releases, continuous integration, test-driven development and good team communications.

Practical problems with agile methods



- ✧ The informality of agile development is incompatible with the legal approach to contract definition that is commonly used in large companies.
- ✧ Agile methods are most appropriate for new software development rather than software maintenance. Yet the majority of software costs in large companies come from maintaining their existing software systems.
- ✧ Agile methods are designed for small co-located teams yet much software development now involves worldwide distributed teams.



Contractual issues

- ✧ Most software contracts for custom systems are based around a specification, which sets out what has to be implemented by the system developer for the system customer.
- ✧ However, this precludes interleaving specification and development as is the norm in agile development.
- ✧ A contract that pays for developer time rather than functionality is required.
 - However, this is seen as a high risk my many legal departments because what has to be delivered cannot be guaranteed.

Agile methods and software maintenance



- ✧ Most organizations spend more on maintaining existing software than they do on new software development. So, if agile methods are to be successful, they have to support maintenance as well as original development.
- ✧ Two key issues:
 - Are systems that are developed using an agile approach maintainable, given the emphasis in the development process of minimizing formal documentation?
 - Can agile methods be used effectively for evolving a system in response to customer change requests?
- ✧ Problems may arise if original development team cannot be maintained.

Agile maintenance



- ✧ Key problems are:
 - Lack of product documentation
 - Keeping customers involved in the development process
 - Maintaining the continuity of the development team
- ✧ Agile development relies on the development team knowing and understanding what has to be done.
- ✧ For long-lifetime systems, this is a real problem as the original developers will not always work on the system.

Agile and plan-driven methods



- ✧ Most projects include elements of plan-driven and agile processes. Deciding on the balance depends on:
 - Is it important to have a very detailed specification and design before moving to implementation? If so, you probably need to use a plan-driven approach.
 - Is an incremental delivery strategy, where you deliver the software to customers and get rapid feedback from them, realistic? If so, consider using agile methods.
 - How large is the system that is being developed? Agile methods are most effective when the system can be developed with a small co-located team who can communicate informally. This may not be possible for large systems that require larger development teams so a plan-driven approach may have to be used.

Agile principles and organizational practice



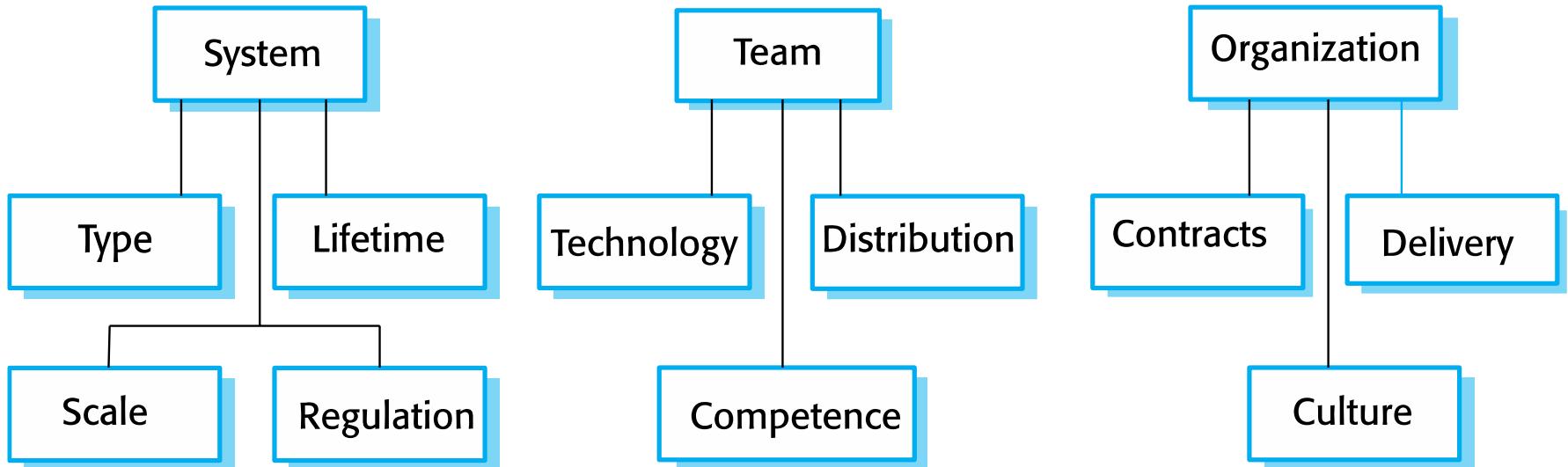
Principle	Practice
Customer involvement	<p>This depends on having a customer who is willing and able to spend time with the development team and who can represent all system stakeholders. Often, customer representatives have other demands on their time and cannot play a full part in the software development.</p> <p>Where there are external stakeholders, such as regulators, it is difficult to represent their views to the agile team.</p>
Embrace change	<p>Prioritizing changes can be extremely difficult, especially in systems for which there are many stakeholders. Typically, each stakeholder gives different priorities to different changes.</p>
Incremental delivery	<p>Rapid iterations and short-term planning for development does not always fit in with the longer-term planning cycles of business planning and marketing. Marketing managers may need to know what product features several months in advance to prepare an effective marketing campaign.</p>

Agile principles and organizational practice



Principle	Practice
Maintain simplicity	Under pressure from delivery schedules, team members may not have time to carry out desirable system simplifications.
People not process	Individual team members may not have suitable personalities for the intense involvement that is typical of agile methods, and therefore may not interact well with other team members.

Agile and plan-based factors





System issues

- ✧ How large is the system being developed?
 - Agile methods are most effective a relatively small co-located team who can communicate informally.
- ✧ What type of system is being developed?
 - Systems that require a lot of analysis before implementation need a fairly detailed design to carry out this analysis.
- ✧ What is the expected system lifetime?
 - Long-lifetime systems require documentation to communicate the intentions of the system developers to the support team.
- ✧ Is the system subject to external regulation?
 - If a system is regulated you will probably be required to produce detailed documentation as part of the system safety case.

People and teams



- ✧ How good are the designers and programmers in the development team?
 - It is sometimes argued that agile methods require higher skill levels than plan-based approaches in which programmers simply translate a detailed design into code.
- ✧ How is the development team organized?
 - Design documents may be required if the team is distributed.
- ✧ What support technologies are available?
 - IDE support for visualisation and program analysis is essential if design documentation is not available.



Organizational issues

- ✧ Traditional engineering organizations have a culture of plan-based development, as this is the norm in engineering.
- ✧ Is it standard organizational practice to develop a detailed system specification?
- ✧ Will customer representatives be available to provide feedback of system increments?
- ✧ Can informal agile development fit into the organizational culture of detailed documentation?

Agile methods for large systems



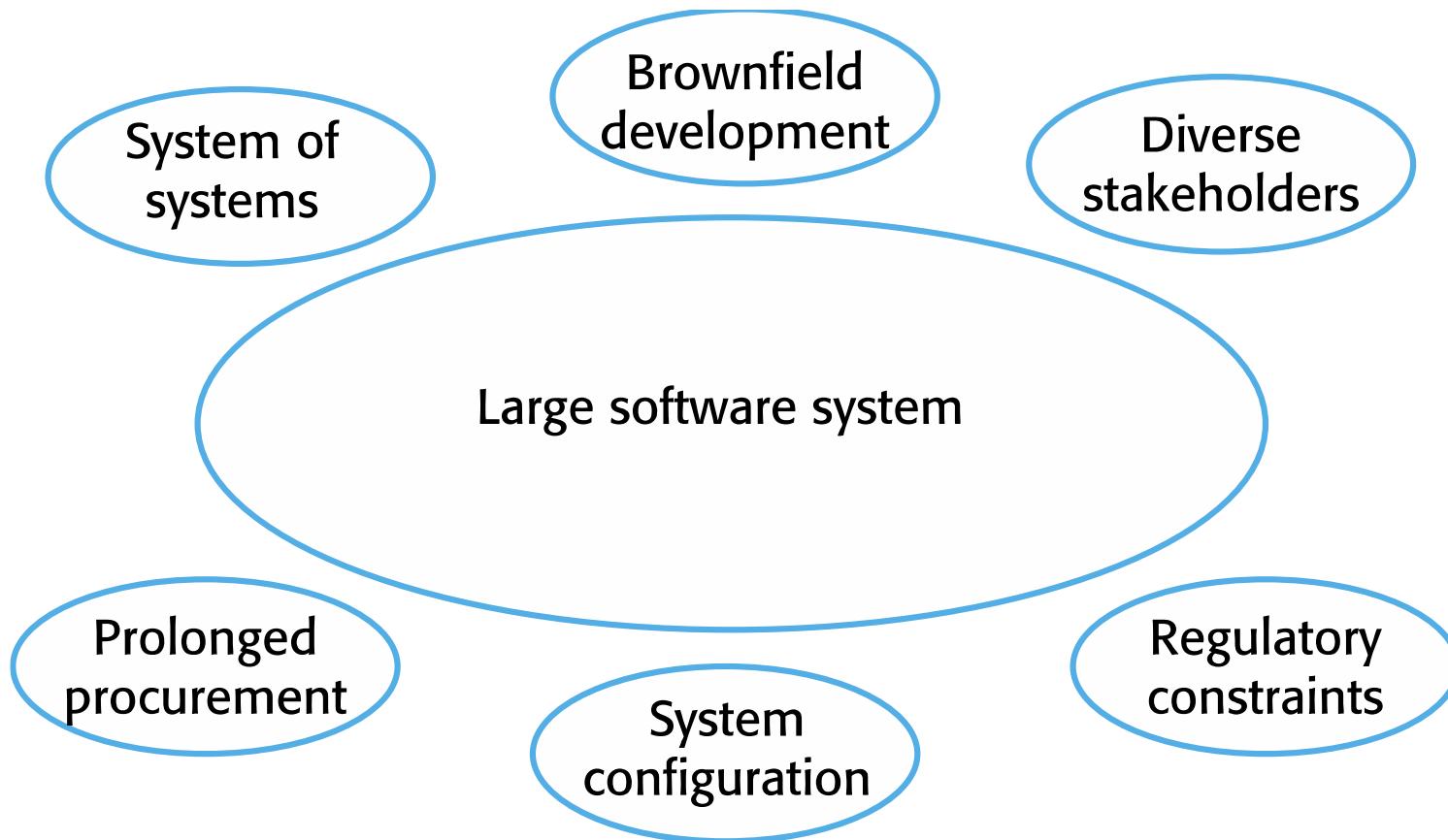
- ✧ Large systems are usually collections of separate, communicating systems, where separate teams develop each system. Frequently, these teams are working in different places, sometimes in different time zones.
- ✧ Large systems are ‘brownfield systems’, that is they include and interact with a number of existing systems. Many of the system requirements are concerned with this interaction and so don’t really lend themselves to flexibility and incremental development.
- ✧ Where several systems are integrated to create a system, a significant fraction of the development is concerned with system configuration rather than original code development.

Large system development

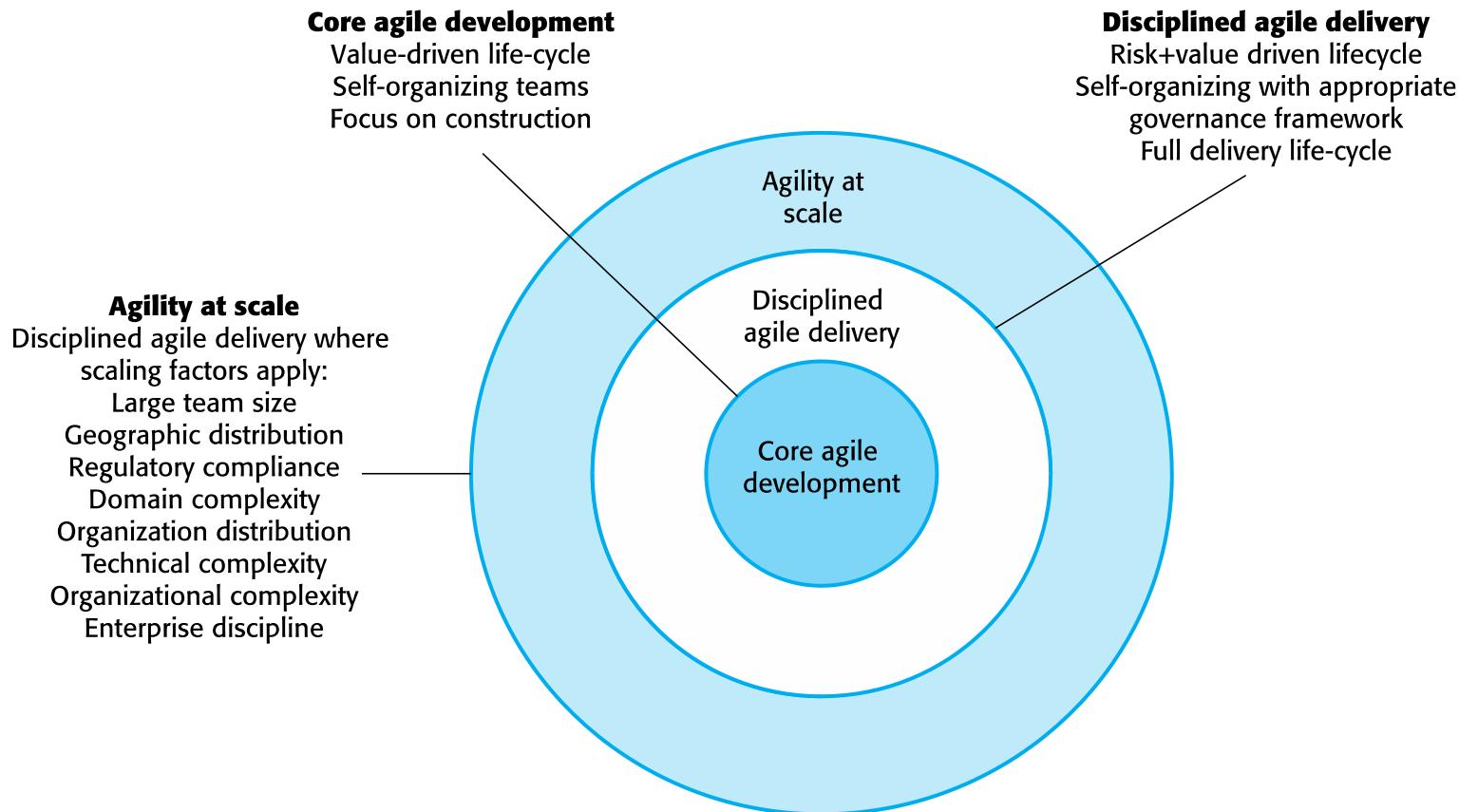


- ✧ Large systems and their development processes are often constrained by external rules and regulations limiting the way that they can be developed.
- ✧ Large systems have a long procurement and development time. It is difficult to maintain coherent teams who know about the system over that period as, inevitably, people move on to other jobs and projects.
- ✧ Large systems usually have a diverse set of stakeholders. It is practically impossible to involve all of these different stakeholders in the development process.

Factors in large systems



IBM's agility at scale model



Scaling up to large systems



- ✧ A completely incremental approach to requirements engineering is impossible.
- ✧ There cannot be a single product owner or customer representative.
- ✧ For large systems development, it is not possible to focus only on the code of the system.
- ✧ Cross-team communication mechanisms have to be designed and used.
- ✧ Continuous integration is practically impossible. However, it is essential to maintain frequent system builds and regular releases of the system.



Multi-team Scrum

✧ *Role replication*

- Each team has a Product Owner for their work component and ScrumMaster.

✧ *Product architects*

- Each team chooses a product architect and these architects collaborate to design and evolve the overall system architecture.

✧ *Release alignment*

- The dates of product releases from each team are aligned so that a demonstrable and complete system is produced.

✧ *Scrum of Scrums*

- There is a daily Scrum of Scrums where representatives from each team meet to discuss progress and plan work to be done.

Agile methods across organizations



- ✧ Project managers who do not have experience of agile methods may be reluctant to accept the risk of a new approach.
- ✧ Large organizations often have quality procedures and standards that all projects are expected to follow and, because of their bureaucratic nature, these are likely to be incompatible with agile methods.
- ✧ Agile methods seem to work best when team members have a relatively high skill level. However, within large organizations, there are likely to be a wide range of skills and abilities.
- ✧ There may be cultural resistance to agile methods, especially in those organizations that have a long history of using conventional systems engineering processes.



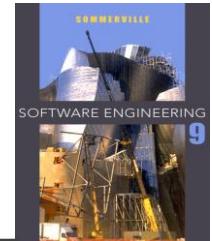
Key points

- ✧ Agile methods are incremental development methods that focus on rapid software development, frequent releases of the software, reducing process overheads by minimizing documentation and producing high-quality code.
- ✧ Agile development practices include
 - User stories for system specification
 - Frequent releases of the software,
 - Continuous software improvement
 - Test-first development
 - Customer participation in the development team.

Key points

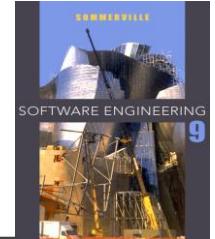


- ✧ Scrum is an agile method that provides a project management framework.
 - It is centred round a set of sprints, which are fixed time periods when a system increment is developed.
- ✧ Many practical development methods are a mixture of plan-based and agile development.
- ✧ Scaling agile methods for large systems is difficult.
 - Large systems need up-front design and some documentation and organizational practice may conflict with the informality of agile approaches.



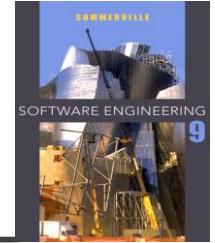
Chapter 4 – Requirements Engineering

Lecture 1



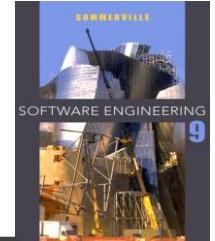
Topics covered

- ✧ Functional and non-functional requirements
- ✧ The software requirements document
- ✧ Requirements specification
- ✧ Requirements engineering processes
- ✧ Requirements elicitation and analysis
- ✧ Requirements validation
- ✧ Requirements management



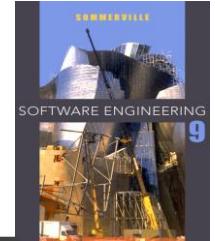
Requirements engineering

- ✧ The process of establishing the services that the customer requires from a system and the constraints under which it operates and is developed.
- ✧ The requirements themselves are the descriptions of the system services and constraints that are generated during the requirements engineering process.



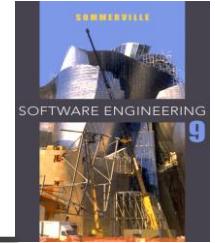
What is a requirement?

- ✧ It may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification.
- ✧ This is inevitable as requirements may serve a dual function
 - May be the basis for a bid for a contract - therefore must be open to interpretation;
 - May be the basis for the contract itself - therefore must be defined in detail;
 - Both these statements may be called requirements.



Requirements abstraction (Davis)

“If a company wishes to let a contract for a large software development project, it must define its needs in a sufficiently abstract way that a solution is not pre-defined. The requirements must be written so that several contractors can bid for the contract, offering, perhaps, different ways of meeting the client organization’s needs. Once a contract has been awarded, the contractor must write a system definition for the client in more detail so that the client understands and can validate what the software will do. Both of these documents may be called the requirements document for the system.”



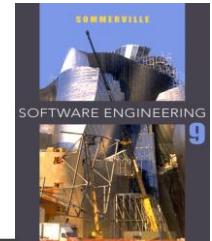
Types of requirement

✧ User requirements

- Statements in natural language plus diagrams of the services the system provides and its operational constraints. Written for customers.

✧ System requirements

- A structured document setting out detailed descriptions of the system's functions, services and operational constraints. Defines what should be implemented so may be part of a contract between client and contractor.



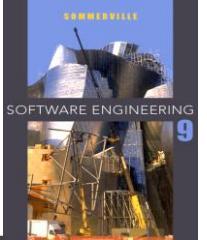
User and system requirements

User requirement definition

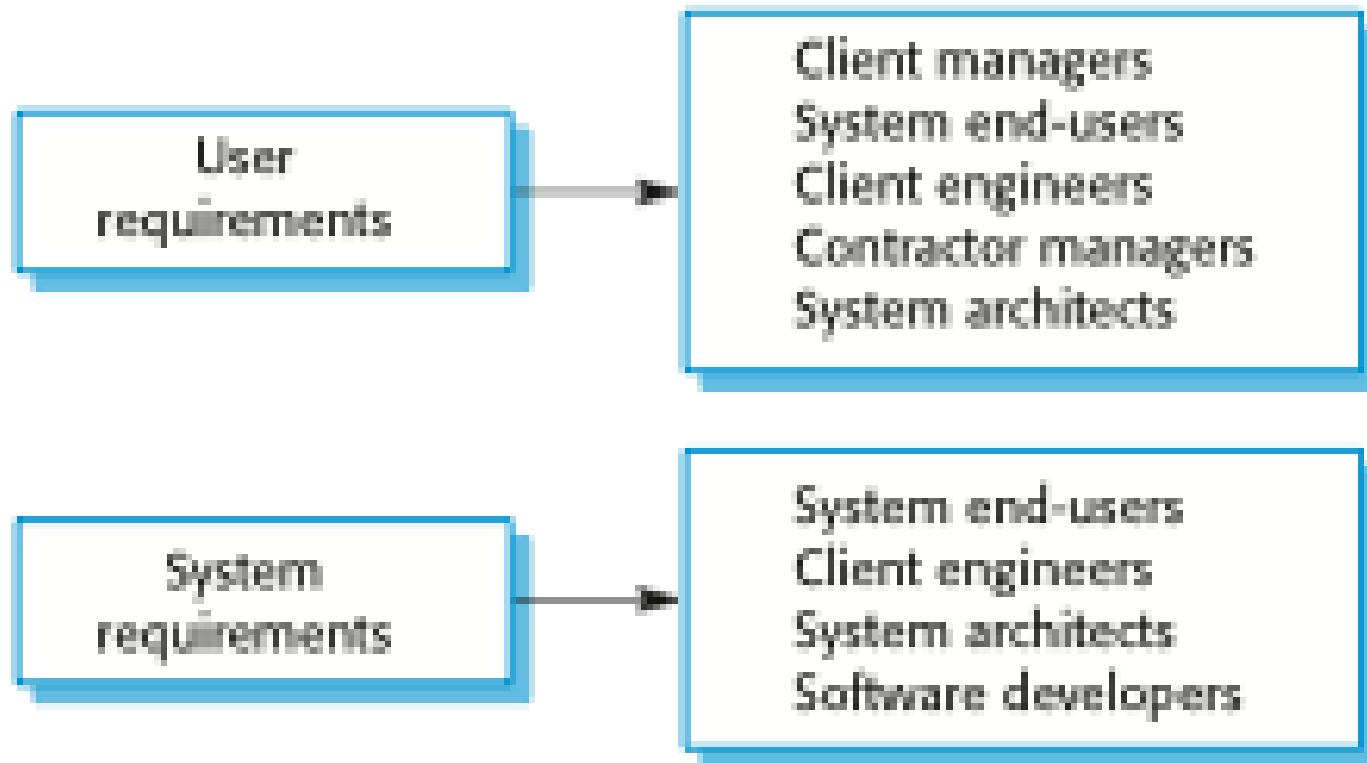
1. The MHC-PMS shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.

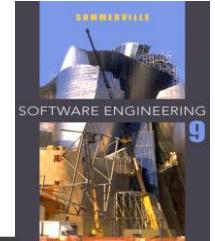
System requirements specification

- 1.1 On the last working day of each month, a summary of the drugs prescribed, their cost and the prescribing clinics shall be generated.
- 1.2 The system shall automatically generate the report for printing after 17:30 on the last working day of the month.
- 1.3 A report shall be created for each clinic and shall list the individual drug names, the total number of prescriptions, the number of doses prescribed and the total cost of the prescribed drugs.
- 1.4 If drugs are available in different dose units (e.g. 10mg, 20 mg, etc.) separate reports shall be created for each dose unit.
- 1.5 Access to all cost reports shall be restricted to authorized users listed on a management access control list.



Readers of different types of requirements specification





Functional and non-functional requirements

✧ Functional requirements

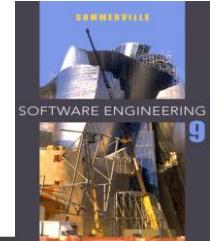
- Statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.
- May state what the system should not do.

✧ Non-functional requirements

- Constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.
- Often apply to the system as a whole rather than individual features or services.

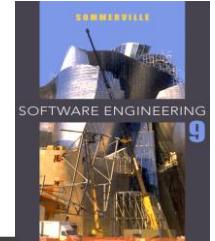
✧ Domain requirements

- Constraints on the system from the domain of operation



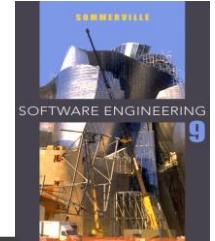
Functional requirements

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



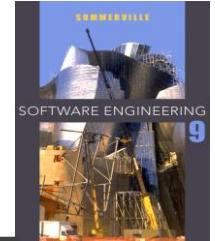
Functional requirements for the MHC-PMS

- ✧ A user shall be able to search the appointments lists for all clinics.
- ✧ The system shall generate each day, for each clinic, a list of patients who are expected to attend appointments that day.
- ✧ Each staff member using the system shall be uniquely identified by his or her 8-digit employee number.



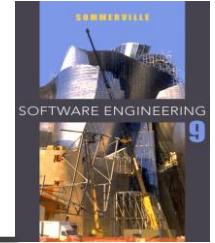
Requirements imprecision

- ✧ Problems arise when requirements are not precisely stated.
- ✧ Ambiguous requirements may be interpreted in different ways by developers and users.
- ✧ Consider the term ‘search’ in requirement 1
 - User intention – search for a patient name across all appointments in all clinics;
 - Developer interpretation – search for a patient name in an individual clinic. User chooses clinic then search.



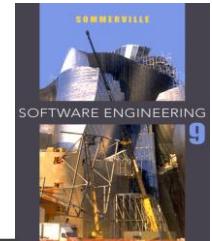
Requirements completeness and consistency

- ✧ 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.

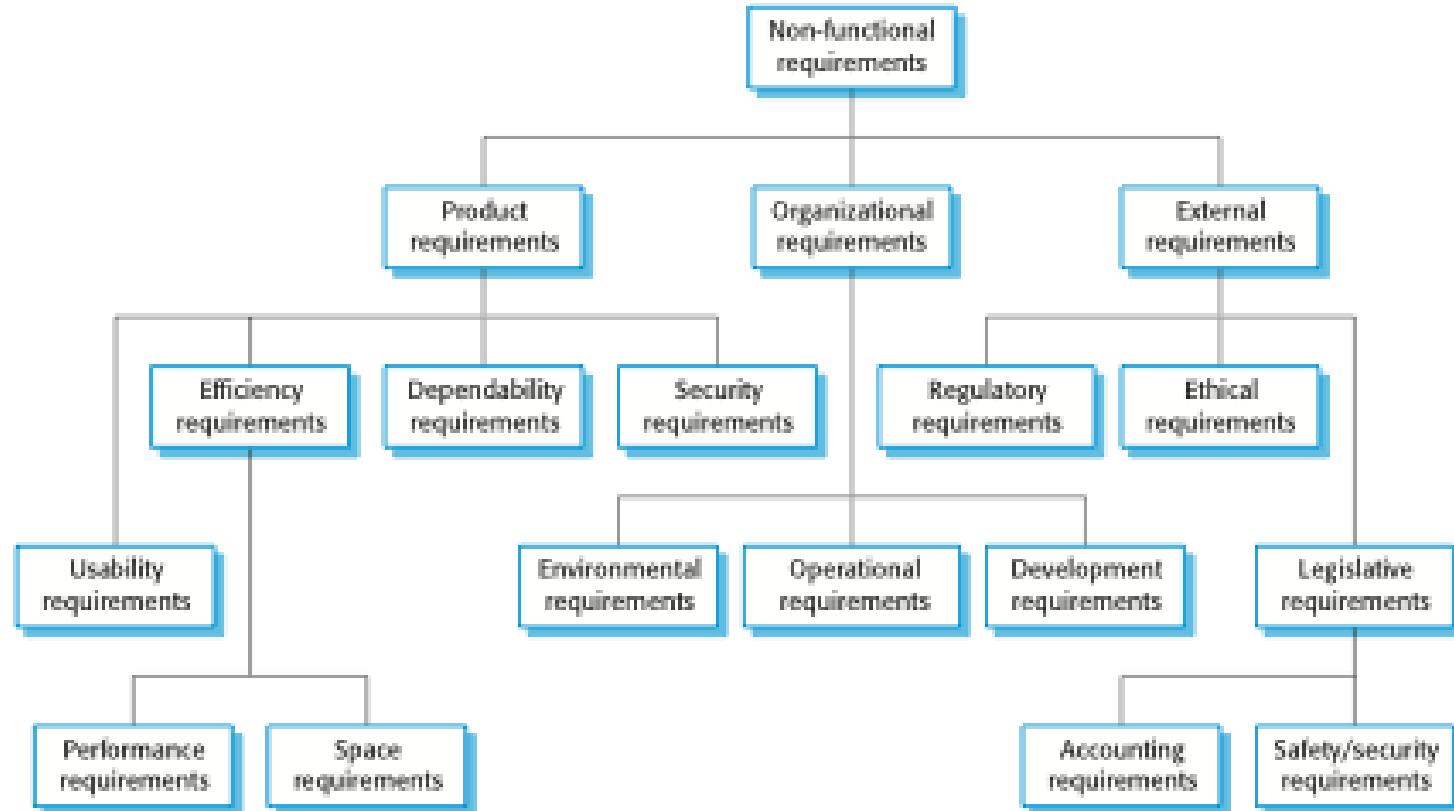


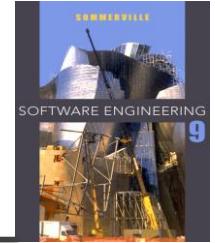
Non-functional requirements

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



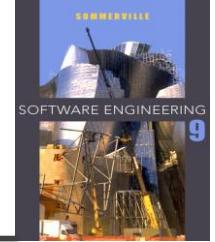
Types of nonfunctional requirement





Non-functional requirements implementation

- ✧ Non-functional requirements may affect the overall architecture of a system rather than the individual components.
 - For example, to ensure that performance requirements are met, you may have to organize the system to minimize communications between components.
- ✧ A single non-functional requirement, such as a security requirement, may generate a number of related functional requirements that define system services that are required.
 - It may also generate requirements that restrict existing requirements.



Non-functional classifications

✧ Product requirements

- Requirements which specify that the delivered product must behave in a particular way e.g. execution speed, reliability, etc.

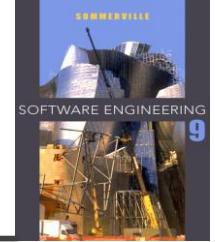
✧ Organisational requirements

- Requirements which are a consequence of organisational policies and procedures e.g. process standards used, implementation requirements, etc.

✧ External requirements

- Requirements which arise from factors which are external to the system and its development process e.g. interoperability requirements, legislative requirements, etc.

Examples of nonfunctional requirements in the MHC-PMS



Product requirement

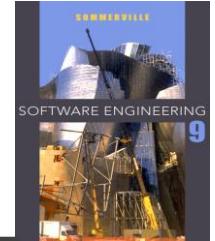
The MHC-PMS shall be available to all clinics during normal working hours (Mon–Fri, 0830–17.30). Downtime within normal working hours shall not exceed five seconds in any one day.

Organizational requirement

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

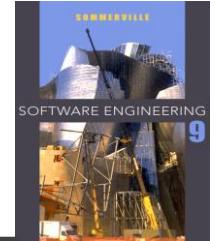
External requirement

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



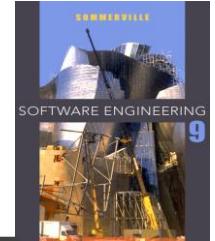
Goals and requirements

- ✧ Non-functional requirements may be very difficult to state precisely and imprecise requirements may be difficult to verify.
- ✧ Goal
 - A general intention of the user such as ease of use.
- ✧ Verifiable non-functional requirement
 - A statement using some measure that can be objectively tested.
- ✧ Goals are helpful to developers as they convey the intentions of the system users.



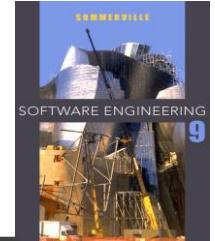
Usability requirements

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



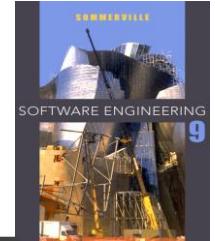
Metrics for specifying nonfunctional requirements

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



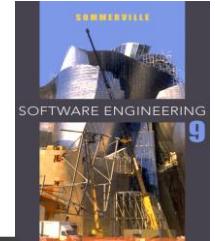
Domain requirements

- ✧ The system's operational domain imposes requirements on the system.
 - For example, a train control system has to take into account the braking characteristics in different weather conditions.
- ✧ Domain requirements be new functional requirements, constraints on existing requirements or define specific computations.
- ✧ If domain requirements are not satisfied, the system may be unworkable.



Train protection system

- ✧ This is a domain requirement for a train protection system:
- ✧ The deceleration of the train shall be computed as:
 - $D_{train} = D_{control} + D_{gradient}$
 - where $D_{gradient}$ is $9.81\text{ms}^2 * \text{compensated gradient}/\alpha$ and where the values of $9.81\text{ms}^2 / \alpha$ are known for different types of train.
- ✧ It is difficult for a non-specialist to understand the implications of this and how it interacts with other requirements.



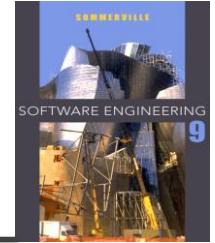
Domain requirements problems

✧ Understandability

- Requirements are expressed in the language of the application domain;
- This is often not understood by software engineers developing the system.

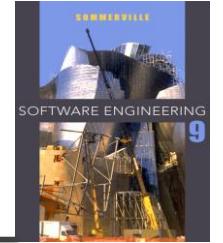
✧ Implicitness

- Domain specialists understand the area so well that they do not think of making the domain requirements explicit.



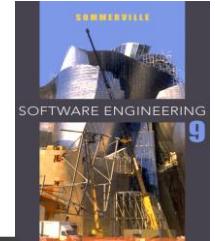
Key points

- ✧ Requirements for a software system set out what the system should do and define constraints on its operation and implementation.
- ✧ Functional requirements are statements of the services that the system must provide or are descriptions of how some computations must be carried out.
- ✧ Non-functional requirements often constrain the system being developed and the development process being used.
- ✧ They often relate to the emergent properties of the system and therefore apply to the system as a whole.



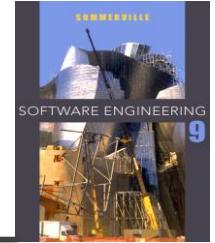
Chapter 4 – Requirements Engineering

Lecture 2



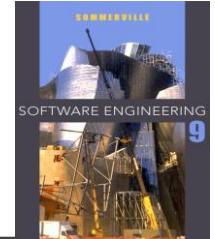
The software requirements document

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

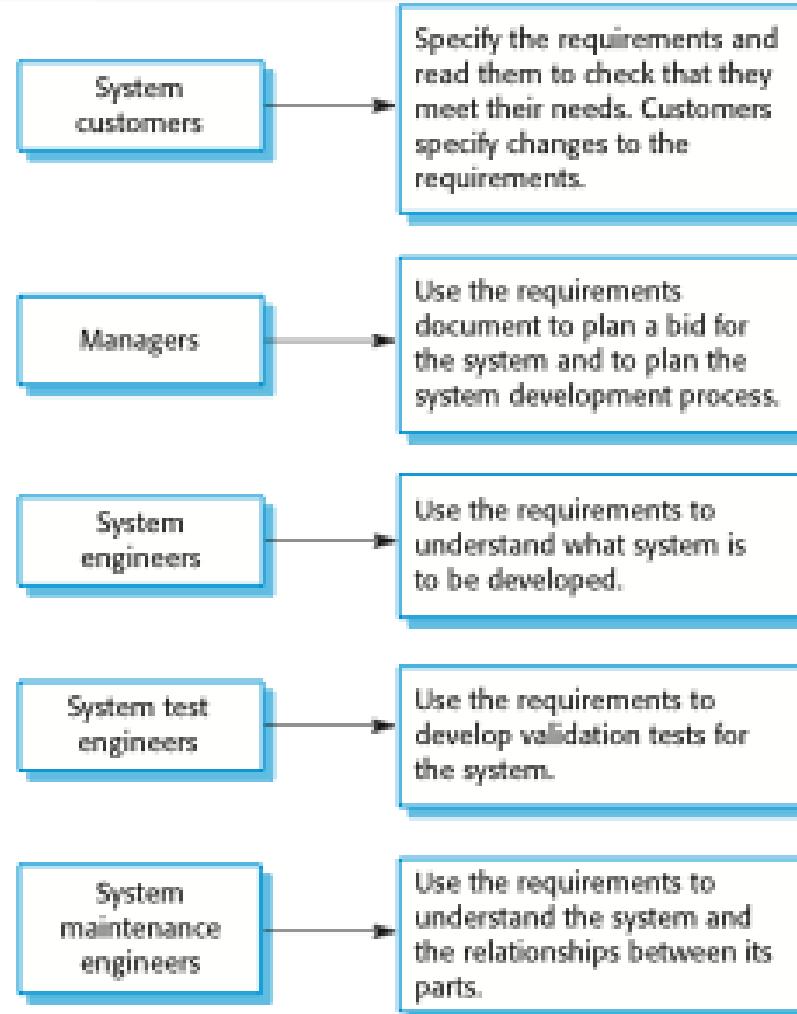


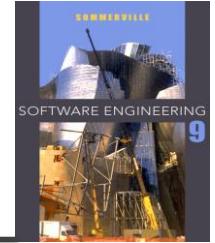
Agile methods and requirements

- ✧ Many agile methods argue that producing a requirements document is a waste of time as requirements change so quickly.
- ✧ The document is therefore always out of date.
- ✧ Methods such as XP use incremental requirements engineering and express requirements as ‘user stories’ (discussed in Chapter 3).
- ✧ This is practical for business systems but problematic for systems that require a lot of pre-delivery analysis (e.g. critical systems) or systems developed by several teams.



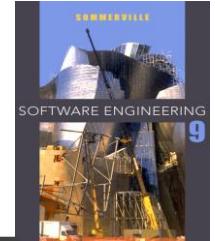
Users of a requirements document





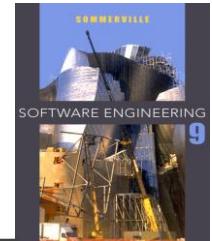
Requirements document variability

- ✧ Information in requirements document depends on type of system and the approach to development used.
- ✧ Systems developed incrementally will, typically, have less detail in the requirements document.
- ✧ Requirements documents standards have been designed e.g. IEEE standard. These are mostly applicable to the requirements for large systems engineering projects.



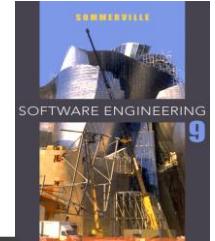
The structure of a requirements document

Chapter	Description
Preface	This should define the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version.
Introduction	This should describe the need for the system. It should briefly describe the system's functions and explain how it will work with other systems. It should also describe how the system fits into the overall business or strategic objectives of the organization commissioning the software.
Glossary	This should define the technical terms used in the document. You should not make assumptions about the experience or expertise of the reader.
User requirements definition	Here, you describe the services provided for the user. The nonfunctional system requirements should also be described in this section. This description may use natural language, diagrams, or other notations that are understandable to customers. Product and process standards that must be followed should be specified.
System architecture	This chapter should present a high-level overview of the anticipated system architecture, showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.



The structure of a requirements document

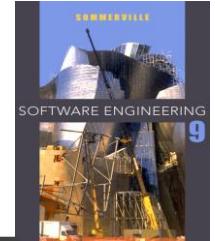
Chapter	Description
System requirements specification	This should describe the functional and nonfunctional requirements in more detail. If necessary, further detail may also be added to the nonfunctional requirements. Interfaces to other systems may be defined.
System models	This might include graphical system models showing the relationships between the system components and the system and its environment. Examples of possible models are object models, data-flow models, or semantic data models.
System evolution	This should describe the fundamental assumptions on which the system is based, and any anticipated changes due to hardware evolution, changing user needs, and so on. This section is useful for system designers as it may help them avoid design decisions that would constrain likely future changes to the system.
Appendices	These should provide detailed, specific information that is related to the application being developed; for example, hardware and database descriptions. Hardware requirements define the minimal and optimal configurations for the system. Database requirements define the logical organization of the data used by the system and the relationships between data.
Index	Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, and so on.



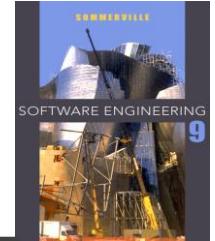
Requirements specification

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

Ways of writing a system requirements specification

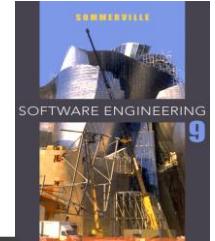


Notation	Description
Natural language	The requirements are written using numbered sentences in natural language. Each sentence should express one requirement.
Structured natural language	The requirements are written in natural language on a standard form or template. Each field provides information about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



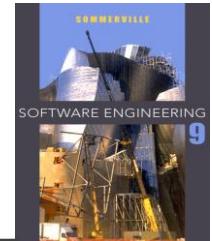
Requirements and design

- ✧ In principle, requirements should state what the system should do and the design should describe how it does this.
- ✧ In practice, requirements and design are inseparable
 - A system architecture may be designed to structure the requirements;
 - The system may inter-operate with other systems that generate design requirements;
 - The use of a specific architecture to satisfy non-functional requirements may be a domain requirement.
 - This may be the consequence of a regulatory requirement.



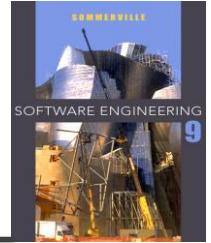
Natural language specification

- ✧ Requirements are written as natural language sentences supplemented by diagrams and tables.
- ✧ Used for writing requirements because it is expressive, intuitive and universal. This means that the requirements can be understood by users and customers.



Guidelines for writing requirements

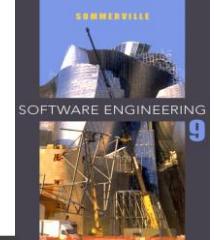
- ✧ Invent a standard format and use it for all requirements.
- ✧ Use language in a consistent way. Use shall for mandatory requirements, should for desirable requirements.
- ✧ Use text highlighting to identify key parts of the requirement.
- ✧ Avoid the use of computer jargon.
- ✧ Include an explanation (rationale) of why a requirement is necessary.



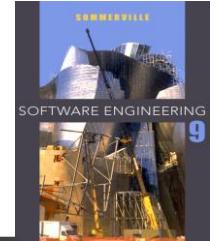
Problems with natural language

- ✧ Lack of clarity
 - Precision is difficult without making the document difficult to read.
- ✧ Requirements confusion
 - Functional and non-functional requirements tend to be mixed-up.
- ✧ Requirements amalgamation
 - Several different requirements may be expressed together.

Example requirements for the insulin pump software system

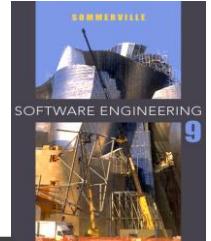


- 3.2 The system shall measure the blood sugar and deliver insulin, if required, every 10 minutes. (*Changes in blood sugar are relatively slow so more frequent measurement is unnecessary; less frequent measurement could lead to unnecessarily high sugar levels.*)
- 3.6 The system shall run a self-test routine every minute with the conditions to be tested and the associated actions defined in Table 1. (*A self-test routine can discover hardware and software problems and alert the user to the fact the normal operation may be impossible.*)



Structured specifications

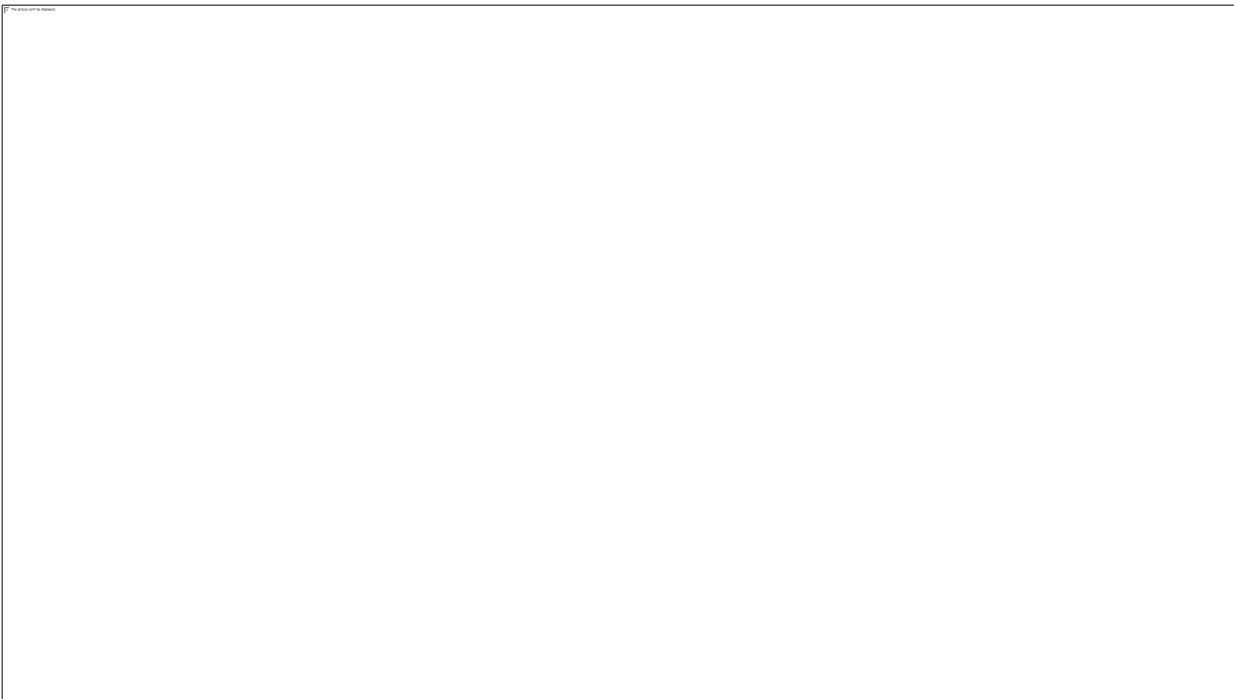
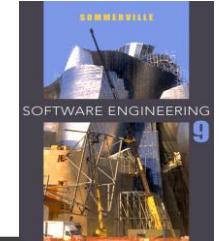
- ✧ An approach to writing requirements where the freedom of the requirements writer is limited and requirements are written in a standard way.
- ✧ This works well for some types of requirements e.g. requirements for embedded control system but is sometimes too rigid for writing business system requirements.



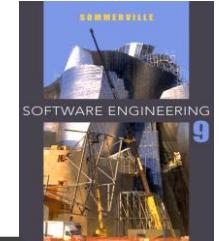
Form-based specifications

- ✧ Definition of the function or entity.
- ✧ Description of inputs and where they come from.
- ✧ Description of outputs and where they go to.
- ✧ Information about the information needed for the computation and other entities used.
- ✧ Description of the action to be taken.
- ✧ Pre and post conditions (if appropriate).
- ✧ The side effects (if any) of the function.

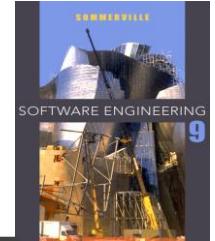
A structured specification of a requirement for an insulin pump



A structured specification of a requirement for an insulin pump

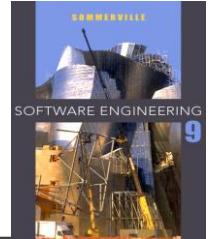


[No document is present.]



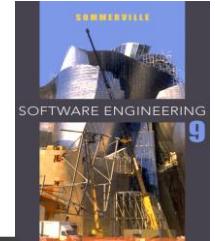
Tabular specification

- ✧ Used to supplement natural language.
- ✧ Particularly useful when you have to define a number of possible alternative courses of action.
- ✧ For example, the insulin pump systems bases its computations on the rate of change of blood sugar level and the tabular specification explains how to calculate the insulin requirement for different scenarios.



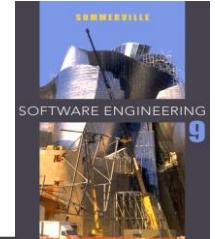
Tabular specification of computation for an insulin pump

Condition	Action
Sugar level falling ($r_2 < r_1$)	$\text{CompDose} = 0$
Sugar level stable ($r_2 = r_1$)	$\text{CompDose} = 0$
Sugar level increasing and rate of increase decreasing $((r_2 - r_1) < (r_1 - r_0))$	$\text{CompDose} = 0$
Sugar level increasing and rate of increase stable or increasing $((r_2 - r_1) \geq (r_1 - r_0))$	$\text{CompDose} = \text{round}((r_2 - r_1)/4)$ If rounded result = 0 then $\text{CompDose} = \text{MinimumDose}$

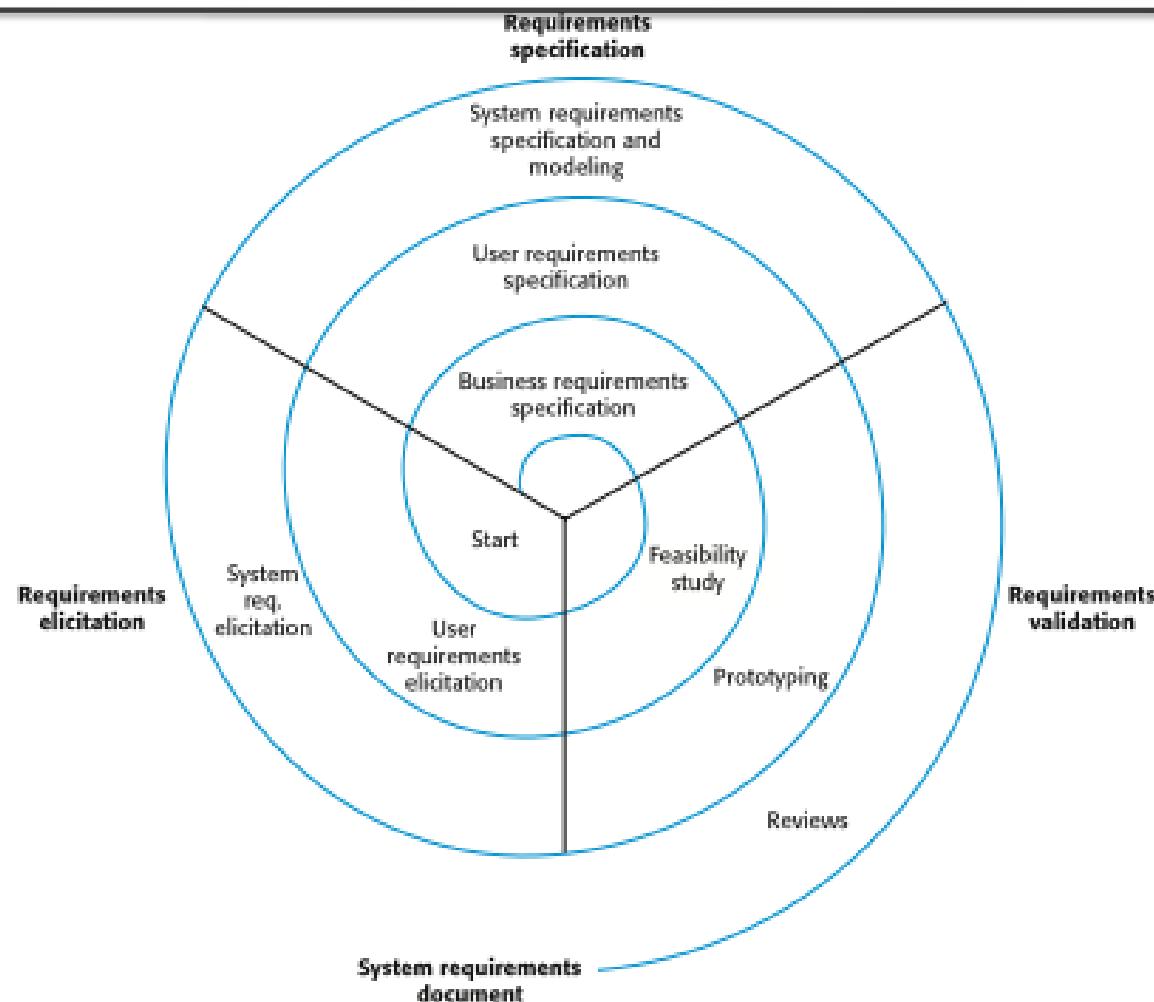


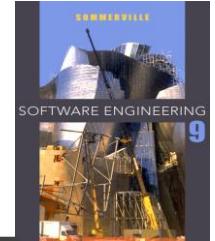
Requirements engineering processes

- ✧ The processes used for RE vary widely depending on the application domain, the people involved and the organisation developing the requirements.
- ✧ However, there are a number of generic activities common to all processes
 - Requirements elicitation;
 - Requirements analysis;
 - Requirements validation;
 - Requirements management.
- ✧ In practice, RE is an iterative activity in which these processes are interleaved.



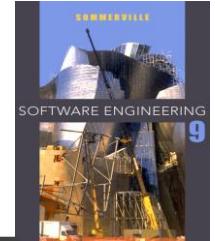
A spiral view of the requirements engineering process





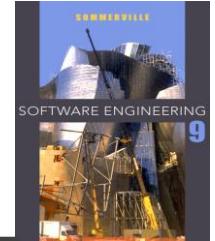
Requirements elicitation and analysis

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



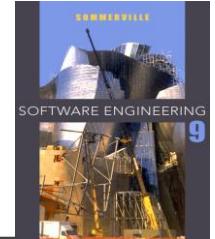
Problems of requirements analysis

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

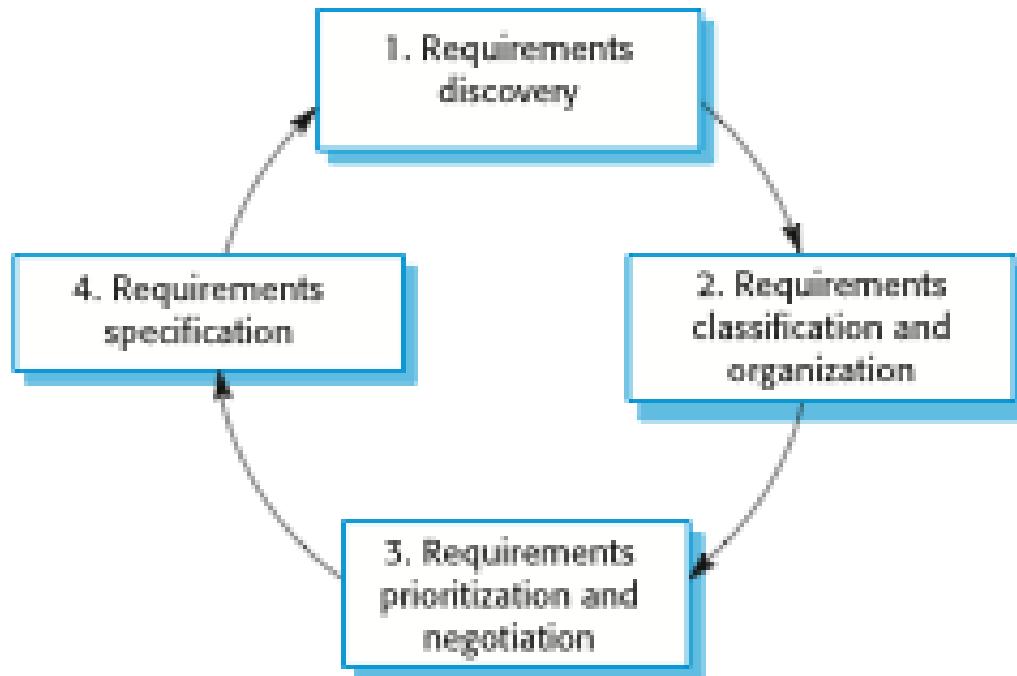


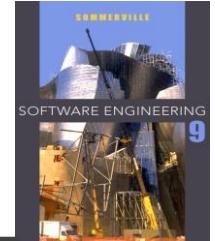
Requirements elicitation and analysis

- ✧ Software engineers work with a range of system stakeholders to find out about the application domain, the services that the system should provide, the required system performance, hardware constraints, other systems, etc.
- ✧ Stages include:
 - Requirements discovery,
 - Requirements classification and organization,
 - Requirements prioritization and negotiation,
 - Requirements specification.



The requirements elicitation and analysis process





Process activities

✧ Requirements discovery

- Interacting with stakeholders to discover their requirements.
Domain requirements are also discovered at this stage.

✧ Requirements classification and organisation

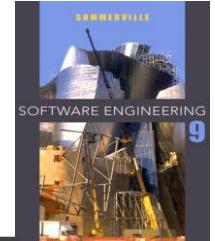
- Groups related requirements and organises them into coherent clusters.

✧ Prioritisation and negotiation

- Prioritising requirements and resolving requirements conflicts.

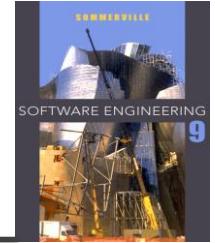
✧ Requirements specification

- Requirements are documented and input into the next round of the spiral.



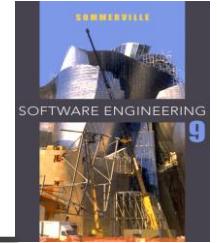
Problems of requirements elicitation

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



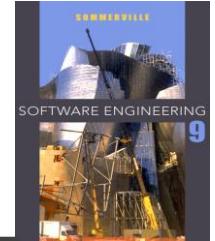
Key points

- ✧ The software requirements document is an agreed statement of the system requirements. It should be organized so that both system customers and software developers can use it.
- ✧ The requirements engineering process is an iterative process including requirements elicitation, specification and validation.
- ✧ Requirements elicitation and analysis is an iterative process that can be represented as a spiral of activities – requirements discovery, requirements classification and organization, requirements negotiation and requirements documentation.



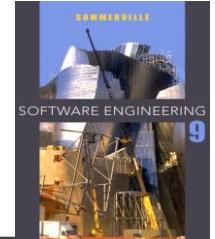
Chapter 4 – Requirements Engineering

Lecture 3



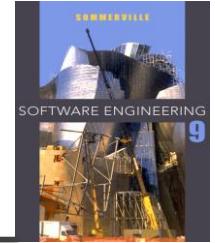
Requirements discovery

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



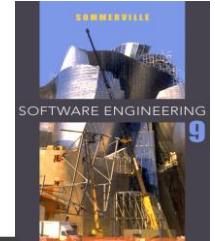
Stakeholders in the MHC-PMS

- ✧ Patients whose information is recorded in the system.
- ✧ Doctors who are responsible for assessing and treating patients.
- ✧ Nurses who coordinate the consultations with doctors and administer some treatments.
- ✧ Medical receptionists who manage patients' appointments.
- ✧ IT staff who are responsible for installing and maintaining the system.



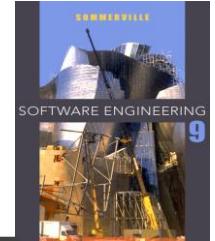
Stakeholders in the MHC-PMS

- ✧ A medical ethics manager who must ensure that the system meets current ethical guidelines for patient care.
- ✧ Health care managers who obtain management information from the system.
- ✧ Medical records staff who are responsible for ensuring that system information can be maintained and preserved, and that record keeping procedures have been properly implemented.



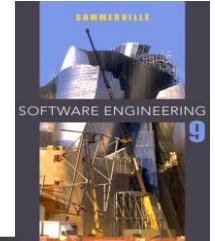
Interviewing

- ✧ Formal or informal interviews with stakeholders are part of most RE processes.
- ✧ Types of interview
 - Closed interviews based on pre-determined list of questions
 - Open interviews where various issues are explored with stakeholders.
- ✧ Effective interviewing
 - Be open-minded, avoid pre-conceived ideas about the requirements and are willing to listen to stakeholders.
 - Prompt the interviewee to get discussions going using a springboard question, a requirements proposal, or by working together on a prototype system.



Interviews in practice

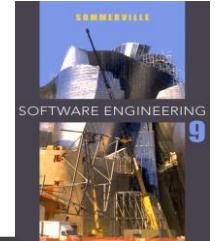
- ✧ Normally a mix of closed and open-ended interviewing.
- ✧ Interviews are good for getting an overall understanding of what stakeholders do and how they might interact with the system.
- ✧ Interviews are not good for understanding domain requirements
 - Requirements engineers cannot understand specific domain terminology;
 - Some domain knowledge is so familiar that people find it hard to articulate or think that it isn't worth articulating.



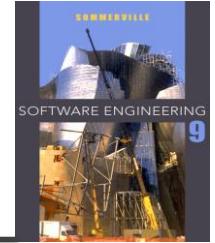
Scenarios

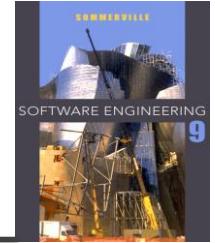
- ✧ Scenarios are real-life examples of how a system can be used.
- ✧ They should include
 - A description of the starting situation;
 - A description of the normal flow of events;
 - A description of what can go wrong;
 - Information about other concurrent activities;
 - A description of the state when the scenario finishes.

Scenario for collecting medical history in MHC-PMS



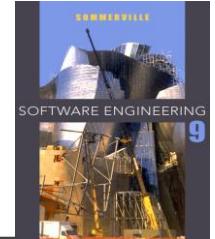
Scenario for collecting medical history in MHC-PMS



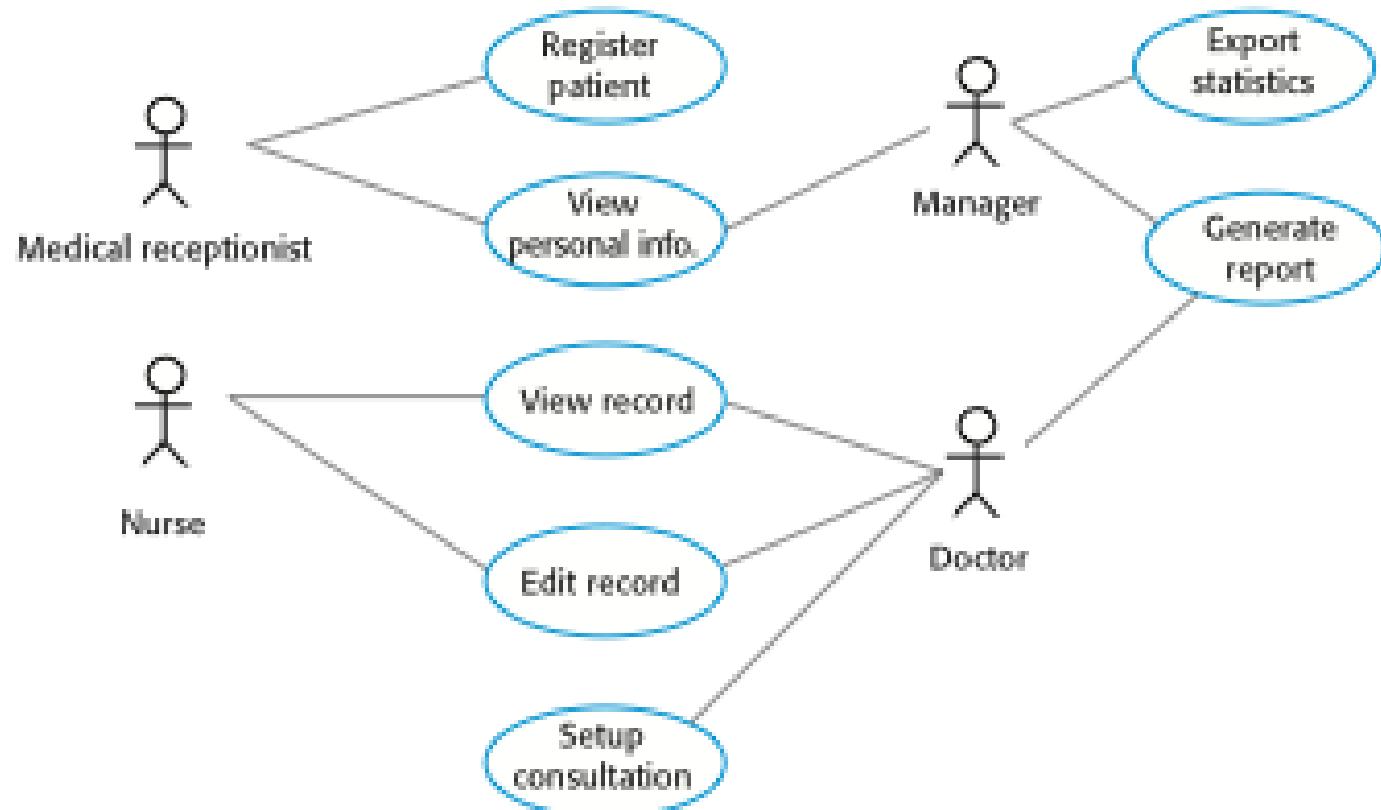


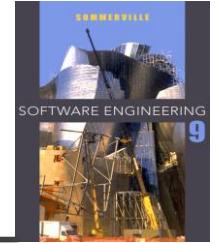
Use cases

- ✧ Use-cases are a scenario based technique in the UML which identify the actors in an interaction and which describe the interaction itself.
- ✧ A set of use cases should describe all possible interactions with the system.
- ✧ High-level graphical model supplemented by more detailed tabular description (see Chapter 5).
- ✧ Sequence diagrams may be used to add detail to use-cases by showing the sequence of event processing in the system.



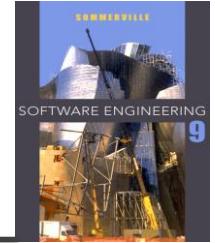
Use cases for the MHC-PMS





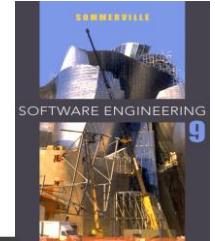
Ethnography

- ✧ A social scientist spends a considerable time observing and analysing how people actually work.
- ✧ People do not have to explain or articulate their work.
- ✧ Social and organisational factors of importance may be observed.
- ✧ Ethnographic studies have shown that work is usually richer and more complex than suggested by simple system models.



Scope of ethnography

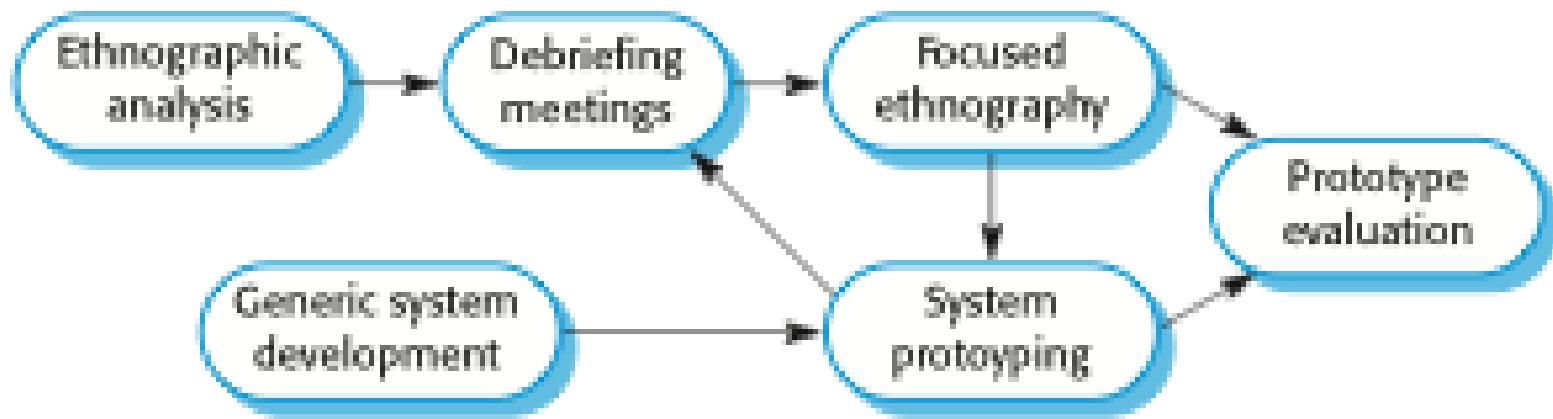
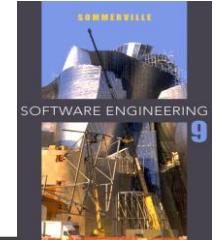
- ✧ Requirements that are derived from the way that people actually work rather than the way in which process definitions suggest that they ought to work.
- ✧ Requirements that are derived from cooperation and awareness of other people's activities.
 - Awareness of what other people are doing leads to changes in the ways in which we do things.
- ✧ Ethnography is effective for understanding existing processes but cannot identify new features that should be added to a system.

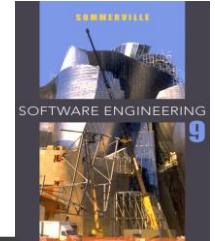


Focused ethnography

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

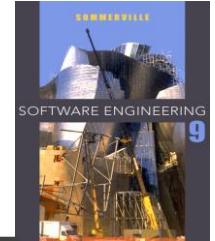
Ethnography and prototyping for requirements analysis





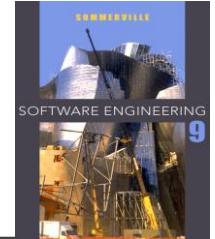
Requirements validation

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



Requirements checking

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



Requirements validation techniques

✧ Requirements reviews

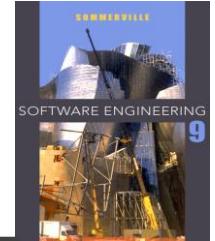
- Systematic manual analysis of the requirements.

✧ Prototyping

- Using an executable model of the system to check requirements.
Covered in Chapter 2.

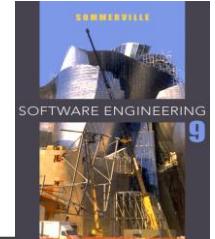
✧ Test-case generation

- Developing tests for requirements to check testability.



Requirements reviews

- ✧ Regular reviews should be held while the requirements definition is being formulated.
- ✧ Both client and contractor staff should be involved in reviews.
- ✧ Reviews may be formal (with completed documents) or informal. Good communications between developers, customers and users can resolve problems at an early stage.



Review checks

✧ Verifiability

- Is the requirement realistically testable?

✧ Comprehensibility

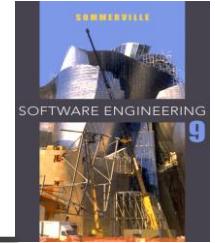
- Is the requirement properly understood?

✧ Traceability

- Is the origin of the requirement clearly stated?

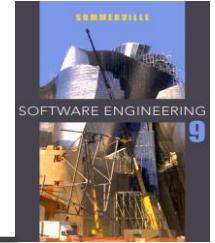
✧ Adaptability

- Can the requirement be changed without a large impact on other requirements?



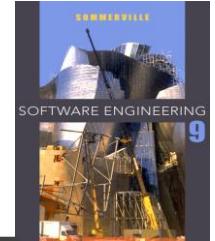
Requirements management

- ✧ Requirements management is the process of managing changing requirements during the requirements engineering process and system development.
- ✧ New requirements emerge as a system is being developed and after it has gone into use.
- ✧ You need to keep track of individual requirements and maintain links between dependent requirements so that you can assess the impact of requirements changes. You need to establish a formal process for making change proposals and linking these to system requirements.



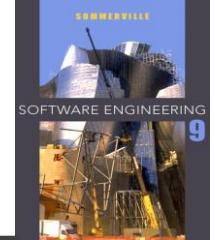
Changing requirements

- ✧ The business and technical environment of the system always changes after installation.
 - New hardware may be introduced, it may be necessary to interface the system with other systems, business priorities may change (with consequent changes in the system support required), and new legislation and regulations may be introduced that the system must necessarily abide by.
- ✧ The people who pay for a system and the users of that system are rarely the same people.
 - System customers impose requirements because of organizational and budgetary constraints. These may conflict with end-user requirements and, after delivery, new features may have to be added for user support if the system is to meet its goals.

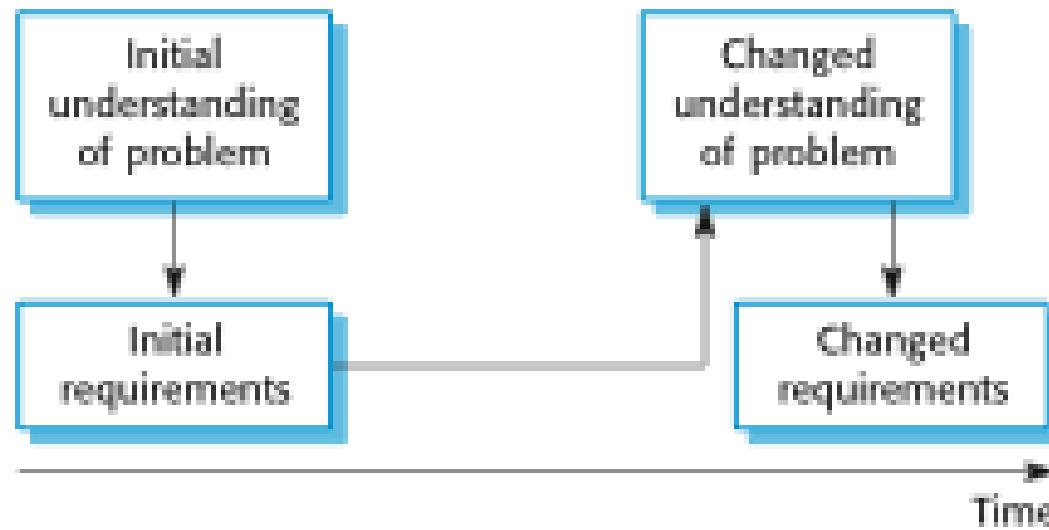


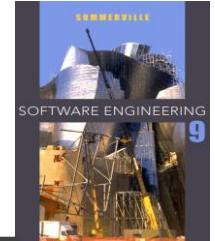
Changing requirements

- ✧ Large systems usually have a diverse user community, with many users having different requirements and priorities that may be conflicting or contradictory.
 - The final system requirements are inevitably a compromise between them and, with experience, it is often discovered that the balance of support given to different users has to be changed.



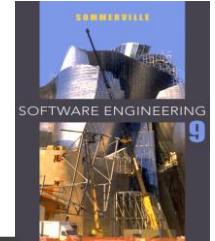
Requirements evolution





Requirements management planning

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



Requirements change management

✧ Deciding if a requirements change should be accepted

- *Problem analysis and change specification*

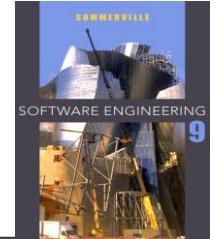
- During this stage, the problem or the change proposal is analyzed to check that it is valid. This analysis is fed back to the change requestor who may respond with a more specific requirements change proposal, or decide to withdraw the request.

- *Change analysis and costing*

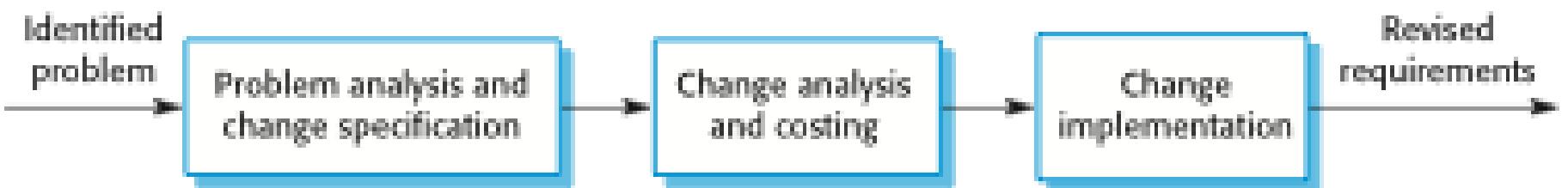
- The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements. Once this analysis is completed, a decision is made whether or not to proceed with the requirements change.

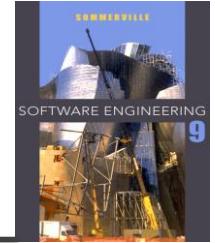
- *Change implementation*

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



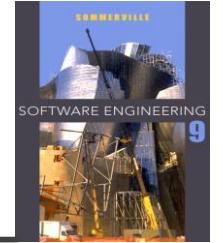
Requirements change management





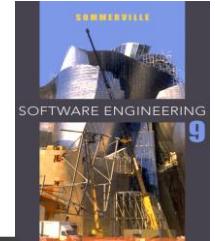
Key points

- ✧ You can use a range of techniques for requirements elicitation including interviews, scenarios, use-cases and ethnography.
- ✧ Requirements validation is the process of checking the requirements for validity, consistency, completeness, realism and verifiability.
- ✧ Business, organizational and technical changes inevitably lead to changes to the requirements for a software system. Requirements management is the process of managing and controlling these changes.



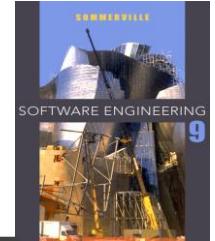
Chapter 3 – Agile Software Development

Lecture 1



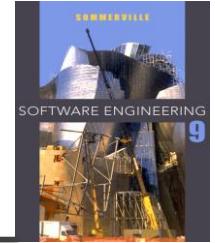
Topics covered

- ✧ Agile methods
- ✧ Plan-driven and agile development
- ✧ Extreme programming
- ✧ Agile project management
- ✧ Scaling agile methods



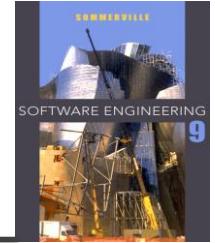
Rapid software development

- ✧ Rapid development and delivery is now often the most important requirement for software systems
 - Businesses operate in a fast –changing requirement and it is practically impossible to produce a set of stable software requirements
 - Software has to evolve quickly to reflect changing business needs.
- ✧ Rapid software development
 - Specification, design and implementation are inter-leaved
 - System is developed as a series of versions with stakeholders involved in version evaluation
 - User interfaces are often developed using an IDE and graphical toolset.



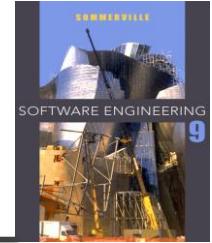
Agile methods

- ✧ Dissatisfaction with the overheads involved in software design methods of the 1980s and 1990s led to the creation of agile methods. These methods:
 - Focus on the code rather than the design
 - Are based on an iterative approach to software development
 - Are intended to deliver working software quickly and evolve this quickly to meet changing requirements.
- ✧ The aim of agile methods is to reduce overheads in the software process (e.g. by limiting documentation) and to be able to respond quickly to changing requirements without excessive rework.



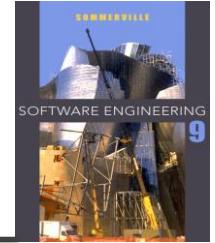
Agile manifesto

- ✧ We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
 - Individuals and interactions over processes and tools
 - Working software over comprehensive documentation
 - Customer collaboration over contract negotiation
 - Responding to change over following a plan
- ✧ That is, while there is value in the items on the right, we value the items on the left more.



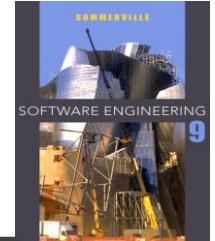
The principles of agile methods

Principle	Description
Customer involvement	Customers should be closely involved throughout the development process. Their role is provide and prioritize new system requirements and to evaluate the iterations of the system.
Incremental delivery	The software is developed in increments with the customer specifying the requirements to be included in each increment.
People not process	The skills of the development team should be recognized and exploited. Team members should be left to develop their own ways of working without prescriptive processes.
Embrace change	Expect the system requirements to change and so design the system to accommodate these changes.
Maintain simplicity	Focus on simplicity in both the software being developed and in the development process. Wherever possible, actively work to eliminate complexity from the system.



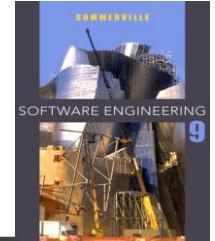
Agile method applicability

- ✧ Product development where a software company is developing a small or medium-sized product for sale.
- ✧ Custom system development within an organization, where there is a clear commitment from the customer to become involved in the development process and where there are not a lot of external rules and regulations that affect the software.
- ✧ Because of their focus on small, tightly-integrated teams, there are problems in scaling agile methods to large systems.



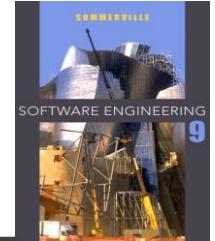
Problems with agile methods

- ✧ It can be difficult to keep the interest of customers who are involved in the process.
- ✧ Team members may be unsuited to the intense involvement that characterises agile methods.
- ✧ Prioritising changes can be difficult where there are multiple stakeholders.
- ✧ Maintaining simplicity requires extra work.
- ✧ Contracts may be a problem as with other approaches to iterative development.



Agile methods and software maintenance

- ✧ Most organizations spend more on maintaining existing software than they do on new software development. So, if agile methods are to be successful, they have to support maintenance as well as original development.
- ✧ Two key issues:
 - Are systems that are developed using an agile approach maintainable, given the emphasis in the development process of minimizing formal documentation?
 - Can agile methods be used effectively for evolving a system in response to customer change requests?
- ✧ Problems may arise if original development team cannot be maintained.



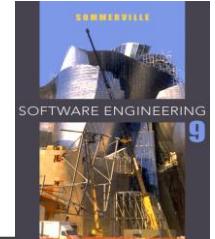
Plan-driven and agile development

✧ Plan-driven development

- A plan-driven approach to software engineering is based around separate development stages with the outputs to be produced at each of these stages planned in advance.
- Not necessarily waterfall model – plan-driven, incremental development is possible
- Iteration occurs within activities.

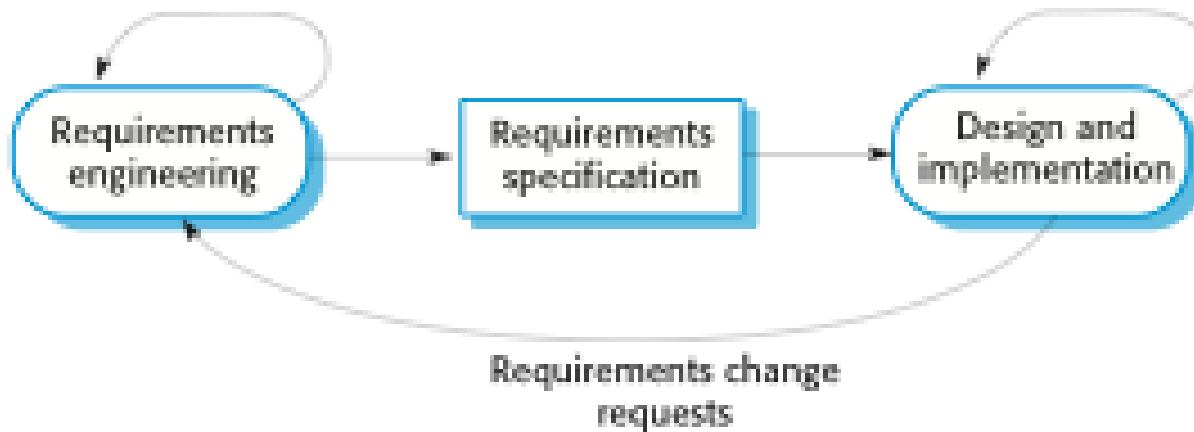
✧ Agile development

- Specification, design, implementation and testing are interleaved and the outputs from the development process are decided through a process of negotiation during the software development process.



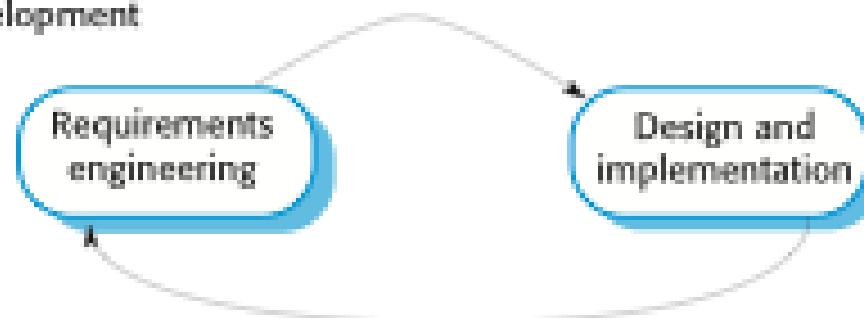
Plan-driven and agile specification

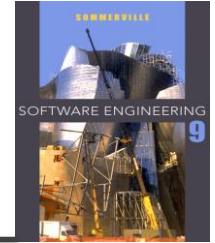
Plan-based development



Requirements change
requests

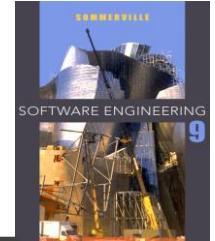
Agile development





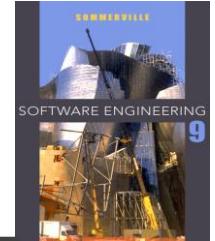
Technical, human, organizational issues

- ✧ Most projects include elements of plan-driven and agile processes. Deciding on the balance depends on:
 - Is it important to have a very detailed specification and design before moving to implementation? If so, you probably need to use a plan-driven approach.
 - Is an incremental delivery strategy, where you deliver the software to customers and get rapid feedback from them, realistic? If so, consider using agile methods.
 - How large is the system that is being developed? Agile methods are most effective when the system can be developed with a small co-located team who can communicate informally. This may not be possible for large systems that require larger development teams so a plan-driven approach may have to be used.



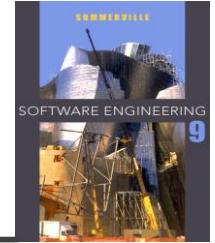
Technical, human, organizational issues

- What type of system is being developed?
 - Plan-driven approaches may be required for systems that require a lot of analysis before implementation (e.g. real-time system with complex timing requirements).
- What is the expected system lifetime?
 - Long-lifetime systems may require more design documentation to communicate the original intentions of the system developers to the support team.
- What technologies are available to support system development?
 - Agile methods rely on good tools to keep track of an evolving design
- How is the development team organized?
 - If the development team is distributed or if part of the development is being outsourced, then you may need to develop design documents to communicate across the development teams.



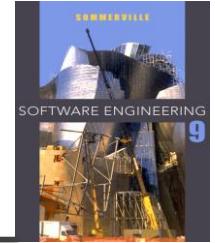
Technical, human, organizational issues

- Are there cultural or organizational issues that may affect the system development?
 - Traditional engineering organizations have a culture of plan-based development, as this is the norm in engineering.
- How good are the designers and programmers in the development team?
 - It is sometimes argued that agile methods require higher skill levels than plan-based approaches in which programmers simply translate a detailed design into code
- Is the system subject to external regulation?
 - If a system has to be approved by an external regulator (e.g. the FAA approve software that is critical to the operation of an aircraft) then you will probably be required to produce detailed documentation as part of the system safety case.



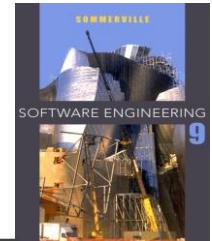
Extreme programming

- ✧ Perhaps the best-known and most widely used agile method.
- ✧ Extreme Programming (XP) takes an ‘extreme’ approach to iterative development.
 - New versions may be built several times per day;
 - Increments are delivered to customers every 2 weeks;
 - All tests must be run for every build and the build is only accepted if tests run successfully.

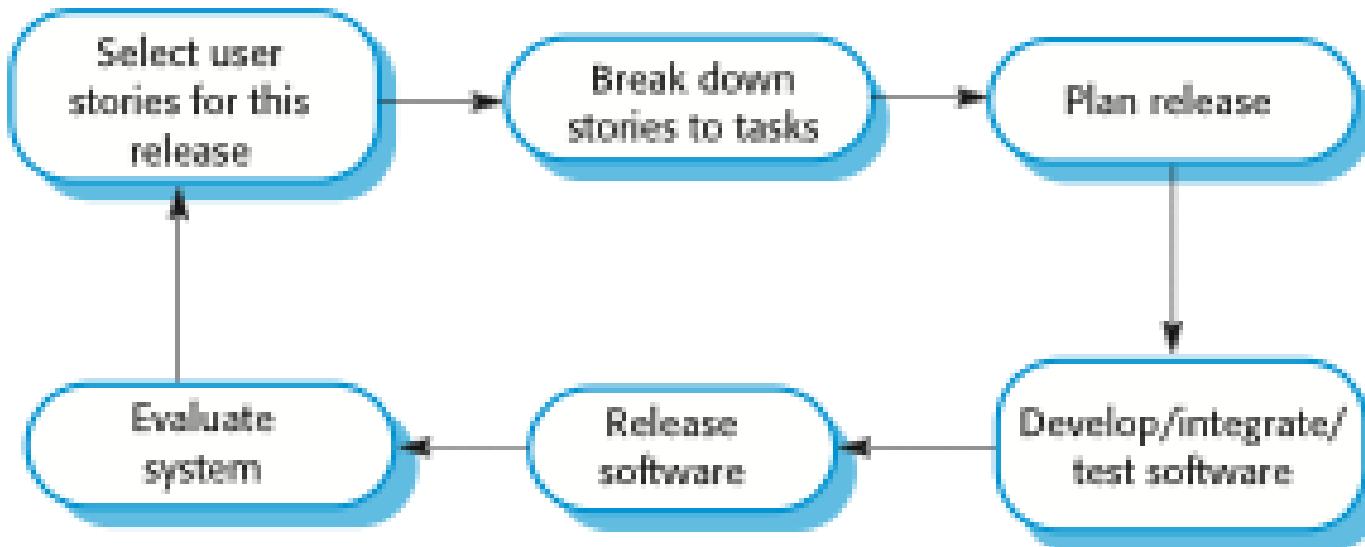


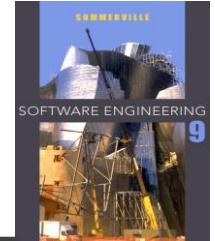
XP and agile principles

- ✧ Incremental development is supported through small, frequent system releases.
- ✧ Customer involvement means full-time customer engagement with the team.
- ✧ People not process through pair programming, collective ownership and a process that avoids long working hours.
- ✧ Change supported through regular system releases.
- ✧ Maintaining simplicity through constant refactoring of code.



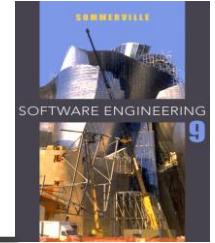
The extreme programming release cycle





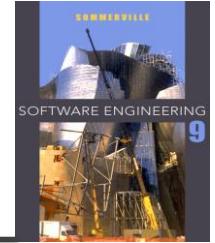
Extreme programming practices (a)

Principle or practice	Description
Incremental planning	Requirements are recorded on story cards and the stories to be included in a release are determined by the time available and their relative priority. The developers break these stories into development 'Tasks'. See Figures 3.5 and 3.6.
Small releases	The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the first release.
Simple design	Enough design is carried out to meet the current requirements and no more.
Test-first development	An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.
Refactoring	All developers are expected to refactor the code continuously as soon as possible code improvements are found. This keeps the code simple and maintainable.



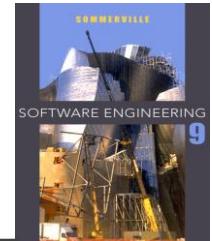
Extreme programming practices (b)

Pair programming	Developers work in pairs, checking each other's work and providing the support to always do a good job.
Collective ownership	The pairs of developers work on all areas of the system, so that no islands of expertise develop and all the developers take responsibility for all of the code. Anyone can change anything.
Continuous integration	As soon as the work on a task is complete, it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.
Sustainable pace	Large amounts of overtime are not considered acceptable as the net effect is often to reduce code quality and medium term productivity
On-site customer	A representative of the end-user of the system (the customer) should be available full time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.



Requirements scenarios

- ✧ In XP, a customer or user is part of the XP team and is responsible for making decisions on requirements.
- ✧ User requirements are expressed as scenarios or user stories.
- ✧ These are written on cards and the development team break them down into implementation tasks. These tasks are the basis of schedule and cost estimates.
- ✧ The customer chooses the stories for inclusion in the next release based on their priorities and the schedule estimates.



A ‘prescribing medication’ story

Prescribing medication

The record of the patient must be open for input. Click on the medication field and select either ‘current medication’, ‘new medication’ or ‘formulary’.

If you select ‘current medication’, you will be asked to check the dose; If you wish to change the dose, enter the new dose then confirm the prescription.

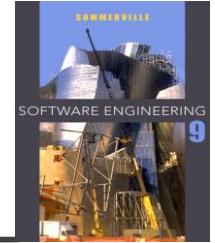
If you choose, ‘new medication’, the system assumes that you know which medication you wish to prescribe. Type the first few letters of the drug name. You will then see a list of possible drugs starting with these letters. Choose the required medication. You will then be asked to check that the medication you have selected is correct. Enter the dose then confirm the prescription.

If you choose ‘formulary’, you will be presented with a search box for the approved formulary. Search for the drug required then select it. You will then be asked to check that the medication you have selected is correct. Enter the dose then confirm the prescription.

In all cases, the system will check that the dose is within the approved range and will ask you to change it if it is outside the range of recommended doses.

After you have confirmed the prescription, it will be displayed for checking. Either click ‘OK’ or ‘Change’. If you click ‘OK’, your prescription will be recorded on the audit database. If you click ‘Change’, you reenter the ‘Prescribing medication’ process.

Examples of task cards for prescribing medication



Task 1: Change dose of prescribed drug

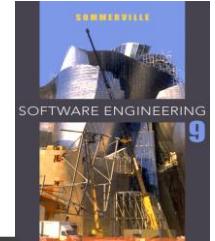
Task 2: Formulary selection

Task 3: Dose checking

Dose checking is a safety precaution to check that the doctor has not prescribed a dangerously small or large dose.

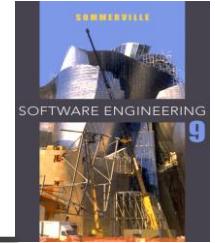
Using the formulary id for the generic drug name, lookup the formulary and retrieve the recommended maximum and minimum dose.

Check the prescribed dose against the minimum and maximum. If outside the range, issue an error message saying that the dose is too high or too low. If within the range, enable the 'Confirm' button.



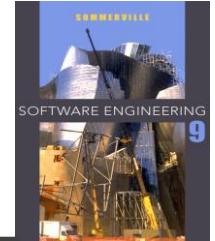
XP and change

- ✧ Conventional wisdom in software engineering is to design for change. It is worth spending time and effort anticipating changes as this reduces costs later in the life cycle.
- ✧ XP, however, maintains that this is not worthwhile as changes cannot be reliably anticipated.
- ✧ Rather, it proposes constant code improvement (refactoring) to make changes easier when they have to be implemented.



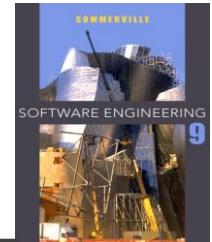
Refactoring

- ✧ Programming team look for possible software improvements and make these improvements even where there is no immediate need for them.
- ✧ This improves the understandability of the software and so reduces the need for documentation.
- ✧ Changes are easier to make because the code is well-structured and clear.
- ✧ However, some changes require architecture refactoring and this is much more expensive.



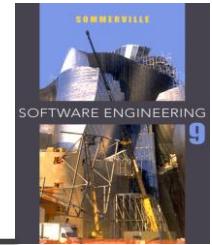
Examples of refactoring

- ✧ Re-organization of a class hierarchy to remove duplicate code.
- ✧ Tidying up and renaming attributes and methods to make them easier to understand.
- ✧ The replacement of inline code with calls to methods that have been included in a program library.



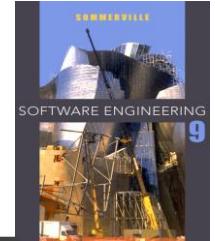
Key points

- ✧ Agile methods are incremental development methods that focus on rapid development, frequent releases of the software, reducing process overheads and producing high-quality code. They involve the customer directly in the development process.
- ✧ The decision on whether to use an agile or a plan-driven approach to development should depend on the type of software being developed, the capabilities of the development team and the culture of the company developing the system.
- ✧ Extreme programming is a well-known agile method that integrates a range of good programming practices such as frequent releases of the software, continuous software improvement and customer participation in the development team.



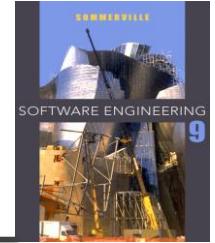
Chapter 3 – Agile Software Development

Lecture 2



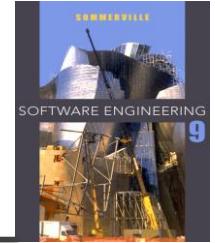
Testing in XP

- ✧ Testing is central to XP and XP has developed an approach where the program is tested after every change has been made.
- ✧ XP testing features:
 - Test-first development.
 - Incremental test development from scenarios.
 - User involvement in test development and validation.
 - Automated test harnesses are used to run all component tests each time that a new release is built.



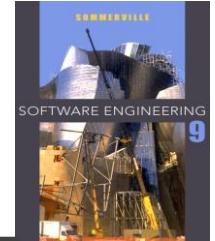
Test-first development

- ✧ Writing tests before code clarifies the requirements to be implemented.
- ✧ Tests are written as programs rather than data so that they can be executed automatically. The test includes a check that it has executed correctly.
 - Usually relies on a testing framework such as Junit.
- ✧ All previous and new tests are run automatically when new functionality is added, thus checking that the new functionality has not introduced errors.



Customer involvement

- ✧ The role of the customer in the testing process is to help develop acceptance tests for the stories that are to be implemented in the next release of the system.
- ✧ The customer who is part of the team writes tests as development proceeds. All new code is therefore validated to ensure that it is what the customer needs.
- ✧ However, people adopting the customer role have limited time available and so cannot work full-time with the development team. They may feel that providing the requirements was enough of a contribution and so may be reluctant to get involved in the testing process.



Test case description for dose checking

Test 4: Dose checking

Input:

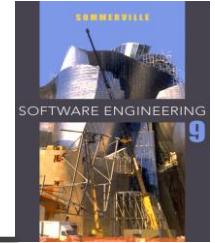
1. A number in mg representing a single dose of the drug.
2. A number representing the number of single doses per day.

Tests:

1. Test for inputs where the single dose is correct but the frequency is too high.
2. Test for inputs where the single dose is too high and too low.
3. Test for inputs where the single dose * frequency is too high and too low.
4. Test for inputs where single dose * frequency is in the permitted range.

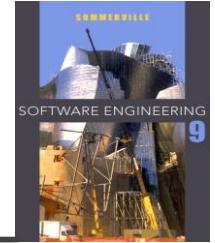
Output:

OK or error message indicating that the dose is outside the safe range.



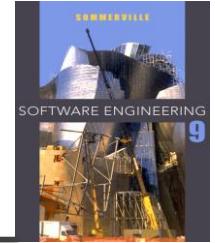
Test automation

- ✧ Test automation means that tests are written as executable components before the task is implemented
 - These testing components should be stand-alone, should simulate the submission of input to be tested and should check that the result meets the output specification. An automated test framework (e.g. Junit) is a system that makes it easy to write executable tests and submit a set of tests for execution.
- ✧ As testing is automated, there is always a set of tests that can be quickly and easily executed
 - Whenever any functionality is added to the system, the tests can be run and problems that the new code has introduced can be caught immediately.



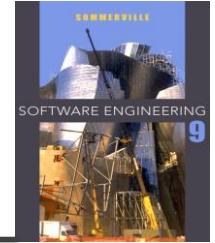
XP testing difficulties

- ✧ Programmers prefer programming to testing and sometimes they take short cuts when writing tests. For example, they may write incomplete tests that do not check for all possible exceptions that may occur.
- ✧ Some tests can be very difficult to write incrementally. For example, in a complex user interface, it is often difficult to write unit tests for the code that implements the 'display logic' and workflow between screens.
- ✧ It difficult to judge the completeness of a set of tests. Although you may have a lot of system tests, your test set may not provide complete coverage.



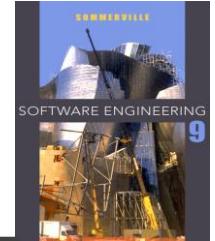
Pair programming

- ✧ In XP, programmers work in pairs, sitting together to develop code.
- ✧ This helps develop common ownership of code and spreads knowledge across the team.
- ✧ It serves as an informal review process as each line of code is looked at by more than 1 person.
- ✧ It encourages refactoring as the whole team can benefit from this.
- ✧ Measurements suggest that development productivity with pair programming is similar to that of two people working independently.



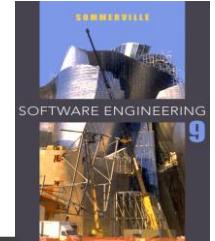
Pair programming

- ✧ In pair programming, programmers sit together at the same workstation to develop the software.
- ✧ Pairs are created dynamically so that all team members work with each other during the development process.
- ✧ The sharing of knowledge that happens during pair programming is very important as it reduces the overall risks to a project when team members leave.
- ✧ Pair programming is not necessarily inefficient and there is evidence that a pair working together is more efficient than 2 programmers working separately.



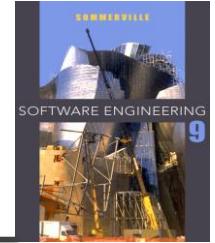
Advantages of pair programming

- ✧ It supports the idea of collective ownership and responsibility for the system.
 - Individuals are not held responsible for problems with the code. Instead, the team has collective responsibility for resolving these problems.
- ✧ It acts as an informal review process because each line of code is looked at by at least two people.
- ✧ It helps support refactoring, which is a process of software improvement.
 - Where pair programming and collective ownership are used, others benefit immediately from the refactoring so they are likely to support the process.



Agile project management

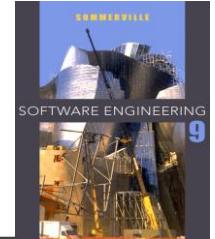
- ✧ The principal responsibility of software project managers is to manage the project so that the software is delivered on time and within the planned budget for the project.
- ✧ The standard approach to project management is plan-driven. Managers draw up a plan for the project showing what should be delivered, when it should be delivered and who will work on the development of the project deliverables.
- ✧ Agile project management requires a different approach, which is adapted to incremental development and the particular strengths of agile methods.



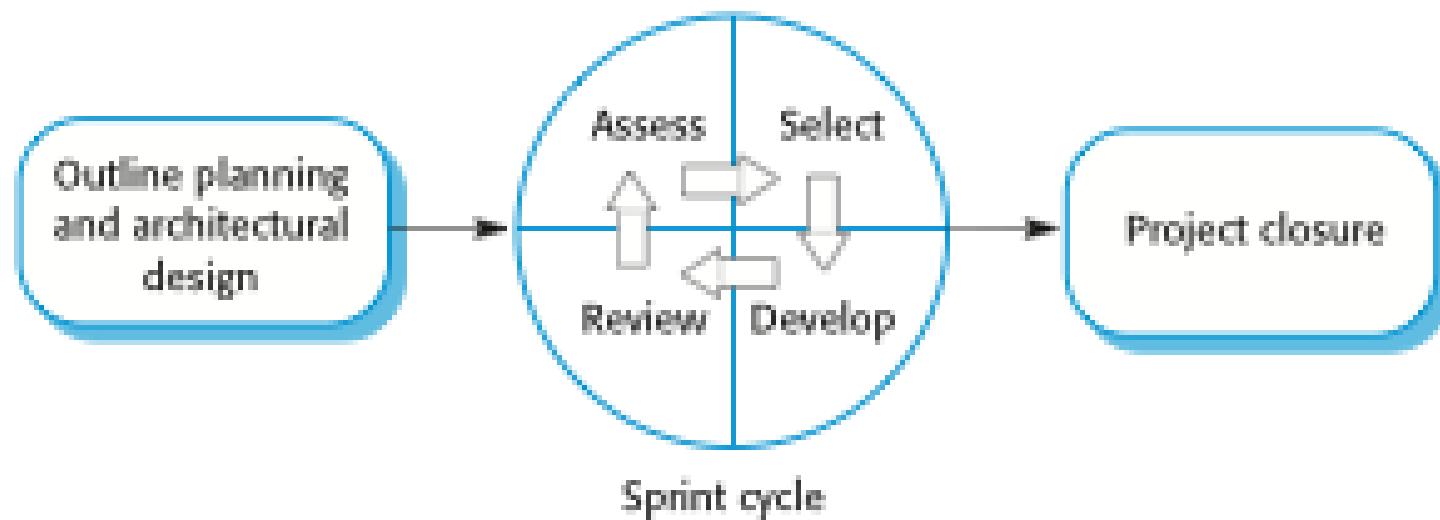
Scrum

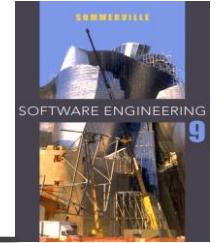
- ✧ The Scrum approach is a general agile method but its focus is on managing iterative development rather than specific agile practices.
- ✧ There are three phases in Scrum.
 - The initial phase is an outline planning phase where you establish the general objectives for the project and design the software architecture.
 - This is followed by a series of sprint cycles, where each cycle develops an increment of the system.
 - The project closure phase wraps up the project, completes required documentation such as system help frames and user manuals and assesses the lessons learned from the project.





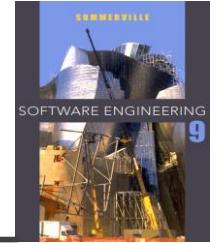
The Scrum process





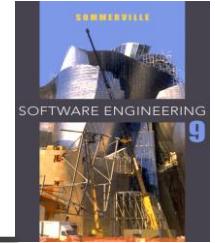
The Sprint cycle

- ✧ Sprints are fixed length, normally 2–4 weeks. They correspond to the development of a release of the system in XP.
- ✧ The starting point for planning is the product backlog, which is the list of work to be done on the project.
- ✧ The selection phase involves all of the project team who work with the customer to select the features and functionality to be developed during the sprint.



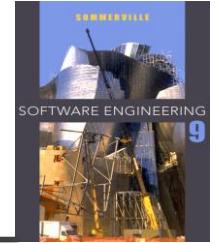
The Sprint cycle

- ✧ Once these are agreed, the team organize themselves to develop the software. During this stage the team is isolated from the customer and the organization, with all communications channelled through the so-called 'Scrum master'.
- ✧ The role of the Scrum master is to protect the development team from external distractions.
- ✧ At the end of the sprint, the work done is reviewed and presented to stakeholders. The next sprint cycle then begins.



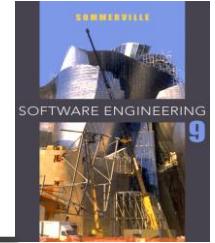
Teamwork in Scrum

- ✧ The ‘Scrum master’ is a facilitator who arranges daily meetings, tracks the backlog of work to be done, records decisions, measures progress against the backlog and communicates with customers and management outside of the team.
- ✧ The whole team attends short daily meetings where all team members share information, describe their progress since the last meeting, problems that have arisen and what is planned for the following day.
 - This means that everyone on the team knows what is going on and, if problems arise, can re-plan short-term work to cope with them.



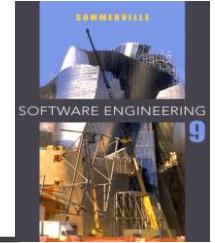
Scrum benefits

- ✧ The product is broken down into a set of manageable and understandable chunks.
- ✧ Unstable requirements do not hold up progress.
- ✧ The whole team have visibility of everything and consequently team communication is improved.
- ✧ Customers see on-time delivery of increments and gain feedback on how the product works.
- ✧ Trust between customers and developers is established and a positive culture is created in which everyone expects the project to succeed.



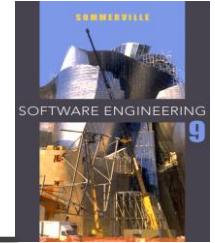
Scaling agile methods

- ✧ Agile methods have proved to be successful for small and medium sized projects that can be developed by a small co-located team.
- ✧ It is sometimes argued that the success of these methods comes because of improved communications which is possible when everyone is working together.
- ✧ Scaling up agile methods involves changing these to cope with larger, longer projects where there are multiple development teams, perhaps working in different locations.



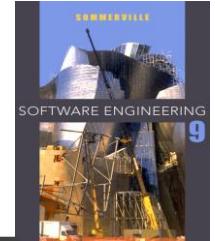
Large systems development

- ✧ Large systems are usually collections of separate, communicating systems, where separate teams develop each system. Frequently, these teams are working in different places, sometimes in different time zones.
- ✧ Large systems are ‘brownfield systems’, that is they include and interact with a number of existing systems. Many of the system requirements are concerned with this interaction and so don’t really lend themselves to flexibility and incremental development.
- ✧ Where several systems are integrated to create a system, a significant fraction of the development is concerned with system configuration rather than original code development.



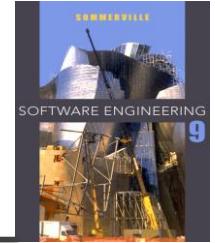
Large system development

- ✧ Large systems and their development processes are often constrained by external rules and regulations limiting the way that they can be developed.
- ✧ Large systems have a long procurement and development time. It is difficult to maintain coherent teams who know about the system over that period as, inevitably, people move on to other jobs and projects.
- ✧ Large systems usually have a diverse set of stakeholders. It is practically impossible to involve all of these different stakeholders in the development process.



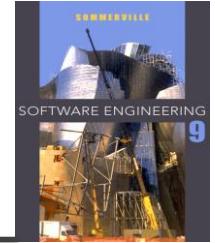
Scaling out and scaling up

- ✧ ‘Scaling up’ is concerned with using agile methods for developing large software systems that cannot be developed by a small team.
- ✧ ‘Scaling out’ is concerned with how agile methods can be introduced across a large organization with many years of software development experience.
- ✧ When scaling agile methods it is essential to maintain agile fundamentals
 - Flexible planning, frequent system releases, continuous integration, test-driven development and good team communications.



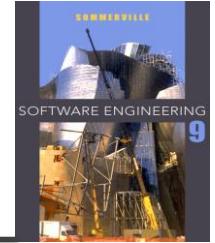
Scaling up to large systems

- ✧ For large systems development, it is not possible to focus only on the code of the system. You need to do more up-front design and system documentation
- ✧ Cross-team communication mechanisms have to be designed and used. This should involve regular phone and video conferences between team members and frequent, short electronic meetings where teams update each other on progress.
- ✧ Continuous integration, where the whole system is built every time any developer checks in a change, is practically impossible. However, it is essential to maintain frequent system builds and regular releases of the system.



Scaling out to large companies

- ✧ Project managers who do not have experience of agile methods may be reluctant to accept the risk of a new approach.
- ✧ Large organizations often have quality procedures and standards that all projects are expected to follow and, because of their bureaucratic nature, these are likely to be incompatible with agile methods.
- ✧ Agile methods seem to work best when team members have a relatively high skill level. However, within large organizations, there are likely to be a wide range of skills and abilities.
- ✧ There may be cultural resistance to agile methods, especially in those organizations that have a long history of using conventional systems engineering processes.



Key points

- ✧ A particular strength of extreme programming is the development of automated tests before a program feature is created. All tests must successfully execute when an increment is integrated into a system.
- ✧ The Scrum method is an agile method that provides a project management framework. It is centred round a set of sprints, which are fixed time periods when a system increment is developed.
- ✧ Scaling agile methods for large systems is difficult. Large systems need up-front design and some documentation.

05 System Modelling

CS3203 – Software Engineering

By Kutila Gunasekera
(Based on the slides by Prof. Ian Sommerville)

Objectives

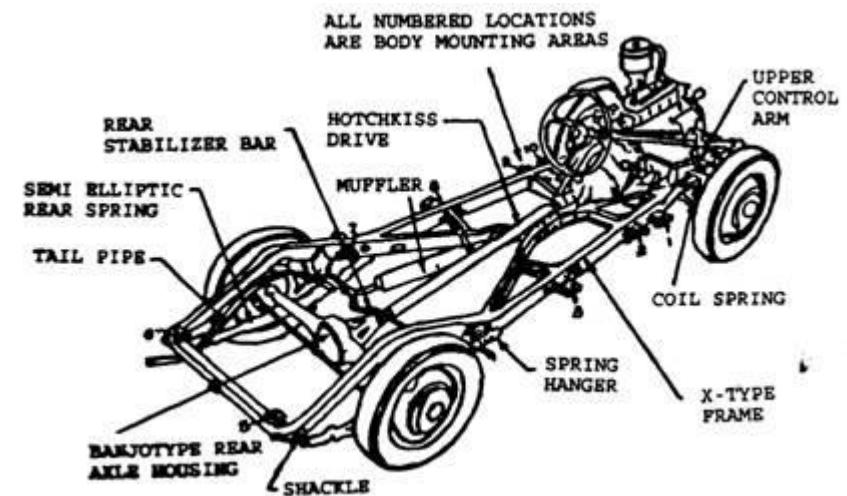
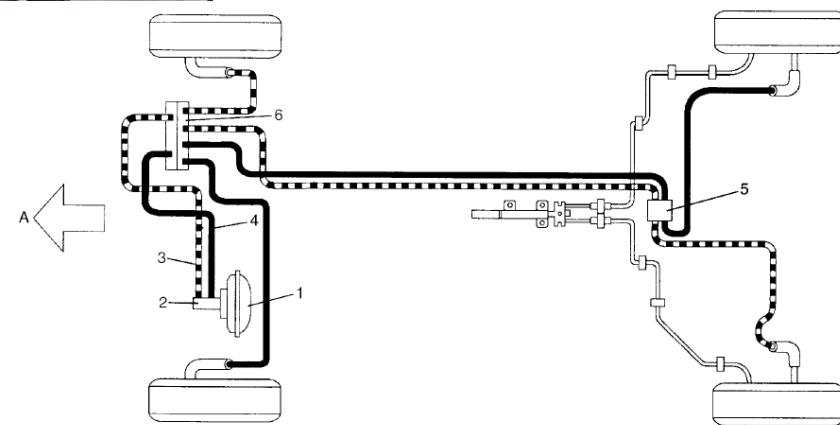
- To understand how **graphical models** can be used to represent software systems
- Understand **why different types** of models are required, and the basic system modelling perspectives of **context, interaction, structure** and **behaviour**
- Be introduced to some diagram types in UML and how they can be used in system modelling
- Be aware of the ideas underlying **model-driven engineering**, where a system is automatically generated from structural and behavioural models.

Topics covered

- Context models
- Interaction models
- Structural models
- Behavioral models
- Model-driven engineering

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.



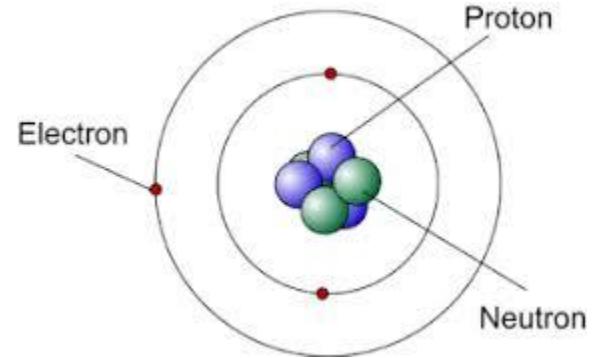
System modeling

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

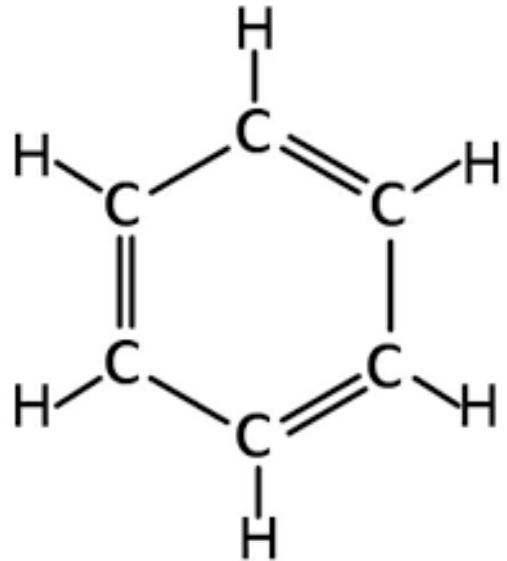
Existing and planned system models

- Models of the **existing system** are used during requirements engineering.
 - 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.
 - 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.

Use of graphical models



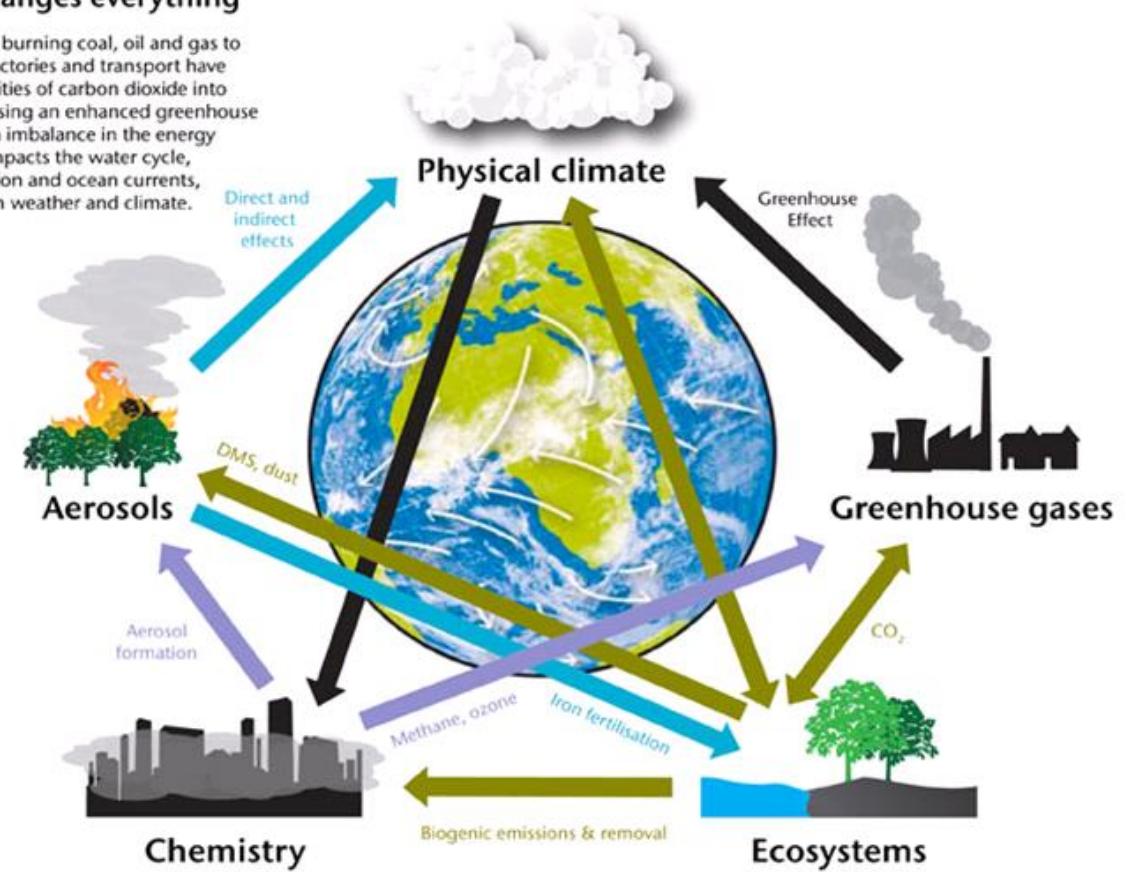
1. As a means of facilitating discussion about an existing or proposed system
 - Incomplete and incorrect models are OK as their role is to support discussion.
2. As a way of documenting an existing system
 - Models should be an **accurate representation** of the system but need not be complete.
3. As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



The Earth System

One thing changes everything

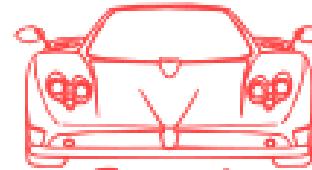
Human activities like burning coal, oil and gas to power our homes, factories and transport have released huge quantities of carbon dioxide into the atmosphere, causing an enhanced greenhouse effect. This causes an imbalance in the energy cycle that, in turn, impacts the water cycle, atmospheric circulation and ocean currents, leading to changes in weather and climate.



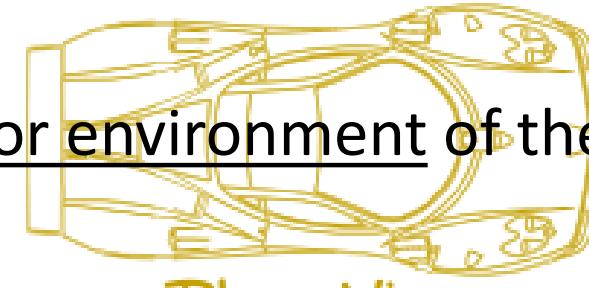
System perspectives (views)



Side View



Front View



Plan View

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

UML diagram types

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

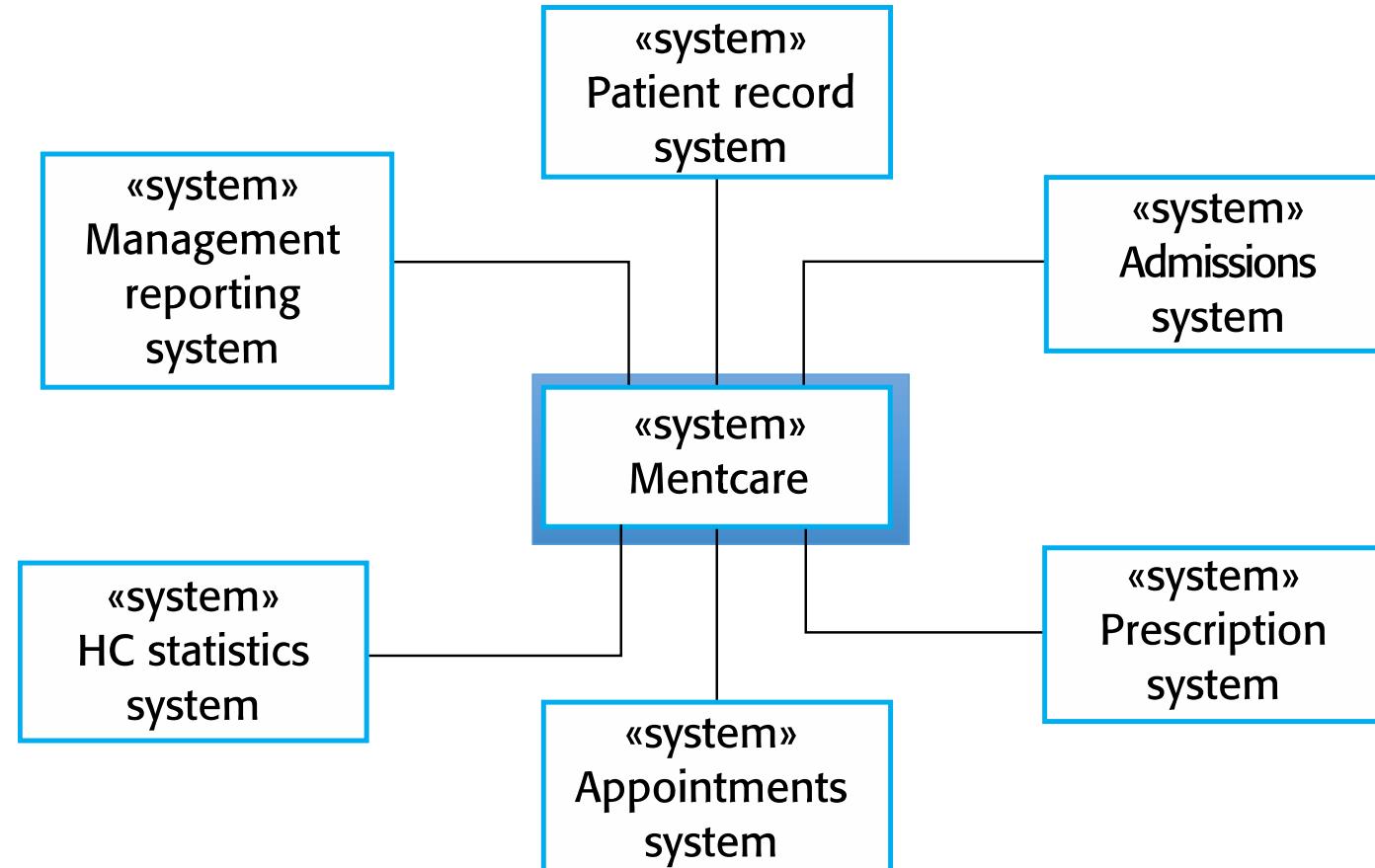
Context models

- 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.

The context of the Mentcare system

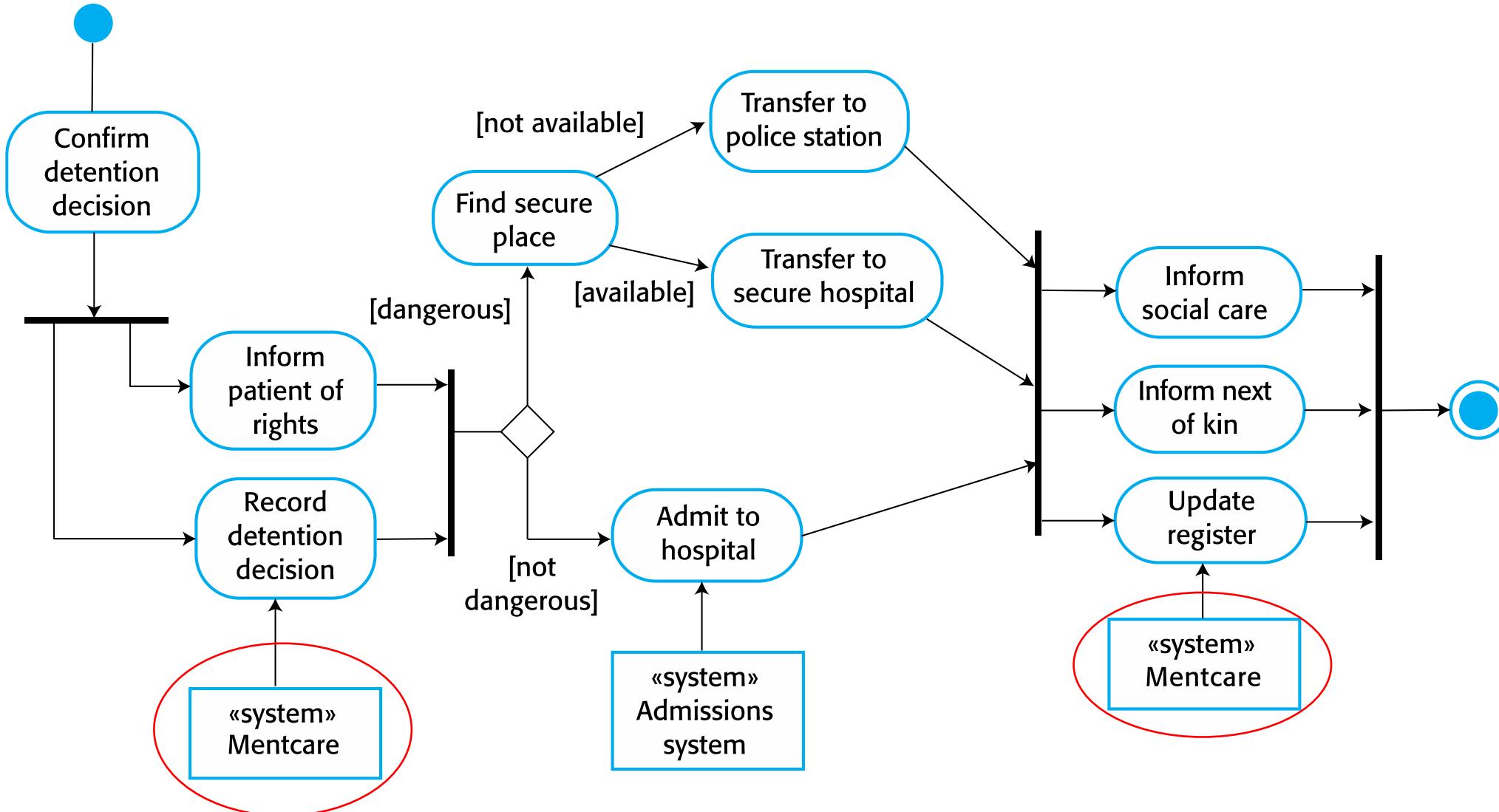


Mentcare: A system to support the clinical management of patients suffering from mental illness

Process perspective

- *Context models simply show the other systems in the environment, not how the system being developed is used in that environment.*
- Process models reveal **how** the system being developed is used in broader business processes.
- **UML activity diagrams** may be used to define business process models.

Process model of involuntary detention



Interaction models

- Modelling user interaction is important as it helps to identify user requirements.
- Modelling system-to-system interaction highlights the communication problems that may arise.
- Modelling component interaction helps understand if a proposed system structure is likely to deliver the required system performance and dependability.
- **Use case diagrams** and **sequence diagrams** may be used for interaction modelling.

Use case modelling

- Use cases were developed originally to support requirements.
- Each use case represents a discrete task that involves external interaction with a system.
- Actors in a use case may be people or other systems.
- Represented diagrammatically to provide an overview of the use case and in a more detailed textual form.

Transfer-data use case

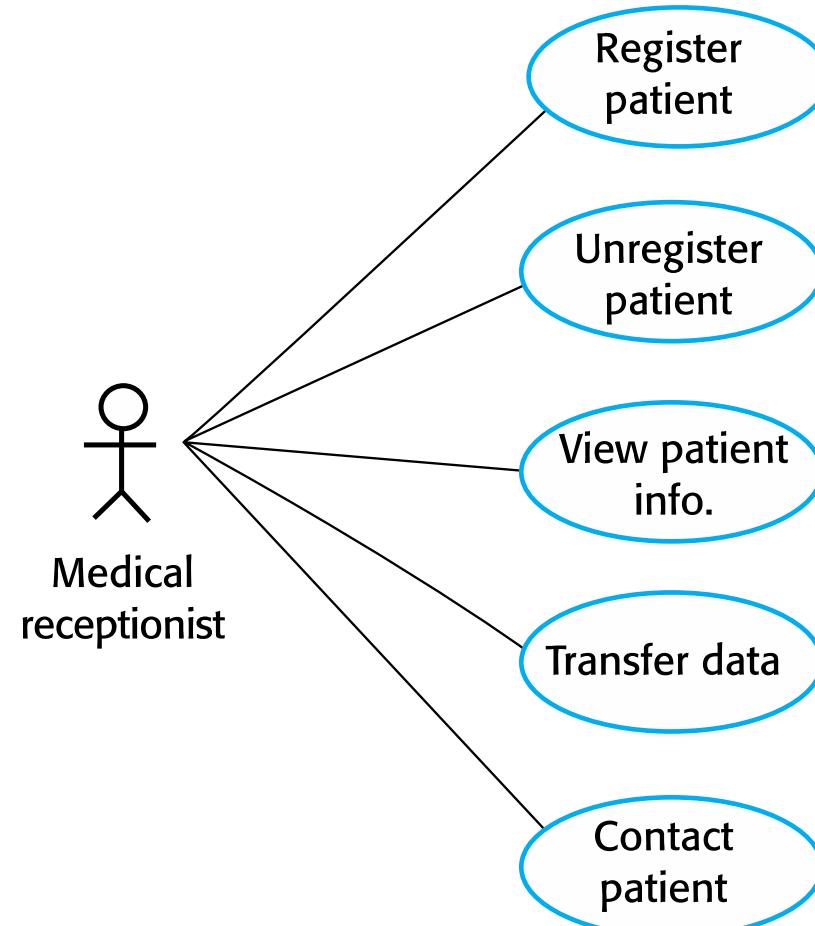
- A use case in the Mentcare system



Tabular description of 'Transfer data' use-case

Mentcare: Transfer data	
Actors	Medical receptionist, patient records system (PRS)
Description	A receptionist may transfer data from the MENTCARE to a general patient record database that is maintained by a health authority. The information transferred may either be updated personal information (address, phone number, etc.) or a summary of the patient's diagnosis and treatment.
Data	Patient's personal information, treatment summary
Stimulus	User command issued by medical receptionist
Response	Confirmation that PRS has been updated
Comments	The receptionist must have appropriate security permissions to access the patient information and the PRS.

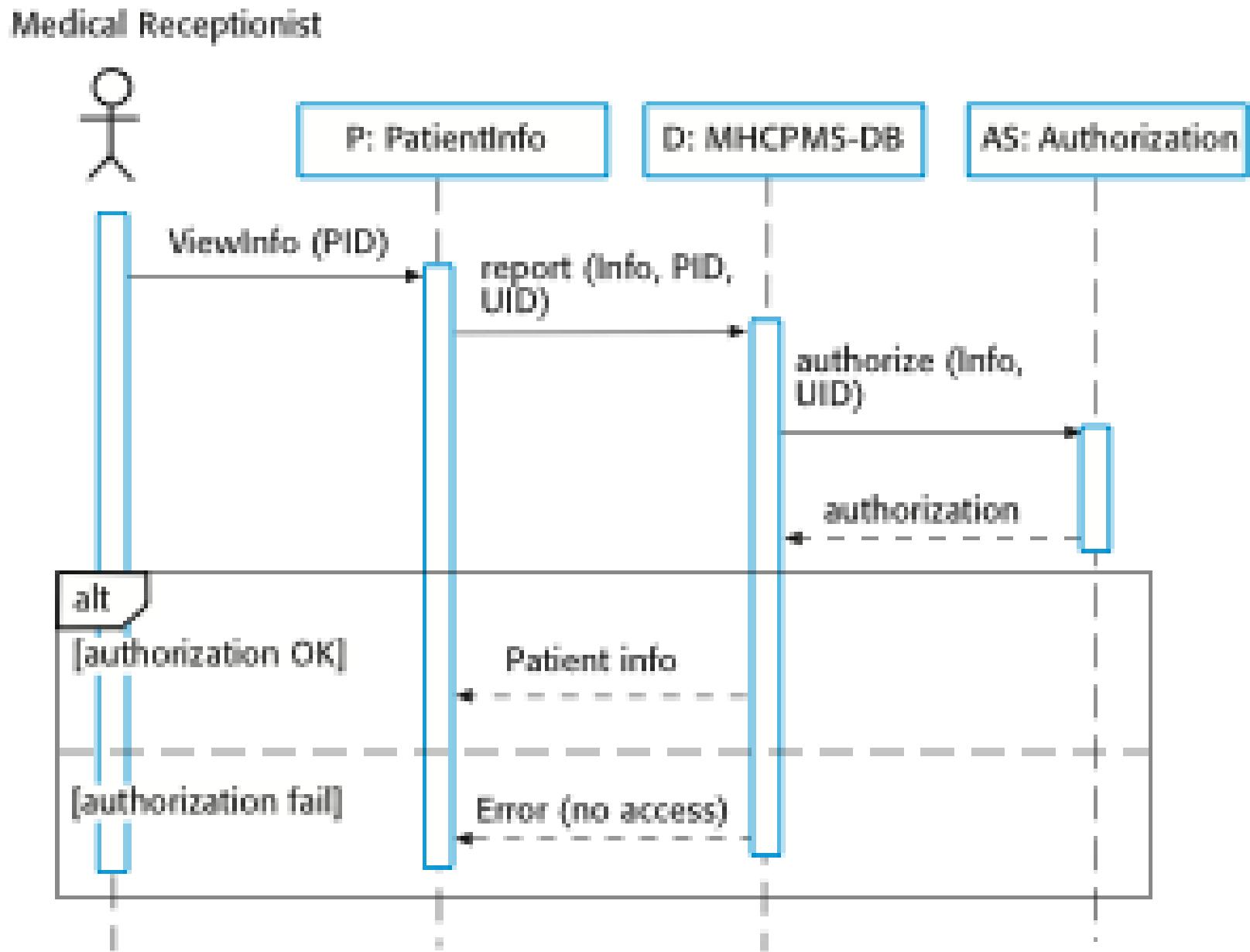
Use cases in the Mentcare involving the role 'Medical Receptionist'



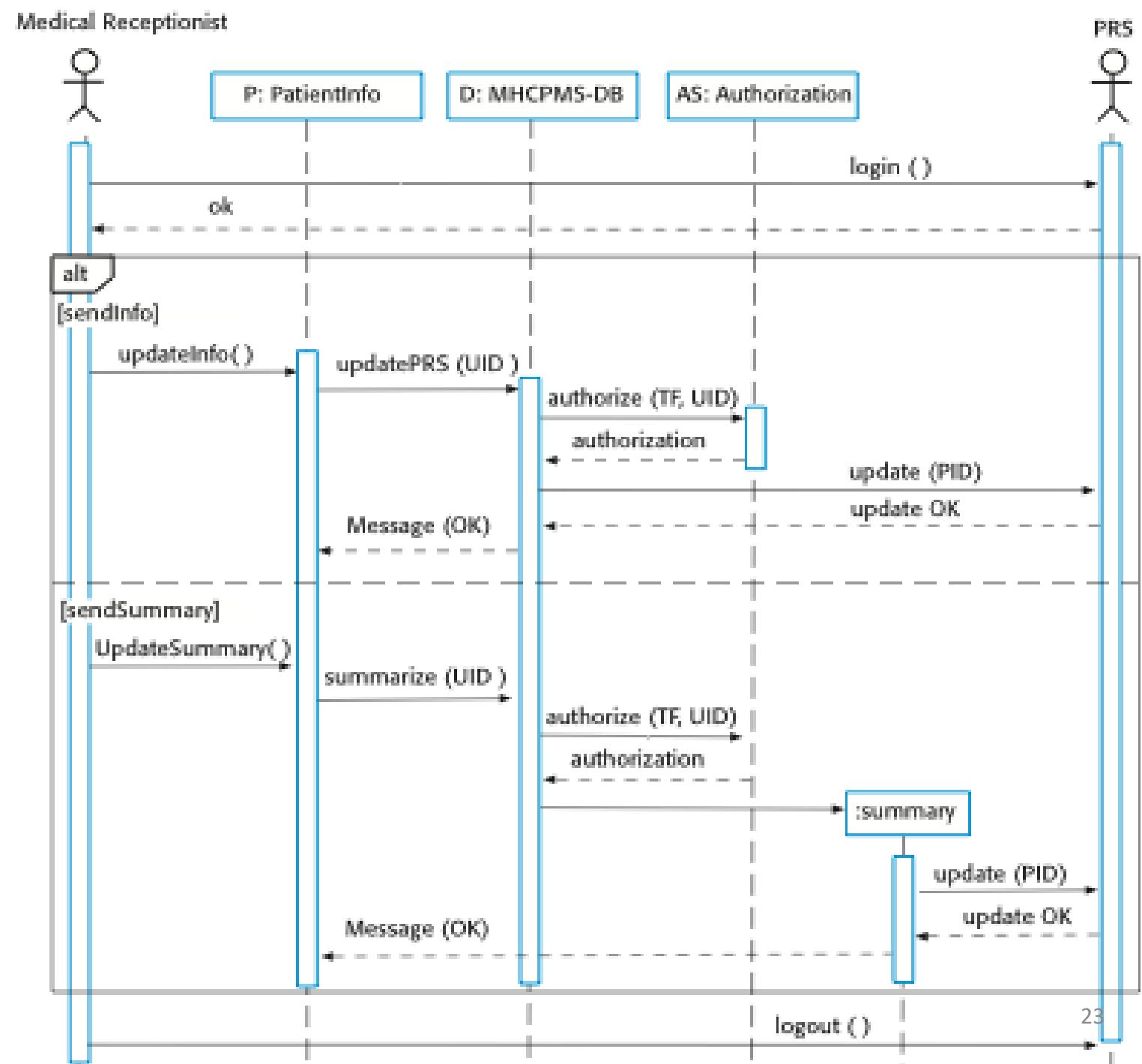
Sequence diagrams

- Sequence diagrams are part of the UML and are used to **model the interactions** between the **actors** and the **objects** **within a system**.
- A sequence diagram shows the sequence of interactions that take place during a particular use case or use case instance.
- The objects and actors involved are listed along the top of the diagram, with a dotted line drawn vertically from these.
- Interactions between objects are indicated by annotated arrows.

Sequence diagram for View patient information



Sequence diagram for Transfer Data



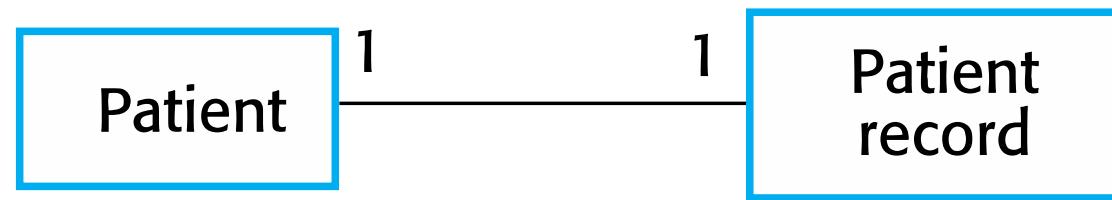
Structural models

- Structural models of software display the organization of a system in terms of the components that make up that system and their relationships.
- Structural models may be static models, which show the structure of the system design, or dynamic models, which show the organization of the system when it is executing.
- You create structural models of a system when **discussing and designing the system architecture**.

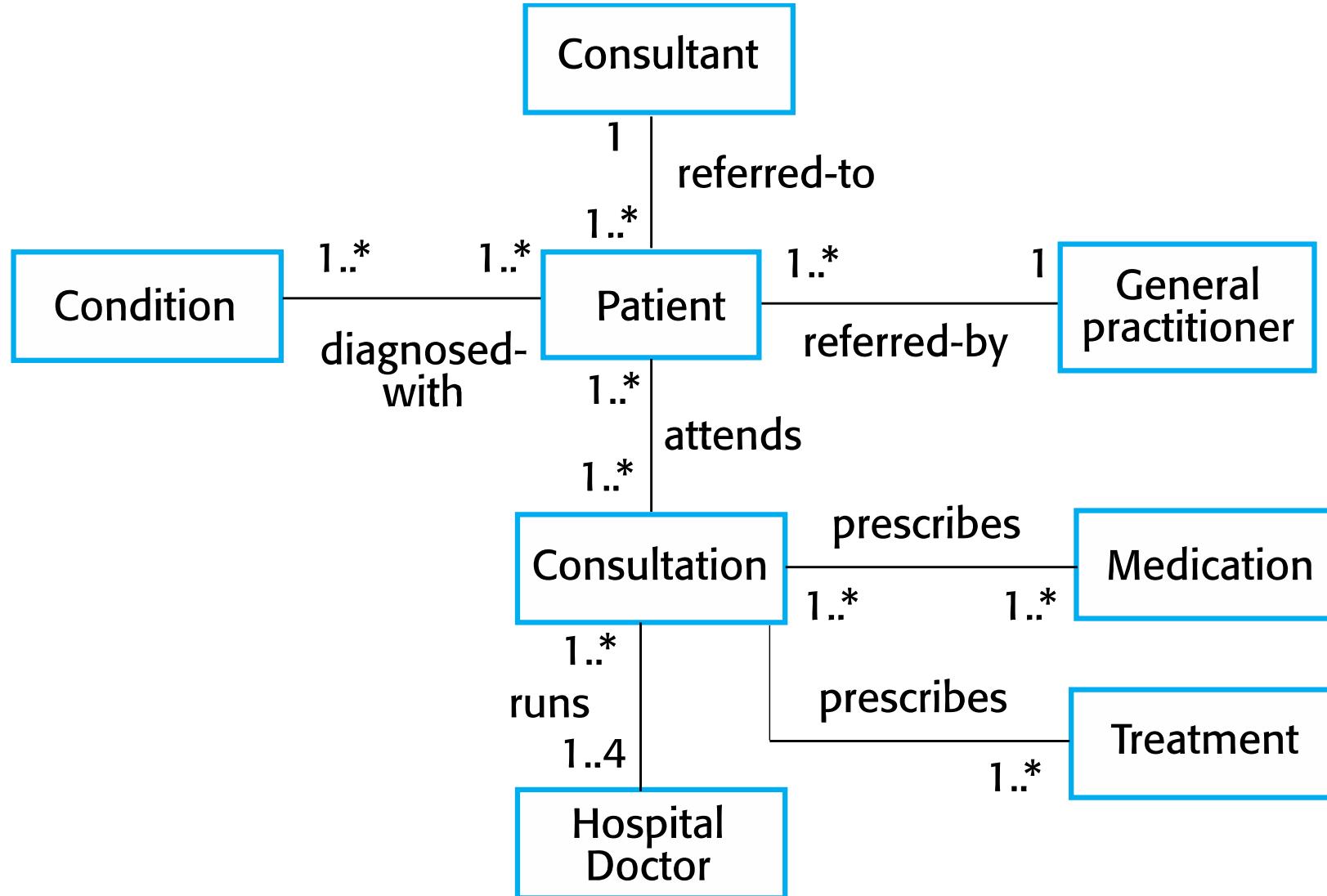
Class diagrams

- 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.

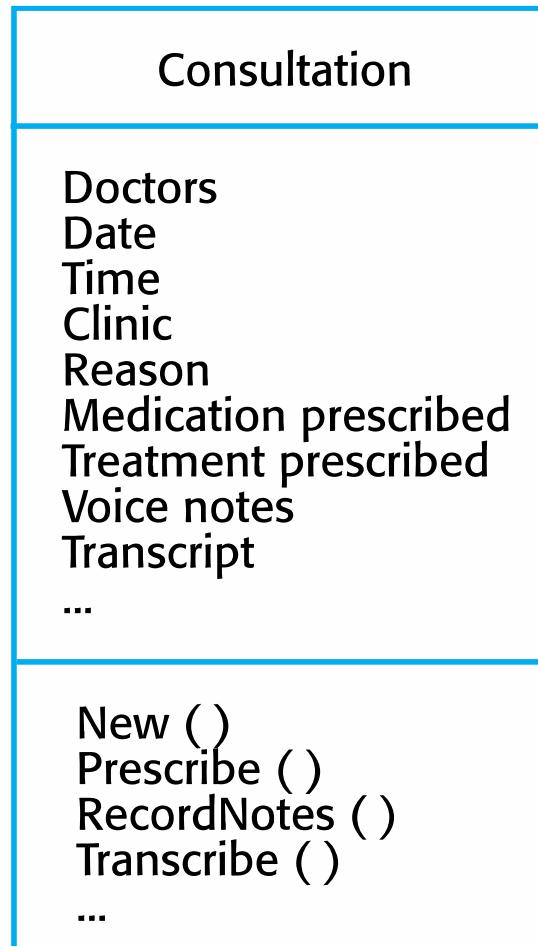
UML classes and association



Classes and associations in Mentcare system



The Consultation class



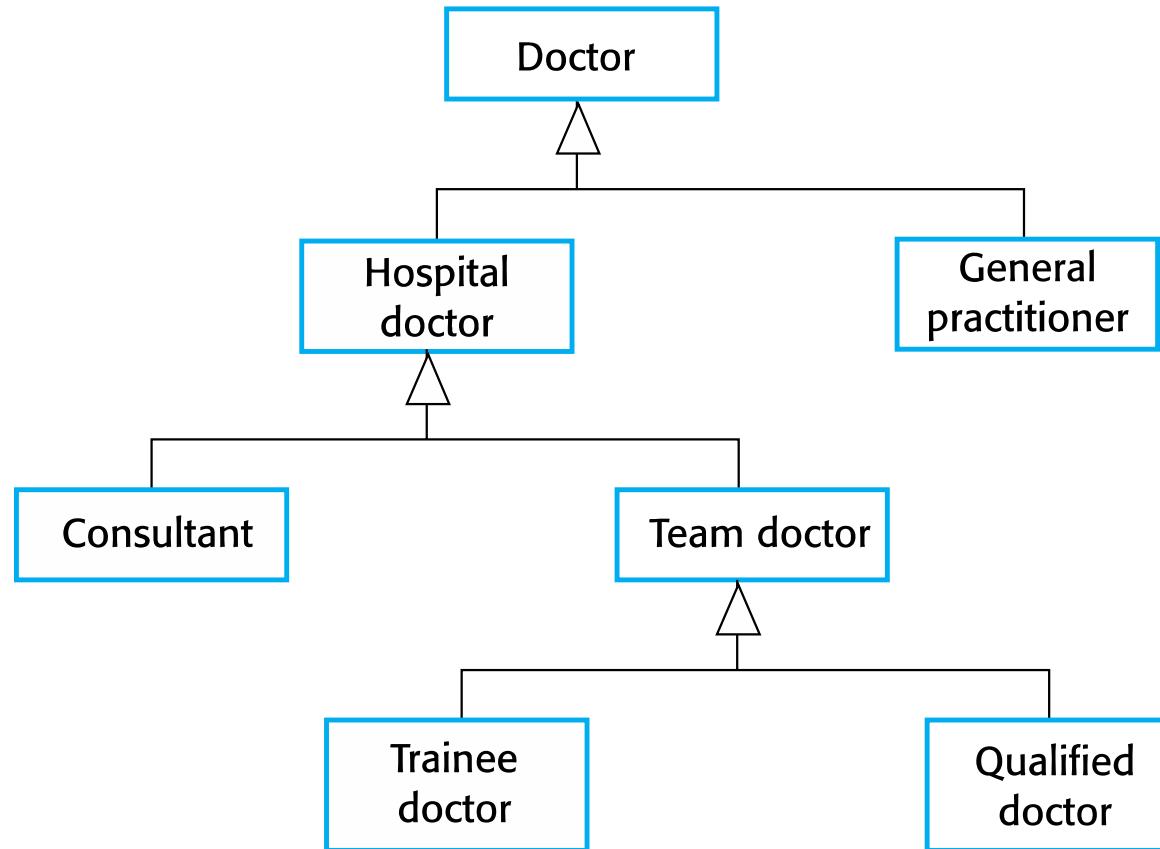
Generalization

- 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.

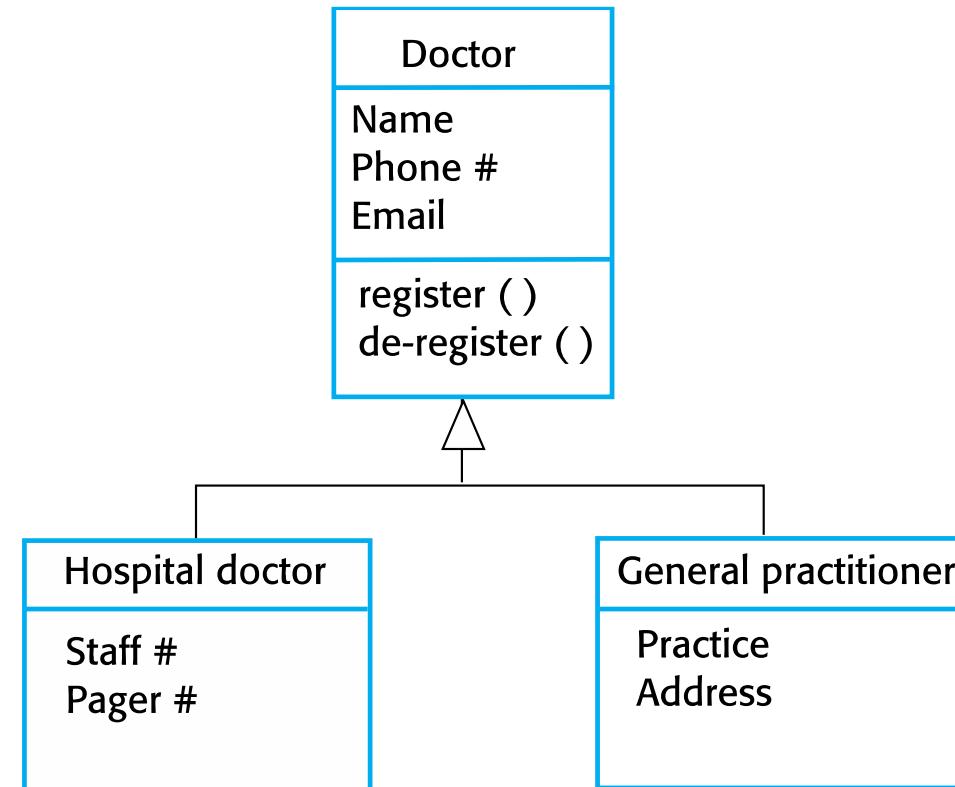
Generalization

- 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 (i.e. subclasses) inherit the attributes and operations from their super classes. These lower-level classes then add more specific attributes and operations.

A generalization hierarchy

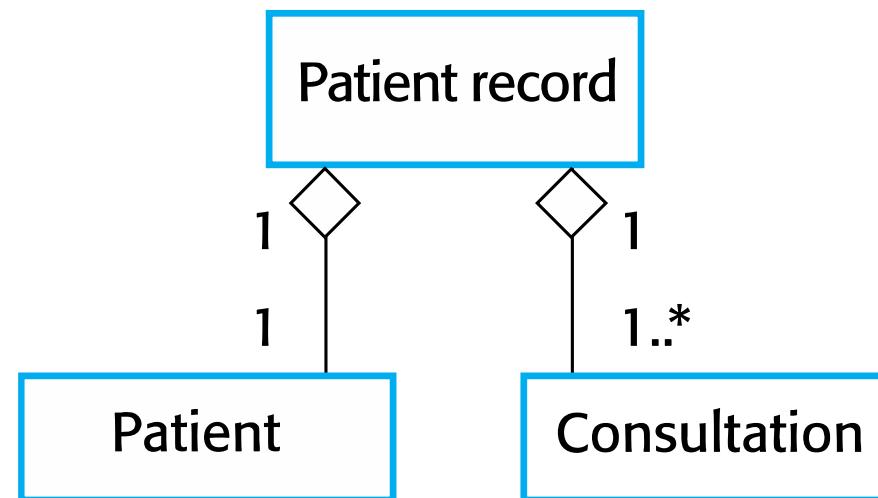


A generalization hierarchy with added detail



Object class aggregation models

- An aggregation model shows how classes that are collections are composed of other classes.
- Aggregation models are similar to the part-of relationship in semantic data models.



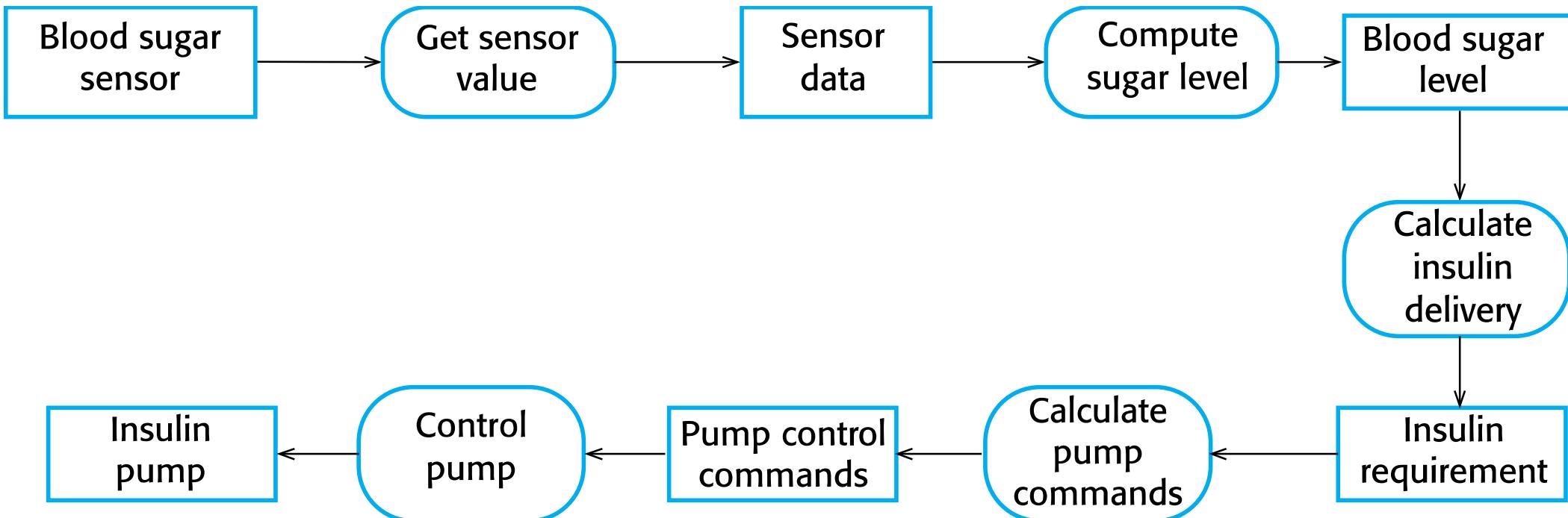
Behavioral models

- Models of the dynamic behavior of a system as it is executing.
- 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)

Data-driven modeling

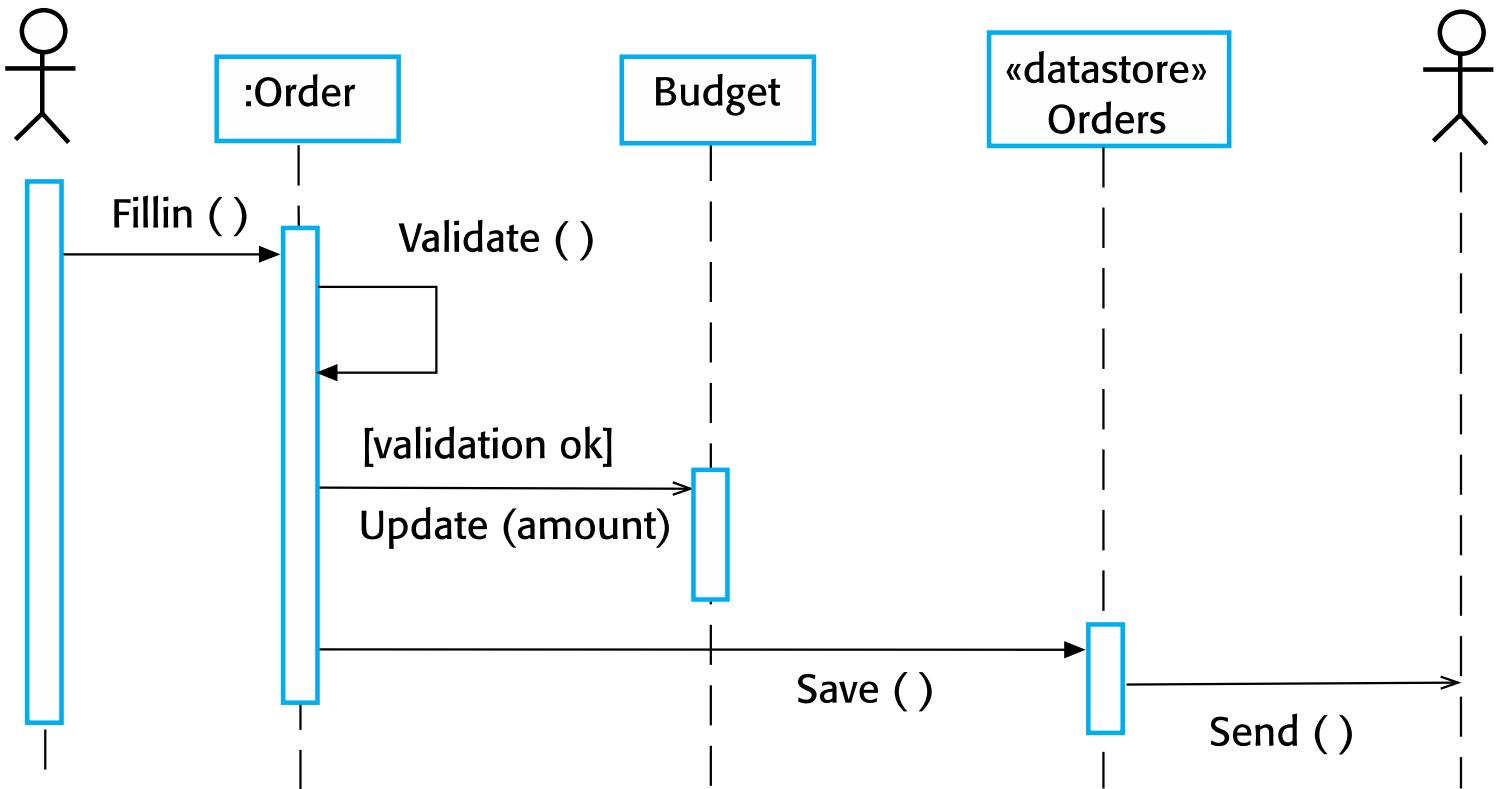
- Many business systems are **data-processing systems** that are primarily driven by data. They are controlled by the data input to the system, with relatively little external event processing.
- Data-driven models show the sequence of actions involved in processing input data and generating an associated output.
 - Data-flow diagrams
 - UML Activity diagrams
- They are particularly useful during the analysis of requirements as they can be used to show end-to-end processing in a system.

An activity model of the insulin pump's operation



Order processing

Purchase officer



Can also be shown as sequence diagrams.

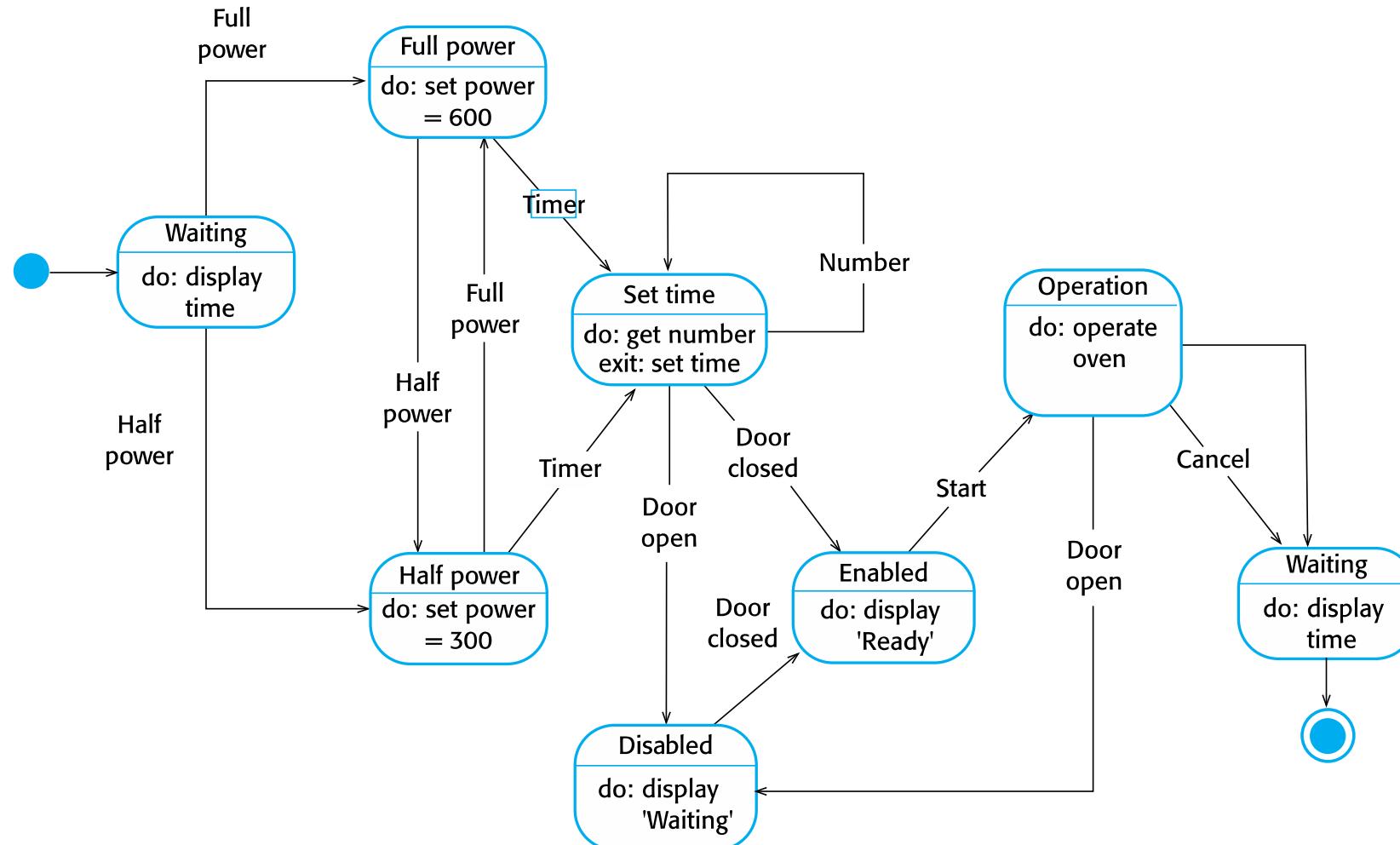
Event-driven modeling

- **Real-time systems** are often event-driven, with minimal data processing.
 - E.g. a landline phone switching system responds to events such as ‘receiver off hook’ by generating a dial tone.
- Event-driven modeling shows how a system responds to external and internal events.
- It assumes that a system has a finite number of states and that events (stimuli) may cause a transition from one state to another.

State machine models

- These model the behaviour of the system in response to external and internal events.
- They show the system's responses to stimuli so are often used for modelling real-time systems.
- State machine models show system states as nodes and events as arcs between these nodes. When an event occurs, the system moves from one state to another.
- Statecharts are an integral part of the UML and are used to represent state machine models.

State diagram of a microwave oven



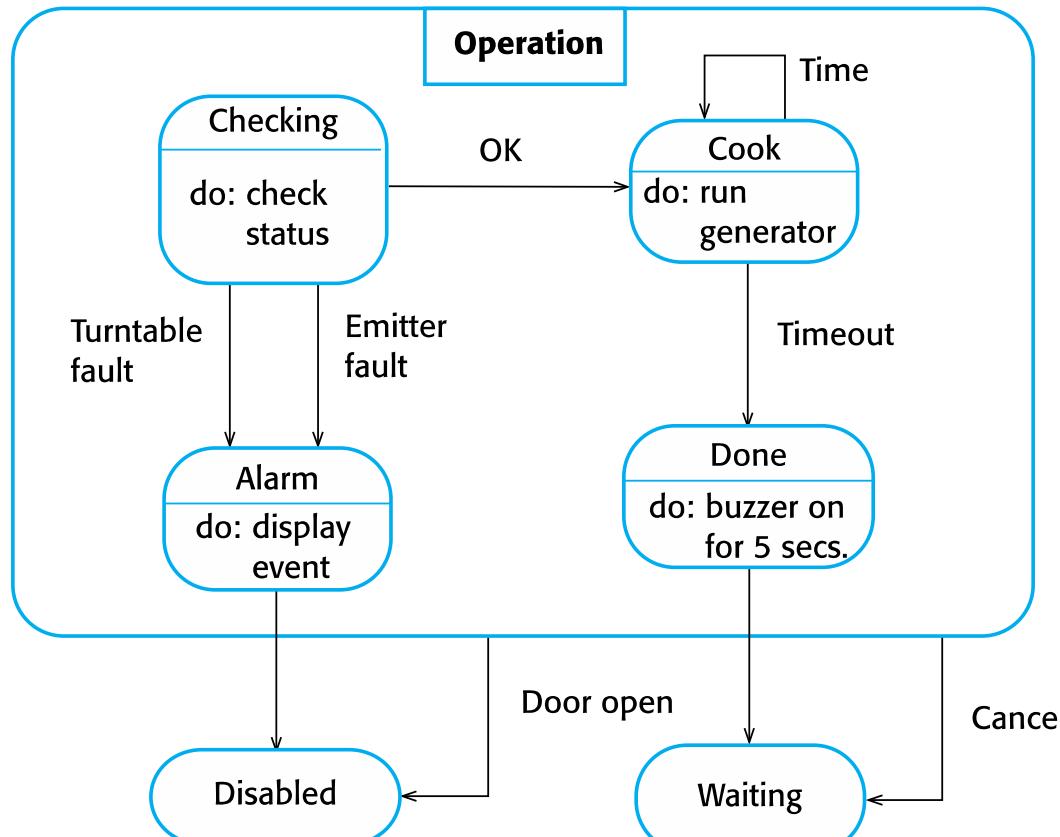
States and stimuli for the microwave oven (a)

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



"Superstates" – expanded in separate diagrams

Model-driven engineering

- MDE is an approach to software development where **models** rather than programs **are the principal outputs of the development process**.
- The programs that execute on a hardware/software platform are then generated automatically from the models.
- Proponents of MDE argue that this raises the level of abstraction in software engineering so that engineers no longer have to be concerned with programming language details or the specifics of execution platforms.

Usage of model-driven engineering

- Model-driven engineering is still at an early stage of development, and it is unclear whether it will have a significant effect on software engineering practice.
- Pros
 - Allows systems to be considered at higher levels of abstraction
 - Generating code automatically means that it is cheaper to adapt systems to new platforms.
- Cons
 - Models for abstraction and not necessarily right for implementation.
 - Savings from generating code may be outweighed by the costs of developing translators for new platforms.

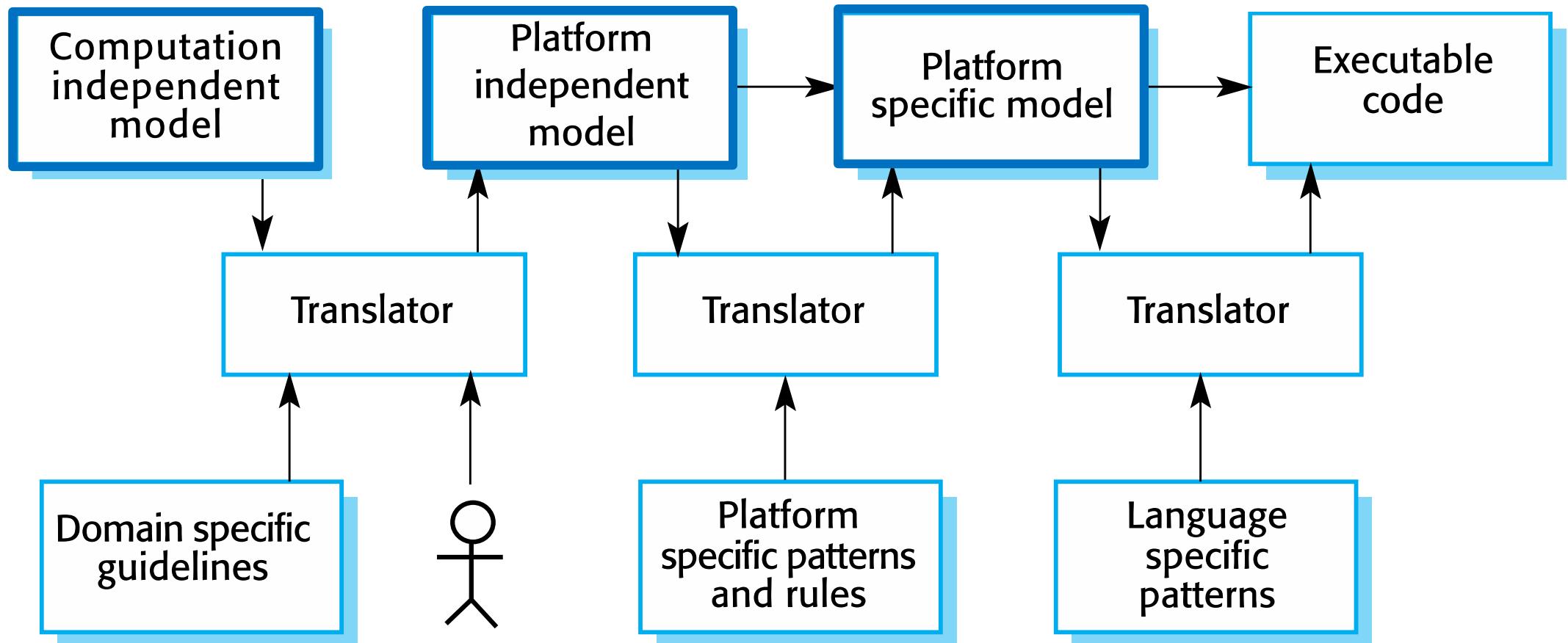
Model driven architecture

- MDA was the precursor of more general model-driven engineering
- MDA is a model-focused approach to software design and implementation that uses a subset of UML models to describe a system.
- Models at different levels of abstraction are created. From a high-level, platform independent model, it is possible, in principle, to generate a working program without manual intervention.

Types of model

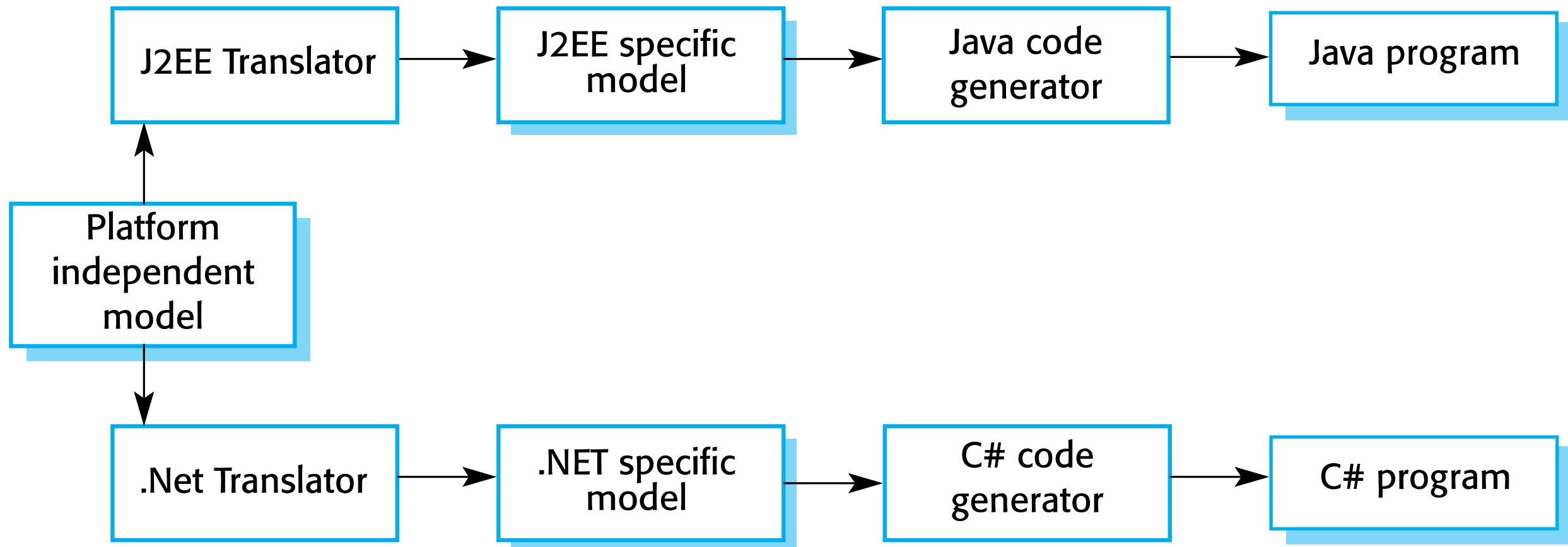
- A computation independent model (CIM)
 - These model the important domain abstractions used in a system. CIMs are sometimes called domain models.
- A platform independent model (PIM)
 - These model the operation of the system without reference to its implementation. The PIM is usually described using UML models that show the static system structure and how it responds to external and internal events.
- Platform specific models (PSM)
 - These are transformations of the platform-independent model with a separate PSM for each application platform. In principle, there may be layers of PSM, with each layer adding some platform-specific detail.

MDA transformations



Multiple platform-specific models

Example



Executable UML

- The fundamental notion behind model-driven engineering is that completely automated transformation of models to code should be possible.
- This is possible using a subset of UML 2, called Executable UML or xUML.

Features of executable UML

- To create an executable subset of UML, the number of model types has been dramatically reduced to these 3 key types:
 - **Domain models** that identify the principal concerns in a system. They are defined using UML class diagrams and include objects, attributes and associations.
 - **Class models** in which classes are defined, along with their attributes and operations.
 - **State models** in which a state diagram is associated with each class and is used to describe the life cycle of the class.
- The dynamic behaviour of the system may be specified declaratively using the **object constraint language (OCL)** or may be expressed using UML's action language.

Key points

- A model is an abstract view of a system that ignores system details. Complementary system models can be developed to show the system's context, interactions, structure and behavior.
- Context models show how a system that is being modeled is positioned in an environment with other systems and processes.
- Use case diagrams and sequence diagrams are used to describe the interactions between users and systems in the system being designed. Use cases describe interactions between a system and external actors; sequence diagrams add more information to these by showing interactions between system objects.
- Structural models show the organization and architecture of a system. Class diagrams are used to define the static structure of classes in a system and their associations.

Key points

- Behavioral models are used to describe the dynamic behavior of an executing system. This behavior can be modeled from the perspective of the data processed by the system, or by the events that stimulate responses from a system.
- Activity diagrams may be used to model the processing of data, where each activity represents one process step.
- State diagrams are used to model a system's behavior in response to internal or external events.
- Model-driven engineering is an approach to software development in which a system is represented as a set of models that can be automatically transformed to executable code.



Chapter 6 – Architectural Design

Topics covered



- ✧ Architectural design decisions
- ✧ Architectural views
- ✧ Architectural patterns
- ✧ Application architectures

Architectural design



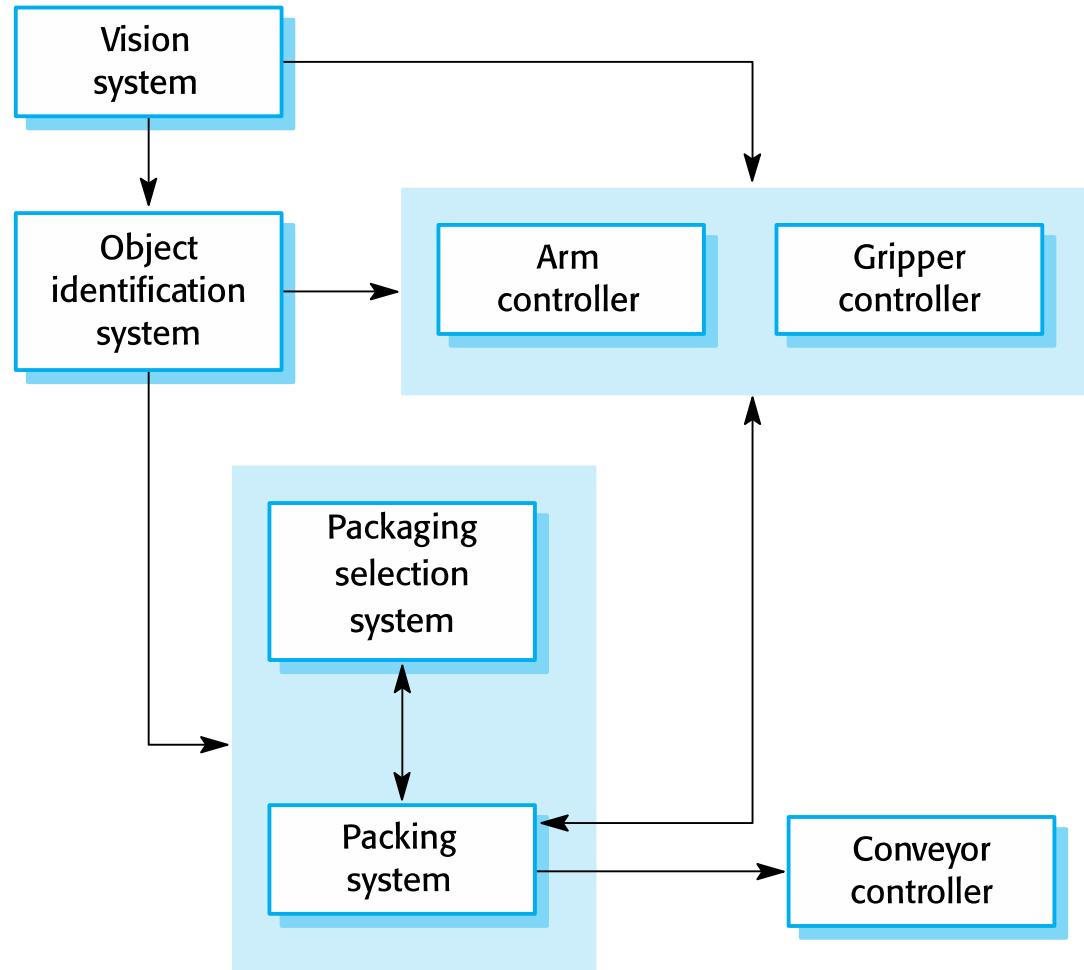
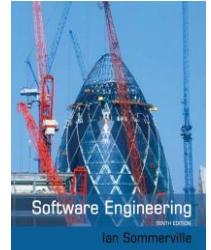
- ✧ Architectural design is concerned with understanding how a software system should be organized and designing the overall structure of that system.
- ✧ Architectural design is the critical link between design and requirements engineering, as it identifies the main structural components in a system and the relationships between them.
- ✧ The output of the architectural design process is an architectural model that describes how the system is organized as a set of communicating components.

Agility and architecture



- ✧ It is generally accepted that an early stage of agile processes is to design an overall systems architecture.
- ✧ Refactoring the system architecture is usually expensive because it affects so many components in the system

The architecture of a packing robot control system



Architectural abstraction



- ✧ Architecture in the small is concerned with the architecture of individual programs. At this level, we are concerned with the way that an individual program is decomposed into components.
- ✧ Architecture in the large is concerned with the architecture of complex enterprise systems that include other systems, programs, and program components. These enterprise systems are distributed over different computers, which may be owned and managed by different companies.

Advantages of explicit architecture



- ✧ Stakeholder communication
 - Architecture may be used as a focus of discussion by system stakeholders.
- ✧ System analysis
 - Means that analysis of whether the system can meet its non-functional requirements is possible.
- ✧ Large-scale reuse
 - The architecture may be reusable across a range of systems
 - Product-line architectures may be developed.

Architectural representations



- ✧ Simple, informal block diagrams showing entities and relationships are the most frequently used method for documenting software architectures.
- ✧ But these have been criticised because they lack semantics, do not show the types of relationships between entities nor the visible properties of entities in the architecture.
- ✧ Depends on the use of architectural models. The requirements for model semantics depends on how the models are used.

Box and line diagrams



- ✧ Very abstract - they do not show the nature of component relationships nor the externally visible properties of the sub-systems.
- ✧ However, useful for communication with stakeholders and for project planning.

Use of architectural models



- ✧ As a way of facilitating discussion about the system design
 - A high-level architectural view of a system is useful for communication with system stakeholders and project planning because it is not cluttered with detail. Stakeholders can relate to it and understand an abstract view of the system. They can then discuss the system as a whole without being confused by detail.
- ✧ As a way of documenting an architecture that has been designed
 - The aim here is to produce a complete system model that shows the different components in a system, their interfaces and their connections.



Architectural design decisions

Architectural design decisions



- ✧ Architectural design is a creative process so the process differs depending on the type of system being developed.
- ✧ However, a number of common decisions span all design processes and these decisions affect the non-functional characteristics of the system.

Architectural design decisions



Is there a generic application architecture that can act as a template for the system that is being designed?

How will the system be distributed across hardware cores or processors?

What architectural patterns or styles might be used?

What will be the fundamental approach used to structure the system?

What strategy will be used to control the operation of the components in the system?

How will the structural components in the system be decomposed into sub-components?

What architectural organization is best for delivering the non-functional requirements of the system?

How should the architecture of the system be documented?

Architecture reuse



- ✧ Systems in the same domain often have similar architectures that reflect domain concepts.
- ✧ Application product lines are built around a core architecture with variants that satisfy particular customer requirements.
- ✧ The architecture of a system may be designed around one of more architectural patterns or 'styles'.
 - These capture the essence of an architecture and can be instantiated in different ways.

Architecture and system characteristics



- ✧ Performance
 - Localise critical operations and minimise communications. Use large rather than fine-grain components.
- ✧ Security
 - Use a layered architecture with critical assets in the inner layers.
- ✧ Safety
 - Localise safety-critical features in a small number of subsystems.
- ✧ Availability
 - Include redundant components and mechanisms for fault tolerance.
- ✧ Maintainability
 - Use fine-grain, replaceable components.



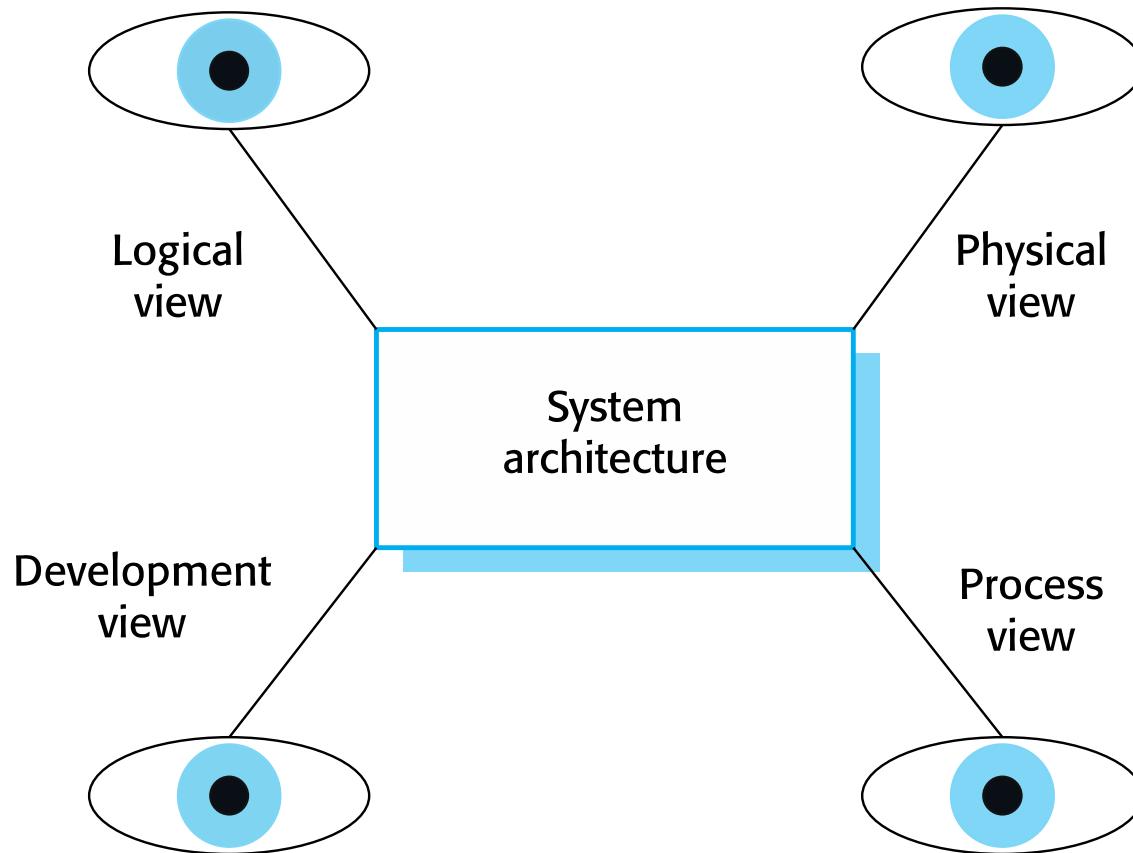
Architectural views

Architectural views



- ✧ What views or perspectives are useful when designing and documenting a system's architecture?
- ✧ What notations should be used for describing architectural models?
- ✧ Each architectural model only shows one view or perspective of the system.
 - It might show how a system is decomposed into modules, how the run-time processes interact or the different ways in which system components are distributed across a network. For both design and documentation, you usually need to present multiple views of the software architecture.

Architectural views



4 + 1 view model of software architecture



- ✧ A logical view, which shows the key abstractions in the system as objects or object classes.
- ✧ A process view, which shows how, at run-time, the system is composed of interacting processes.
- ✧ A development view, which shows how the software is decomposed for development.
- ✧ A physical view, which shows the system hardware and how software components are distributed across the processors in the system.
- ✧ Related using use cases or scenarios (+1)

Representing architectural views



- ✧ Some people argue that the Unified Modeling Language (UML) is an appropriate notation for describing and documenting system architectures
- ✧ I disagree with this as I do not think that the UML includes abstractions appropriate for high-level system description.
- ✧ Architectural description languages (ADLs) have been developed but are not widely used



Architectural patterns

Architectural patterns



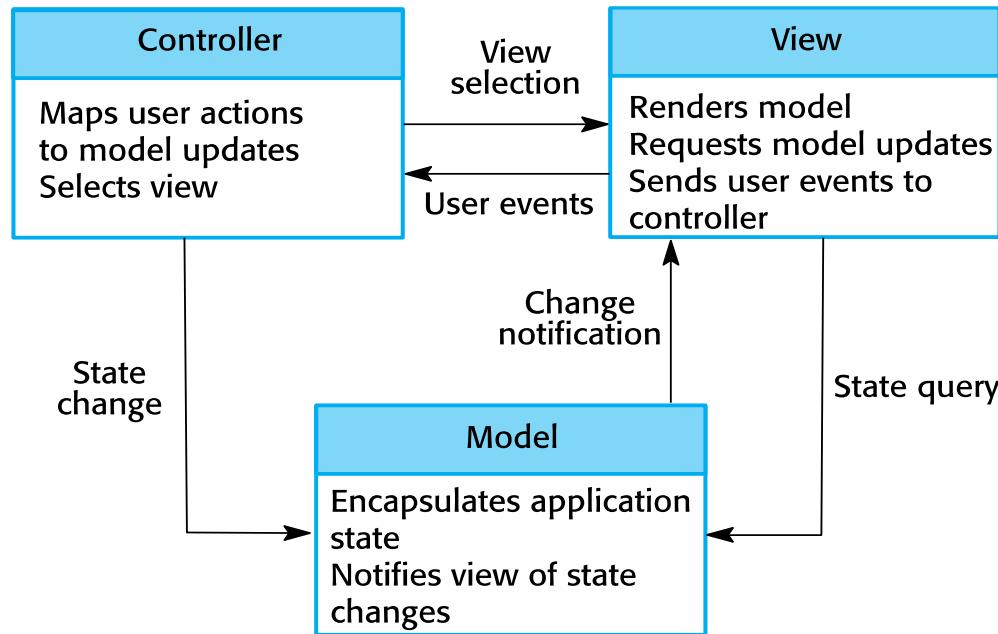
- ✧ Patterns are a means of representing, sharing and reusing knowledge.
- ✧ An architectural pattern is a stylized description of good design practice, which has been tried and tested in different environments.
- ✧ Patterns should include information about when they are and when they are not useful.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

The Model-View-Controller (MVC) pattern

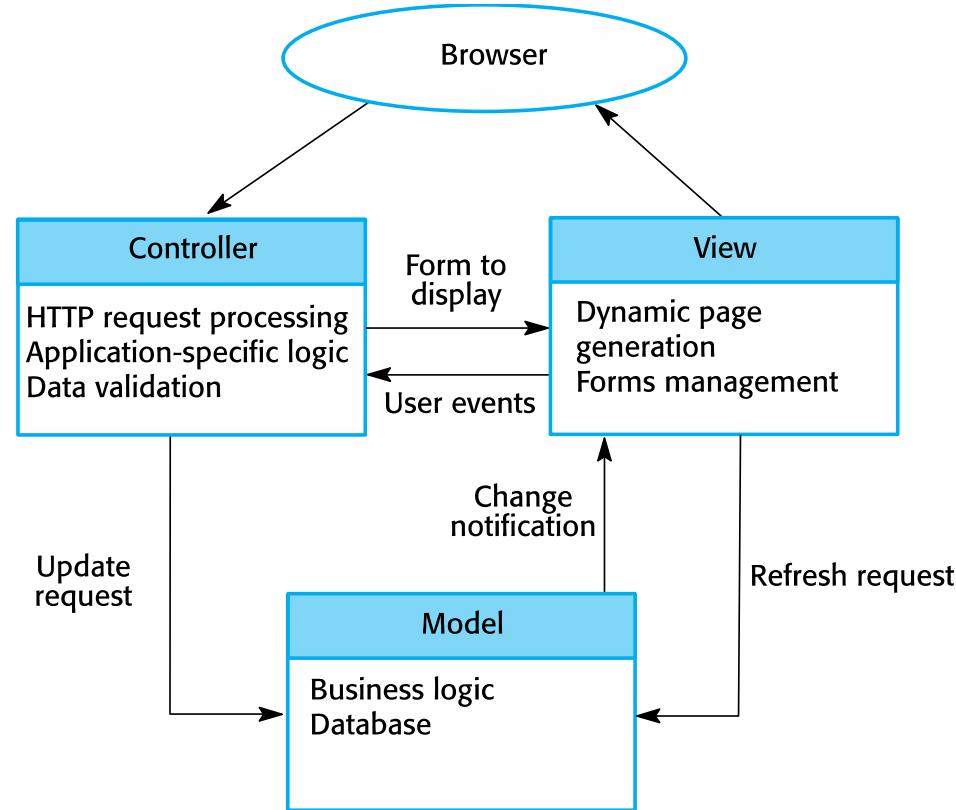


Name	MVC (Model-View-Controller)
Description	Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines 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



Layered architecture



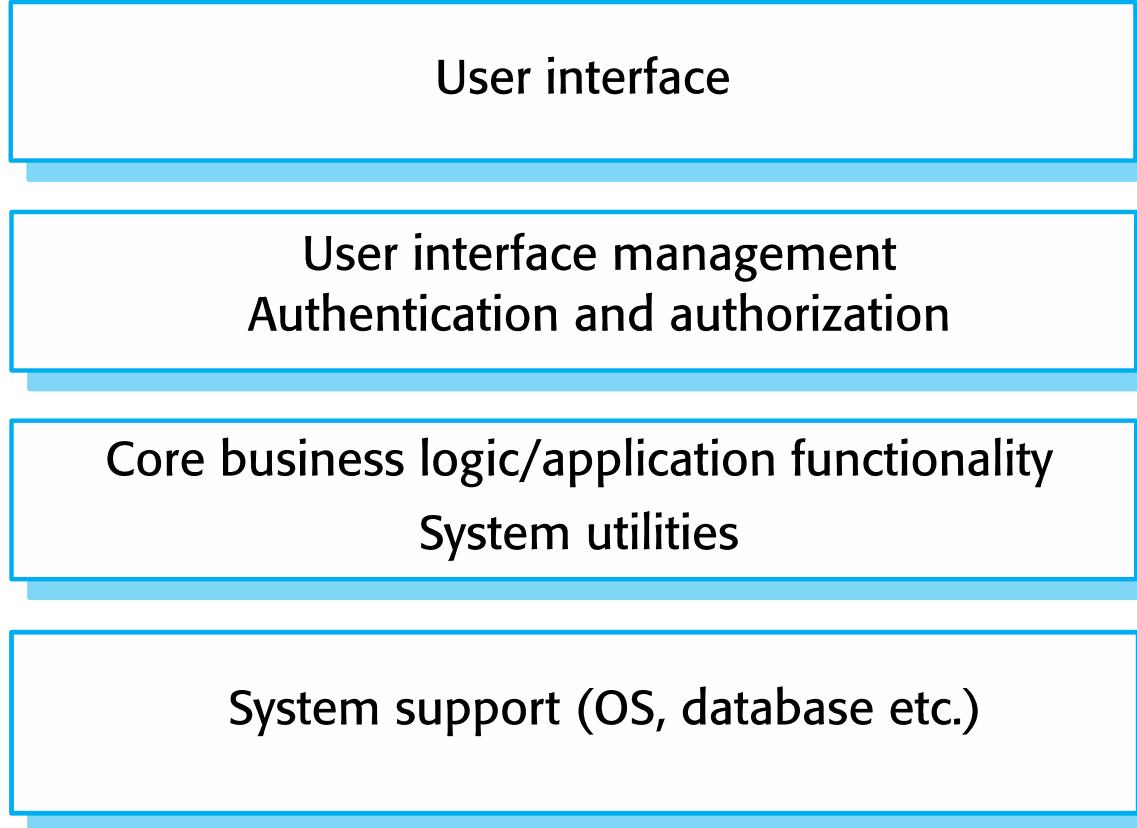
- ✧ Used to model the interfacing of sub-systems.
- ✧ Organises the system into a set of layers (or abstract machines) each of which provide a set of services.
- ✧ Supports the incremental development of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected.
- ✧ However, often artificial to structure systems in this way.

The Layered architecture pattern



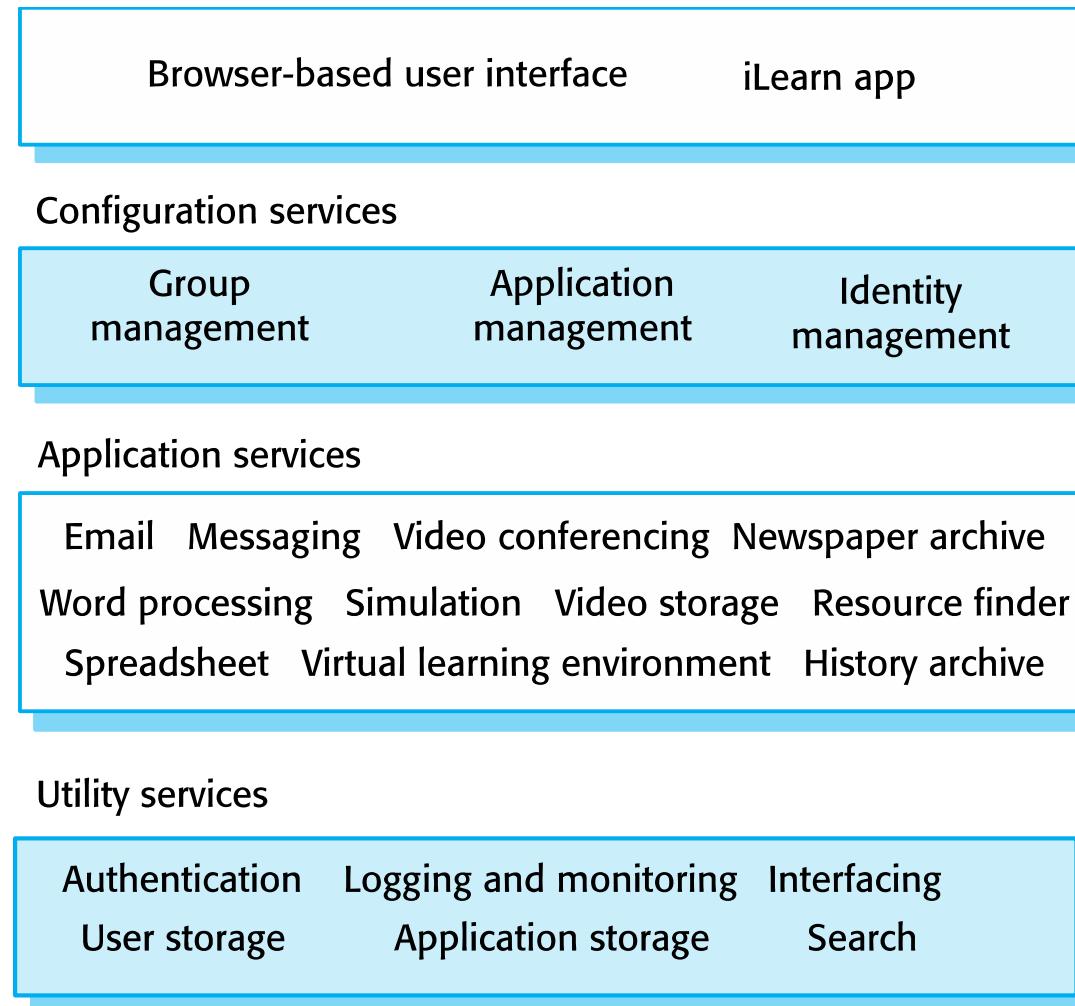
Name	Layered architecture
Description	Organizes the system into layers with related functionality associated with each layer. A layer provides services to the layer above it so the lowest-level layers represent core services that are likely to be used throughout the system. See Figure 6.6.
Example	A layered model of a system for sharing copyright documents held in different libraries, as shown in Figure 6.7.
When used	Used when building new facilities on top of existing systems; when the development is spread across several teams with each team responsibility for a layer of functionality; when there is a requirement for multi-level security.
Advantages	Allows replacement of entire layers so long as the interface is maintained. Redundant facilities (e.g., authentication) can be provided in each layer to increase the dependability of the system.
Disadvantages	In practice, providing a clean separation between layers is often difficult and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it. Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.

A generic layered architecture





The architecture of the iLearn system



Repository architecture



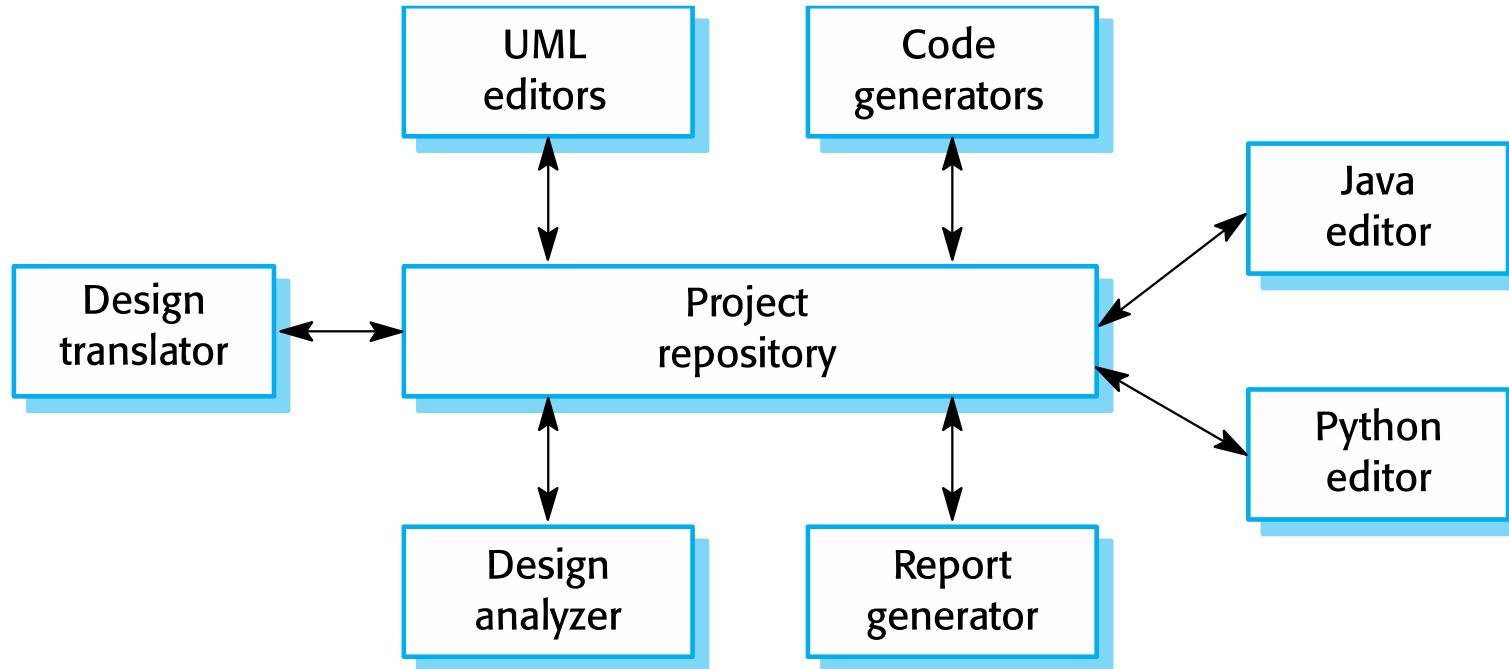
- ✧ Sub-systems must exchange data. This may be done in two ways:
 - Shared data is held in a central database or repository and may be accessed by all sub-systems;
 - Each sub-system maintains its own database and passes data explicitly to other sub-systems.
- ✧ When large amounts of data are to be shared, the repository model of sharing is most commonly used as this is an efficient data sharing mechanism.

The Repository pattern



Name	Repository
Description	All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.
Example	Figure 6.9 is an example of an IDE where the components use a repository of system design information. Each software tool generates information which is then available for use by other tools.
When used	You should use this pattern when you have a system in which large volumes of information are generated that has to be stored for a long time. You may also use it in data-driven systems where the inclusion of data in the repository triggers an action or tool.
Advantages	Components can be independent—they do not need to know of the existence of other components. Changes made by one component can be propagated to all components. All data can be managed consistently (e.g., backups done at the same time) as it is all in one place.
Disadvantages	The repository is a single point of failure so problems in the repository affect the whole system. May be inefficiencies in organizing all communication through the repository. Distributing the repository across several computers may be difficult.

A repository architecture for an IDE



Client-server architecture



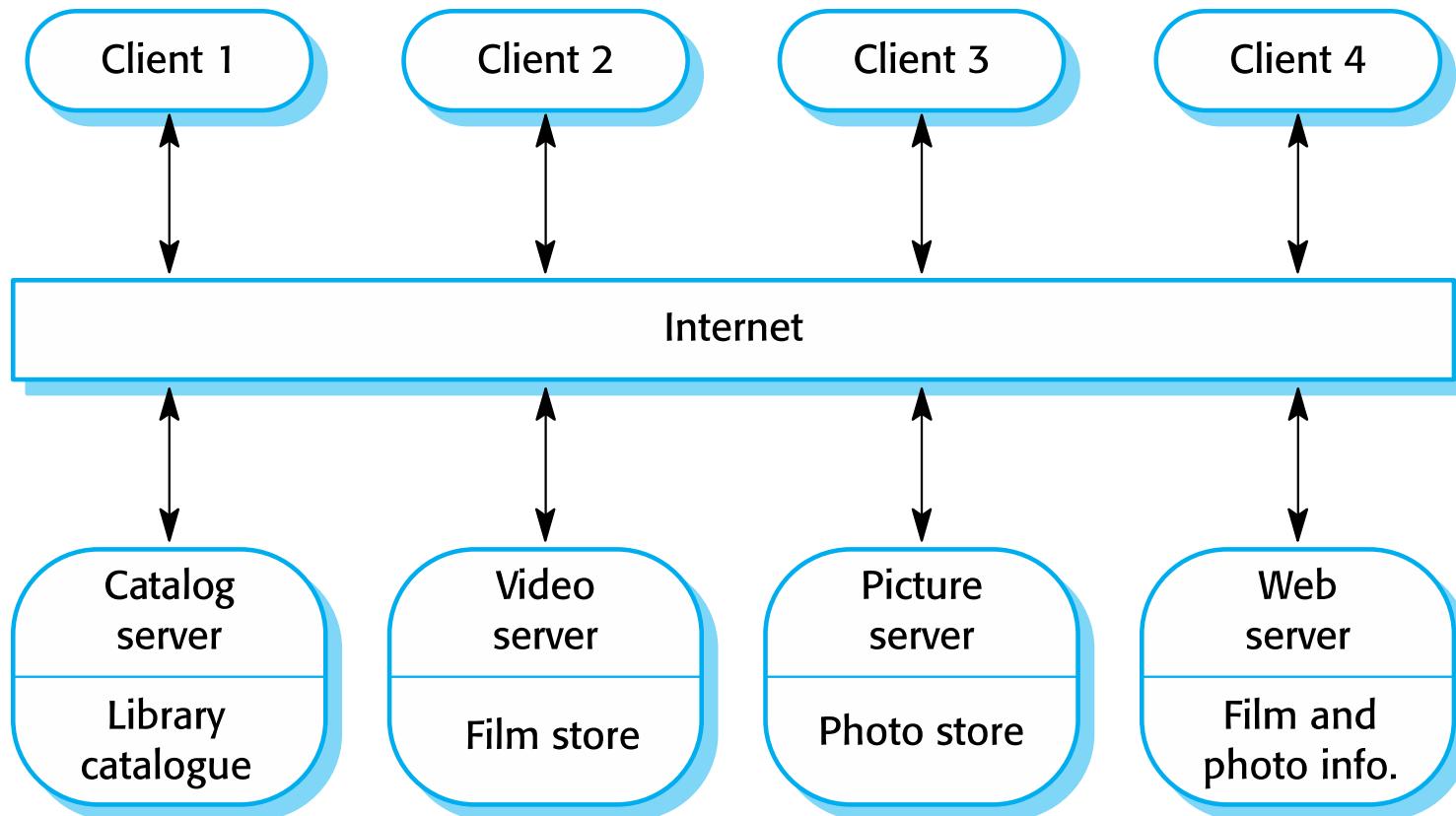
- ✧ Distributed system model which shows how data and processing is distributed across a range of components.
 - Can be implemented on a single computer.
- ✧ Set of stand-alone servers which provide specific services such as printing, data management, etc.
- ✧ Set of clients which call on these services.
- ✧ Network which allows clients to access servers.



The Client–server pattern

Name	Client-server
Description	In a client–server architecture, the functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.
Example	Figure 6.11 is an example of a film and video/DVD library organized as a client–server system.
When used	Used when data in a shared database has to be accessed from a range of locations. Because servers can be replicated, may also be used when the load on a system is variable.
Advantages	The principal advantage of this model is that servers can be distributed across a network. General functionality (e.g., a printing service) can be available to all clients and does not need to be implemented by all services.
Disadvantages	Each service is a single point of failure so susceptible to denial of service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system. May be management problems if servers are owned by different organizations.

A client–server architecture for a film library



Pipe and filter architecture



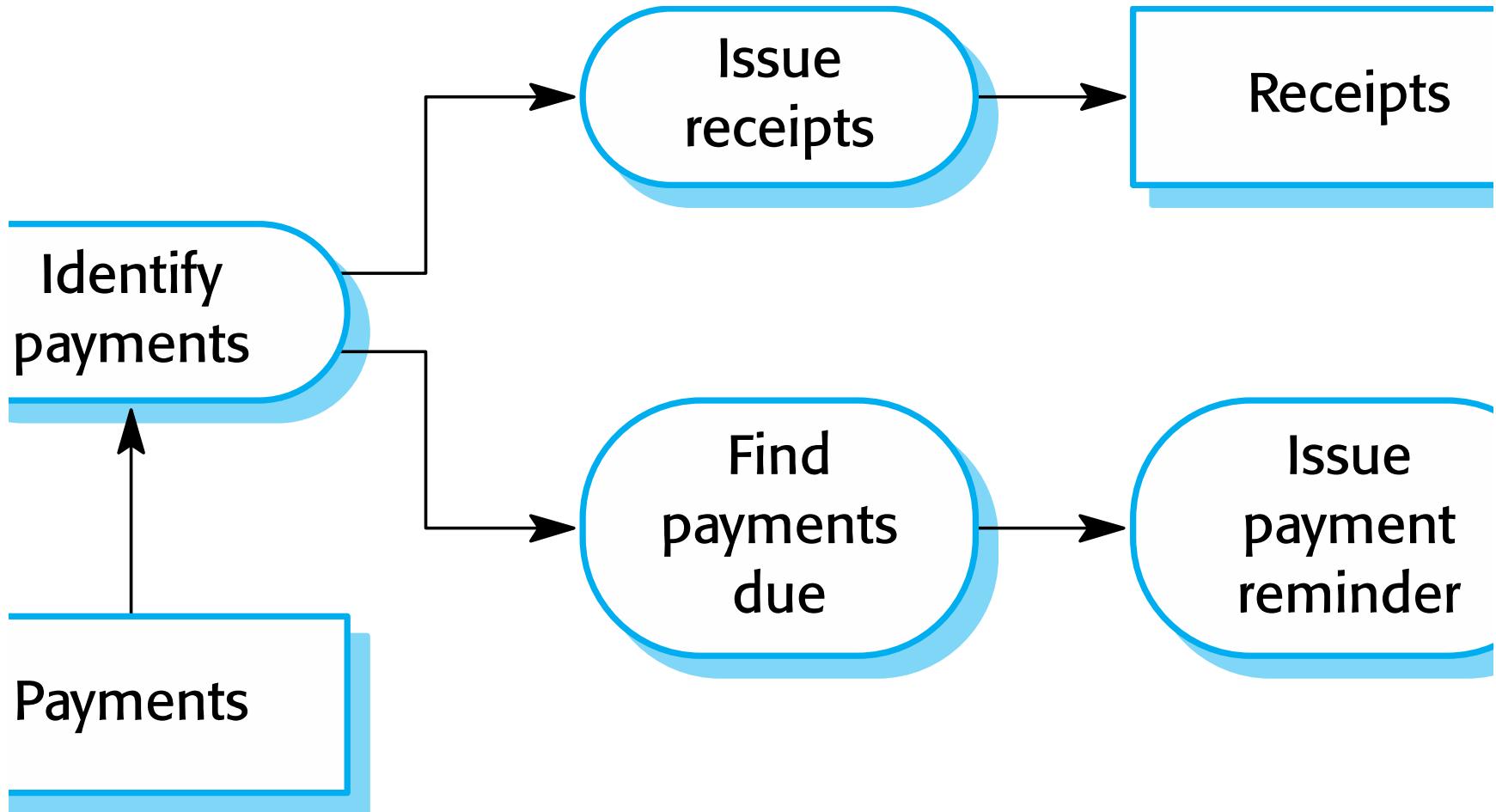
- ✧ Functional transformations process their inputs to produce outputs.
- ✧ May be referred to as a pipe and filter model (as in UNIX shell).
- ✧ Variants of this approach are very common. When transformations are sequential, this is a batch sequential model which is extensively used in data processing systems.
- ✧ Not really suitable for interactive systems.



The pipe and filter pattern

Name	Pipe and filter
Description	The processing of the data in a system is organized so that each processing component (filter) is discrete and carries out one type of data transformation. The data flows (as in a pipe) from one component to another for processing.
Example	Figure 6.13 is an example of a pipe and filter system used for processing invoices.
When used	Commonly used in data processing applications (both batch- and transaction-based) where inputs are processed in separate stages to generate related outputs.
Advantages	Easy to understand and supports transformation reuse. Workflow style matches the structure of many business processes. Evolution by adding transformations is straightforward. Can be implemented as either a sequential or concurrent system.
Disadvantages	The format for data transfer has to be agreed upon between communicating transformations. Each transformation must parse its input and unparse its output to the agreed form. This increases system overhead and may mean that it is impossible to reuse functional transformations that use incompatible data structures.

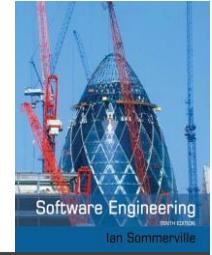
An example of the pipe and filter architecture used in a payments system





Application architectures

Application architectures



- ✧ Application systems are designed to meet an organisational need.
- ✧ As businesses have much in common, their application systems also tend to have a common architecture that reflects the application requirements.
- ✧ A generic application architecture is an architecture for a type of software system that may be configured and adapted to create a system that meets specific requirements.

Use of application architectures



- ✧ As a starting point for architectural design.
- ✧ As a design checklist.
- ✧ As a way of organising the work of the development team.
- ✧ As a means of assessing components for reuse.
- ✧ As a vocabulary for talking about application types.



Examples of application types

- ✧ Data processing applications
 - Data driven applications that process data in batches without explicit user intervention during the processing.
- ✧ Transaction processing applications
 - Data-centred applications that process user requests and update information in a system database.
- ✧ Event processing systems
 - Applications where system actions depend on interpreting events from the system's environment.
- ✧ Language processing systems
 - Applications where the users' intentions are specified in a formal language that is processed and interpreted by the system.



Application type examples

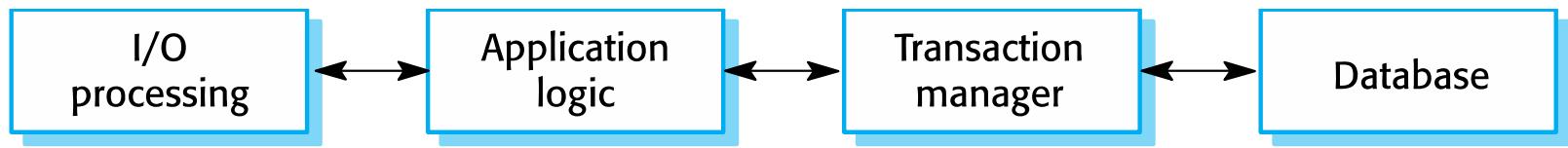
- ✧ Two very widely used generic application architectures are transaction processing systems and language processing systems.
- ✧ Transaction processing systems
 - E-commerce systems;
 - Reservation systems.
- ✧ Language processing systems
 - Compilers;
 - Command interpreters.

Transaction processing systems

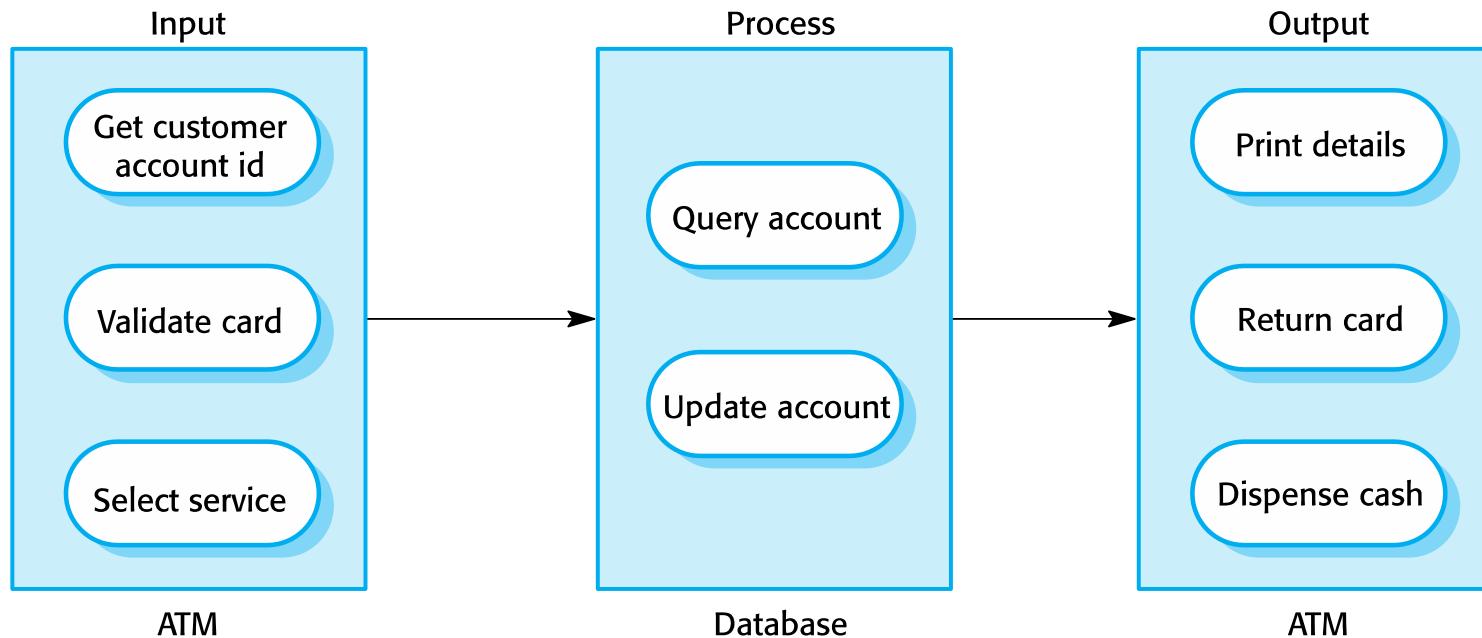


- ✧ Process user requests for information from a database or requests to update the database.
- ✧ From a user perspective a transaction is:
 - Any coherent sequence of operations that satisfies a goal;
 - For example - find the times of flights from London to Paris.
- ✧ Users make asynchronous requests for service which are then processed by a transaction manager.

The structure of transaction processing applications



The software architecture of an ATM system



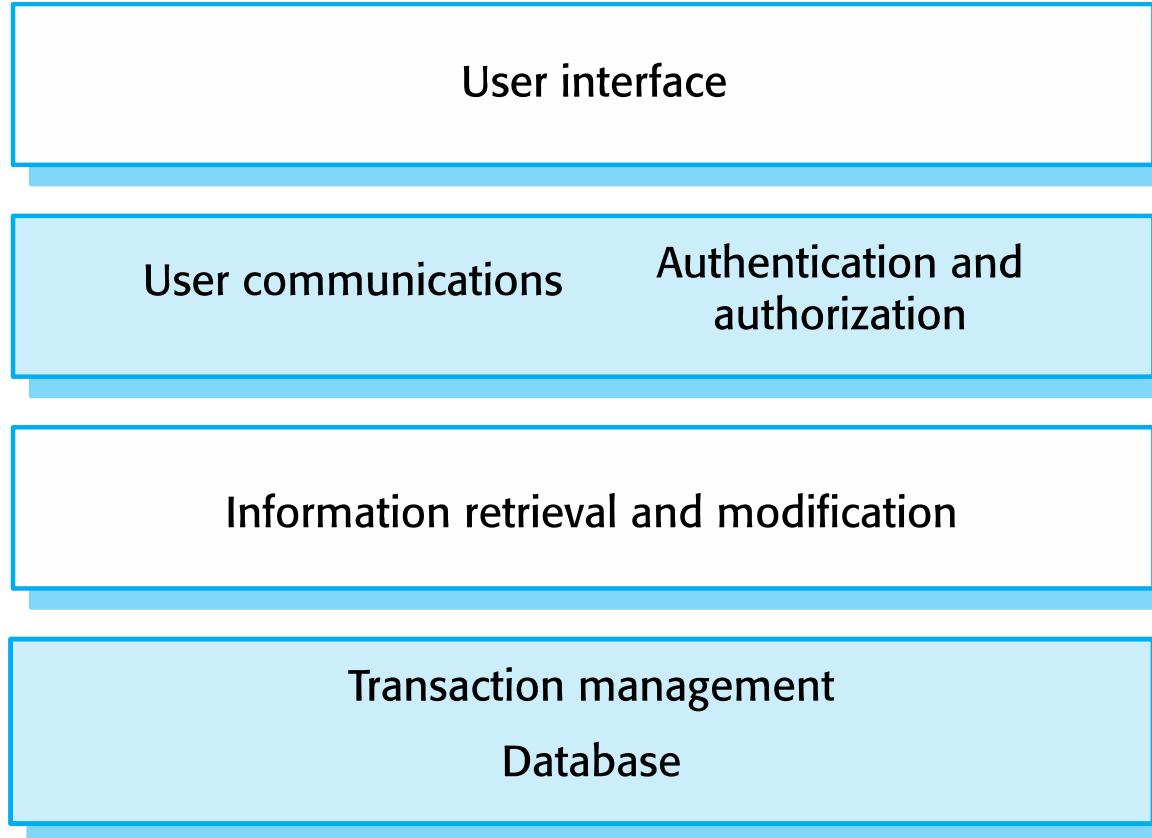
Information systems architecture



- ✧ Information systems have a generic architecture that can be organised as a layered architecture.
- ✧ These are transaction-based systems as interaction with these systems generally involves database transactions.
- ✧ Layers include:
 - The user interface
 - User communications
 - Information retrieval
 - System database

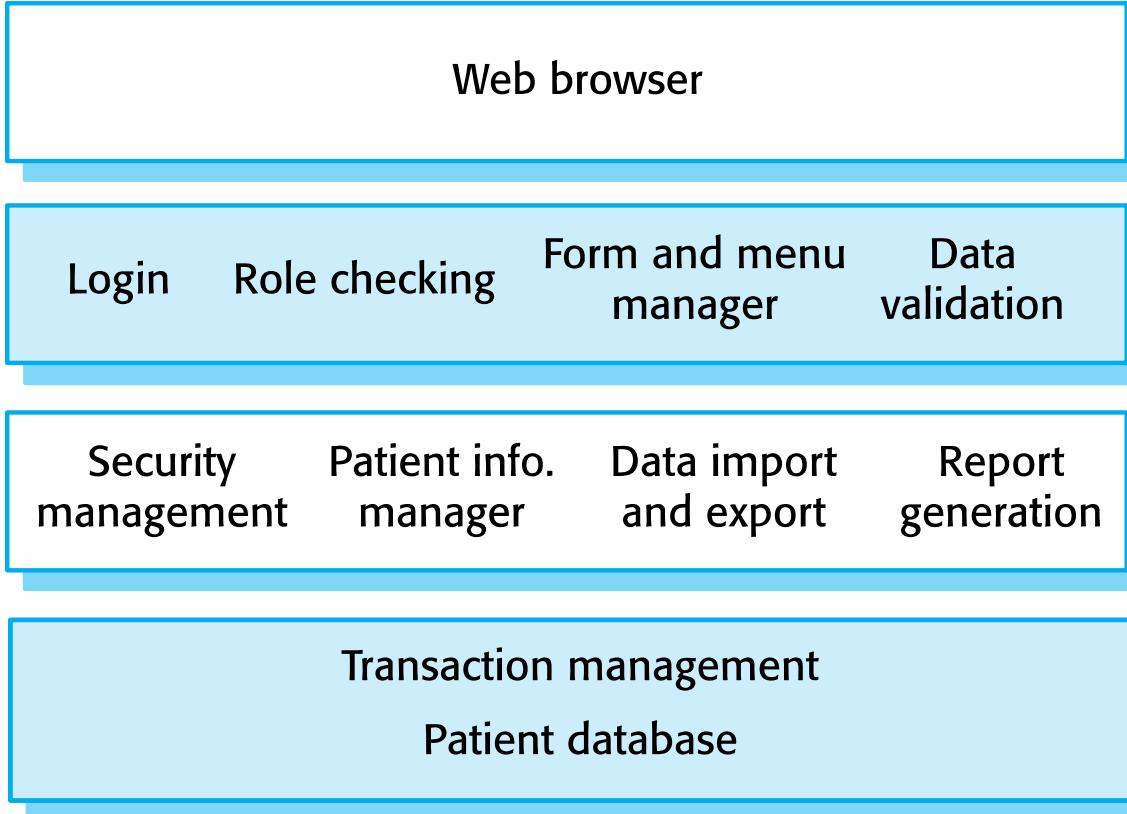


Layered information system architecture





The architecture of the Mentcare system



Web-based information systems



- ✧ Information and resource management systems are now usually web-based systems where the user interfaces are implemented using a web browser.
- ✧ For example, e-commerce systems are Internet-based resource management systems that accept electronic orders for goods or services and then arrange delivery of these goods or services to the customer.
- ✧ In an e-commerce system, the application-specific layer includes additional functionality supporting a 'shopping cart' in which users can place a number of items in separate transactions, then pay for them all together in a single transaction.

Server implementation



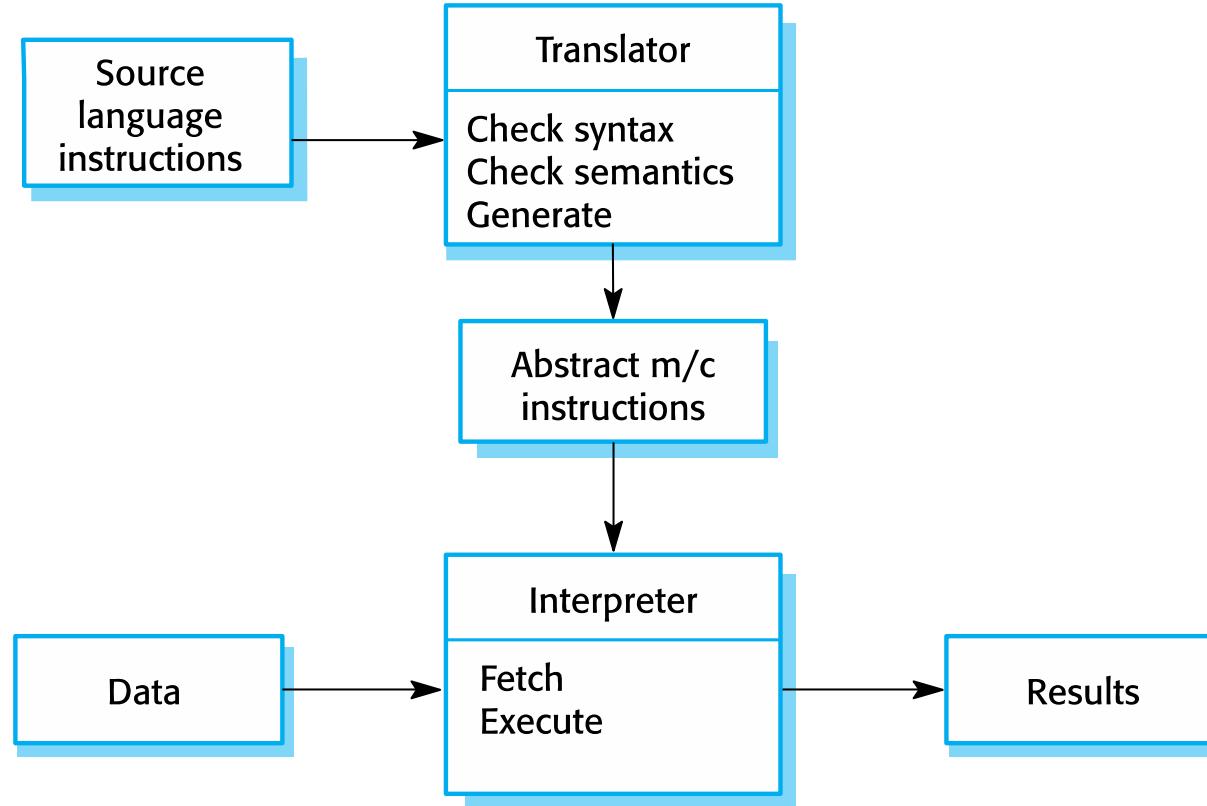
- ✧ These systems are often implemented as multi-tier client server/architectures (discussed in Chapter 17)
 - The web server is responsible for all user communications, with the user interface implemented using a web browser;
 - The application server is responsible for implementing application-specific logic as well as information storage and retrieval requests;
 - The database server moves information to and from the database and handles transaction management.

Language processing systems



- ✧ Accept a natural or artificial language as input and generate some other representation of that language.
- ✧ May include an interpreter to act on the instructions in the language that is being processed.
- ✧ Used in situations where the easiest way to solve a problem is to describe an algorithm or describe the system data
 - Meta-case tools process tool descriptions, method rules, etc and generate tools.

The architecture of a language processing system



Compiler components



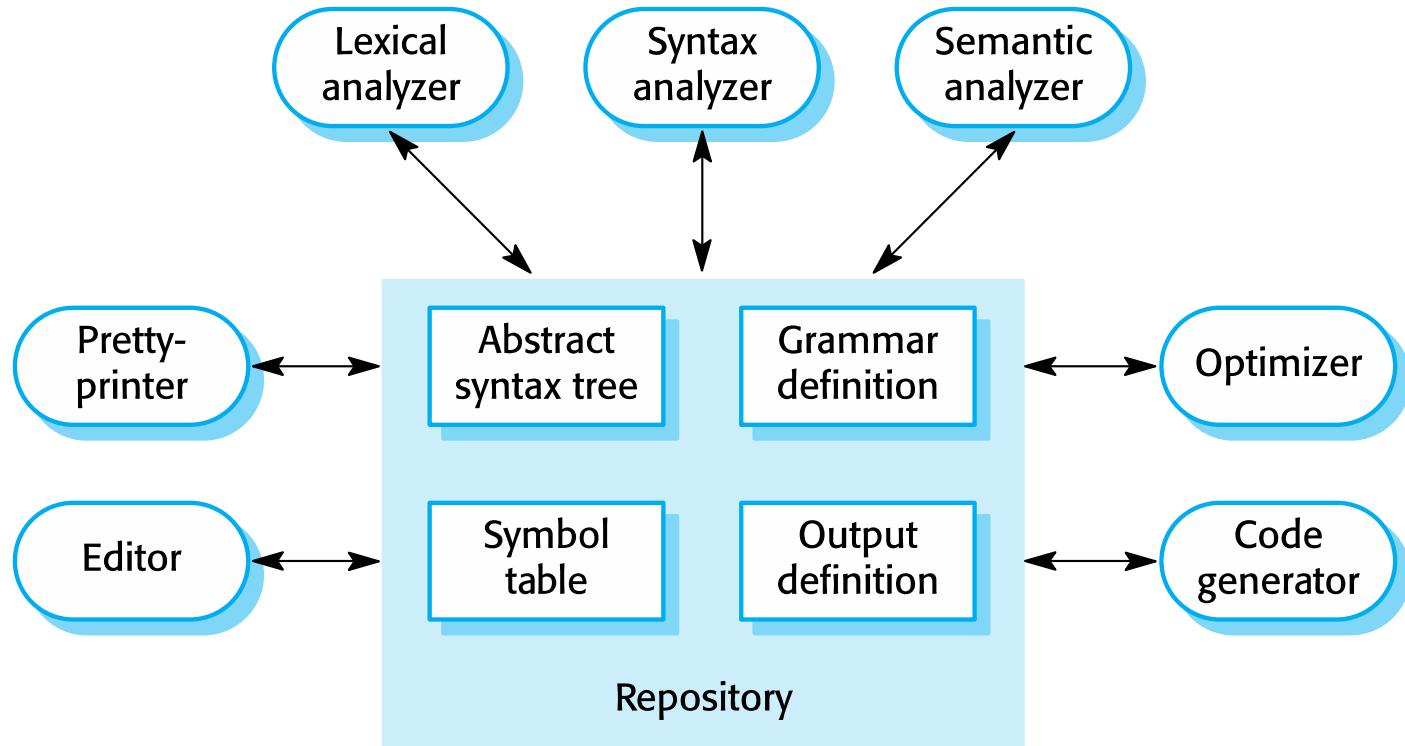
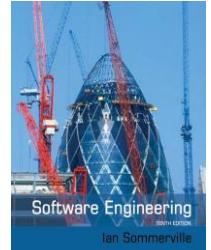
- ✧ A lexical analyzer, which takes input language tokens and converts them to an internal form.
- ✧ A symbol table, which holds information about the names of entities (variables, class names, object names, etc.) used in the text that is being translated.
- ✧ A syntax analyzer, which checks the syntax of the language being translated.
- ✧ A syntax tree, which is an internal structure representing the program being compiled.

Compiler components



- ✧ A semantic analyzer that uses information from the syntax tree and the symbol table to check the semantic correctness of the input language text.
- ✧ A code generator that ‘walks’ the syntax tree and generates abstract machine code.

A repository architecture for a language processing system



A pipe and filter compiler architecture



Key points



- ✧ A software architecture is a description of how a software system is organized.
- ✧ Architectural design decisions include decisions on the type of application, the distribution of the system, the architectural styles to be used.
- ✧ Architectures may be documented from several different perspectives or views such as a conceptual view, a logical view, a process view, and a development view.
- ✧ Architectural patterns are a means of reusing knowledge about generic system architectures. They describe the architecture, explain when it may be used and describe its advantages and disadvantages.



Key points

- ✧ Models of application systems architectures help us understand and compare applications, validate application system designs and assess large-scale components for reuse.
- ✧ Transaction processing systems are interactive systems that allow information in a database to be remotely accessed and modified by a number of users.
- ✧ Language processing systems are used to translate texts from one language into another and to carry out the instructions specified in the input language. They include a translator and an abstract machine that executes the generated language.



Chapter 7 – Design and Implementation



Topics covered

- ✧ Object-oriented design using the UML
- ✧ Design patterns
- ✧ Implementation issues
- ✧ Open source development

Design and implementation



- ✧ Software design and implementation is the stage in the software engineering process at which an executable software system is developed.
- ✧ Software design and implementation activities are invariably inter-leaved.
 - Software design is a creative activity in which you identify software components and their relationships, based on a customer's requirements.
 - Implementation is the process of realizing the design as a program.

Build or buy



- ✧ In a wide range of domains, it is now possible to buy off-the-shelf systems (COTS) that can be adapted and tailored to the users' requirements.
 - For example, if you want to implement a medical records system, you can buy a package that is already used in hospitals. It can be cheaper and faster to use this approach rather than developing a system in a conventional programming language.
- ✧ When you develop an application in this way, the design process becomes concerned with how to use the configuration features of that system to deliver the system requirements.



Object-oriented design using the UML

An object-oriented design process



- ✧ Structured object-oriented design processes involve developing a number of different system models.
- ✧ They require a lot of effort for development and maintenance of these models and, for small systems, this may not be cost-effective.
- ✧ However, for large systems developed by different groups design models are an important communication mechanism.

Process stages



- ✧ There are a variety of different object-oriented design processes that depend on the organization using the process.
- ✧ Common activities in these processes include:
 - Define the context and modes of use of the system;
 - Design the system architecture;
 - Identify the principal system objects;
 - Develop design models;
 - Specify object interfaces.
- ✧ Process illustrated here using a design for a wilderness weather station.

System context and interactions



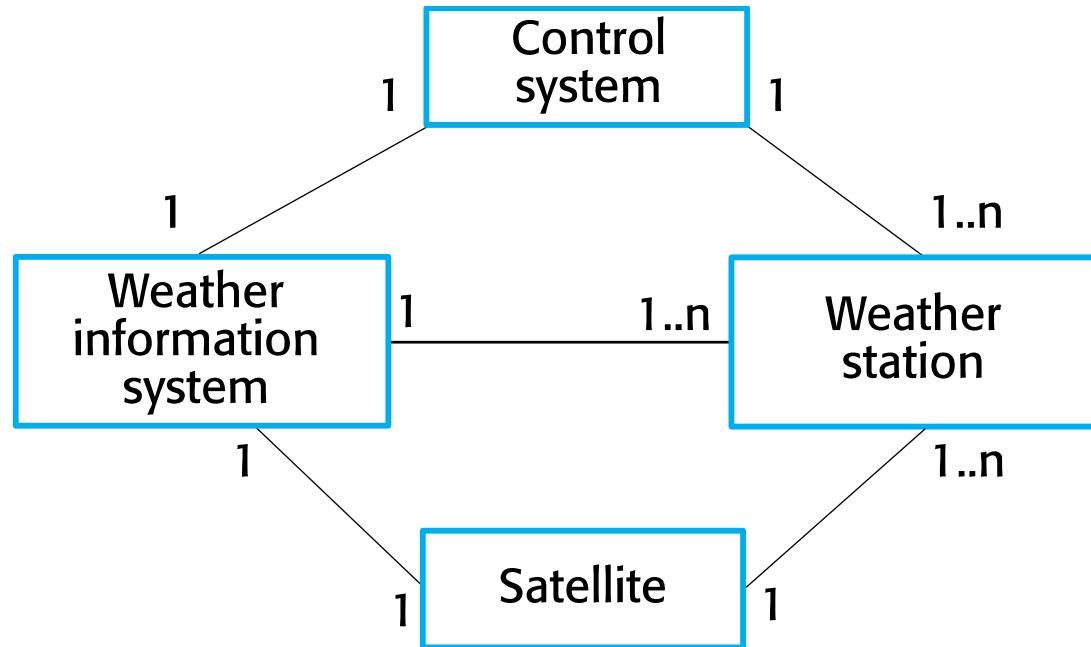
- ✧ Understanding the relationships between the software that is being designed and its external environment is essential for deciding how to provide the required system functionality and how to structure the system to communicate with its environment.
- ✧ Understanding of the context also lets you establish the boundaries of the system. Setting the system boundaries helps you decide what features are implemented in the system being designed and what features are in other associated systems.

Context and interaction models



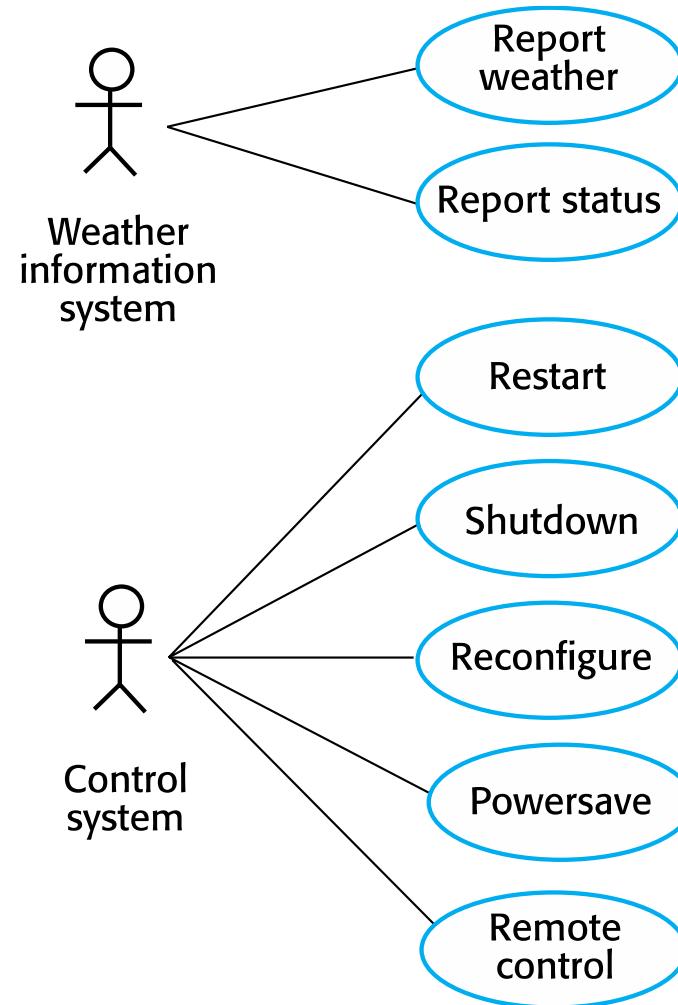
- ✧ A system context model is a structural model that demonstrates the other systems in the environment of the system being developed.
- ✧ An interaction model is a dynamic model that shows how the system interacts with its environment as it is used.

System context for the weather station





Weather station use cases





Use case description—Report weather

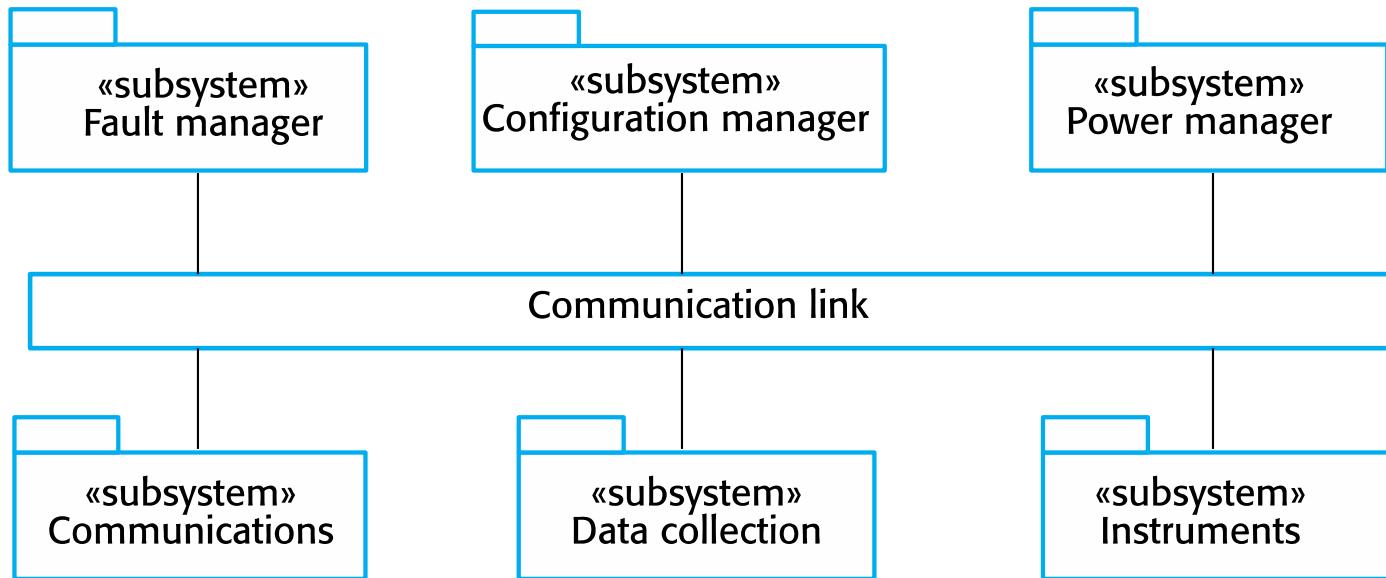
System	Weather station
Use case	Report weather
Actors	Weather information system, Weather station
Description	The weather station sends a summary of the weather data that has been collected from the instruments in the collection period to the weather information system. The data sent are the maximum, minimum, and average ground and air temperatures; the maximum, minimum, and average air pressures; the maximum, minimum, and average wind speeds; the total rainfall; and the wind direction as sampled at five-minute intervals.
Stimulus	The weather information system establishes a satellite communication link with the weather station and requests transmission of the data.
Response	The summarized data is sent to the weather information system.
Comments	Weather stations are usually asked to report once per hour but this frequency may differ from one station to another and may be modified in the future.

Architectural design



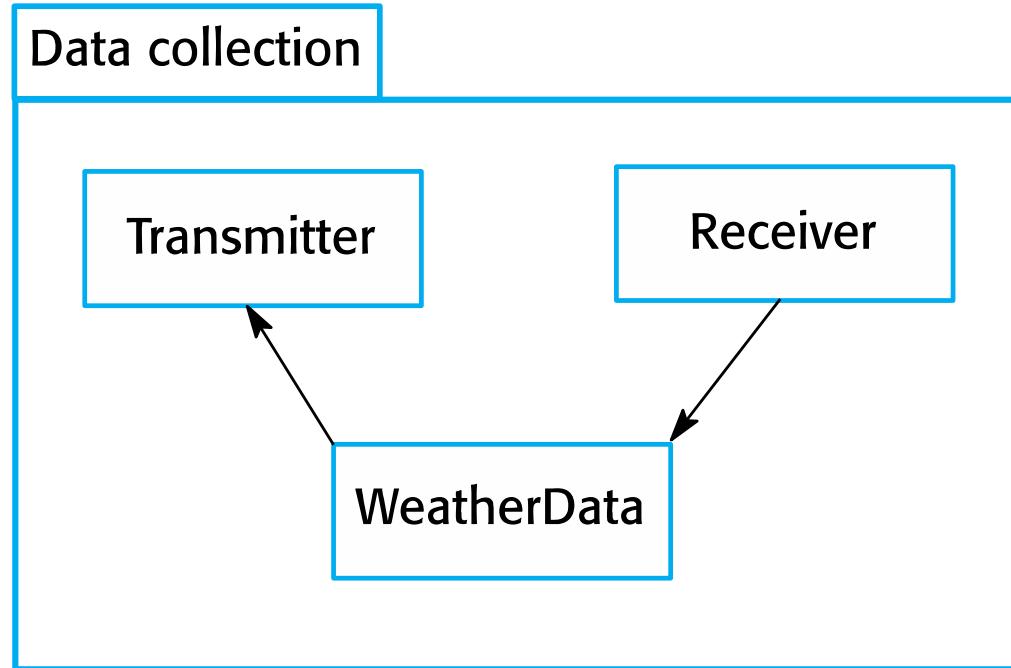
- ✧ Once interactions between the system and its environment have been understood, you use this information for designing the system architecture.
- ✧ You identify the major components that make up the system and their interactions, and then may organize the components using an architectural pattern such as a layered or client-server model.
- ✧ The weather station is composed of independent subsystems that communicate by broadcasting messages on a common infrastructure.

High-level architecture of the weather station





Architecture of data collection system



Object class identification



- ✧ Identifying object classes is often a difficult part of object oriented design.
- ✧ There is no 'magic formula' for object identification. It relies on the skill, experience and domain knowledge of system designers.
- ✧ Object identification is an iterative process. You are unlikely to get it right first time.

Approaches to identification



- ✧ Use a grammatical approach based on a natural language description of the system.
- ✧ Base the identification on tangible things in the application domain.
- ✧ Use a behavioural approach and identify objects based on what participates in what behaviour.
- ✧ Use a scenario-based analysis. The objects, attributes and methods in each scenario are identified.



Weather station object classes

- ✧ Object class identification in the weather station system may be based on the tangible hardware and data in the system:
 - Ground thermometer, Anemometer, Barometer
 - Application domain objects that are ‘hardware’ objects related to the instruments in the system.
 - Weather station
 - The basic interface of the weather station to its environment. It therefore reflects the interactions identified in the use-case model.
 - Weather data
 - Encapsulates the summarized data from the instruments.

Weather station object classes



```
reportWeather ()  
reportStatus ()  
powerSave (instruments)  
remoteControl (commands)  
reconfigure (commands)  
restart (instruments)  
shutdown (instruments)
```

```
groundTemperatures  
windSpeeds  
windDirections  
pressures  
rainfall
```

```
collect ()  
summarize ()
```

Ground thermometer

```
gt_Ident  
temperature
```

Anemometer

```
an_Ident  
windSpeed  
windDirection
```

Barometer

```
bar_Ident  
pressure  
height
```

Design models



- ✧ Design models show the objects and object classes and relationships between these entities.
- ✧ There are two kinds of design model:
 - Structural models describe the static structure of the system in terms of object classes and relationships.
 - Dynamic models describe the dynamic interactions between objects.

Examples of design models



- ✧ Subsystem models that show logical groupings of objects into coherent subsystems.
- ✧ Sequence models that show the sequence of object interactions.
- ✧ State machine models that show how individual objects change their state in response to events.
- ✧ Other models include use-case models, aggregation models, generalisation models, etc.

Subsystem models



- ✧ Shows how the design is organised into logically related groups of objects.
- ✧ In the UML, these are shown using packages - an encapsulation construct. This is a logical model. The actual organisation of objects in the system may be different.

Sequence models

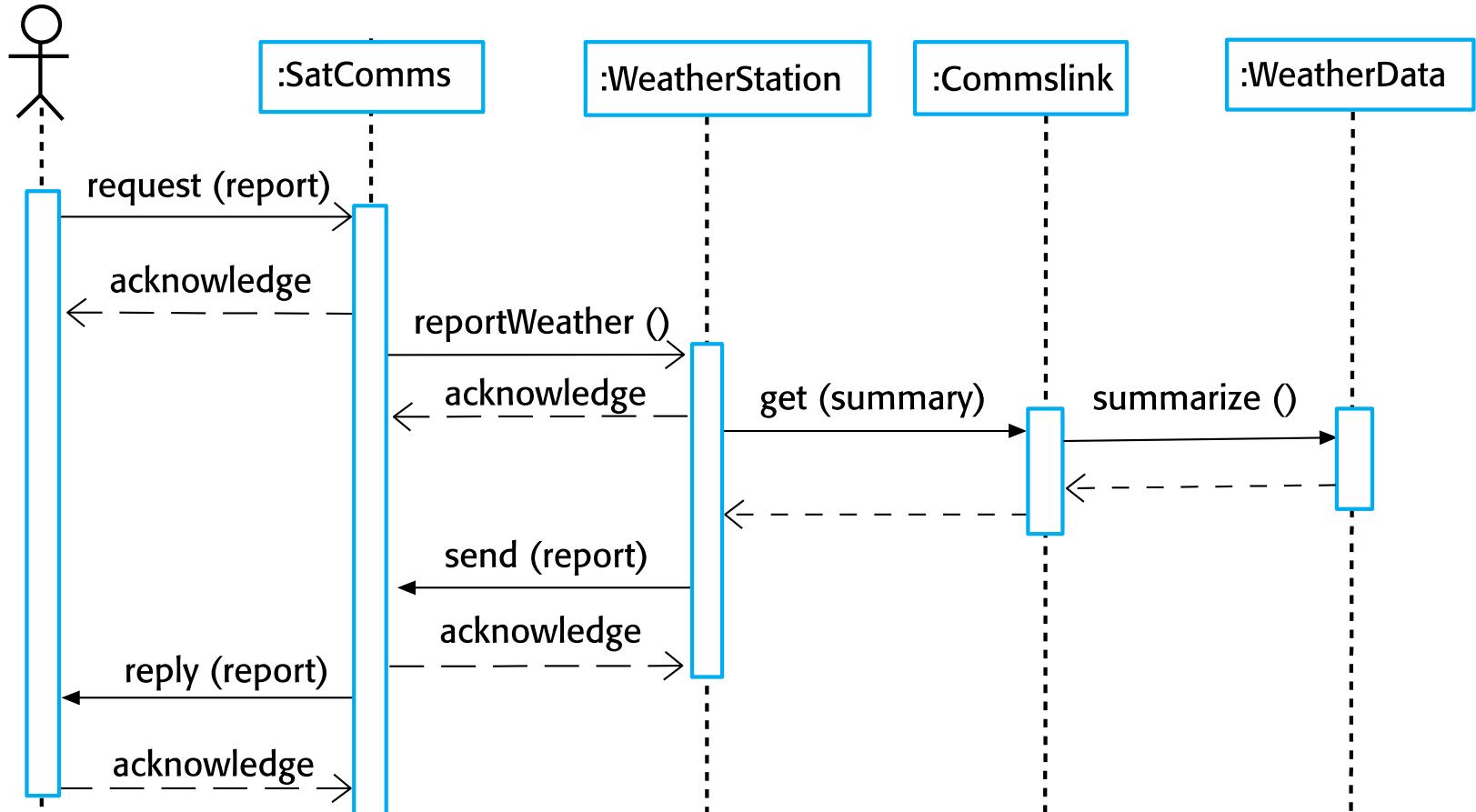


- ✧ Sequence models show the sequence of object interactions that take place
 - Objects are arranged horizontally across the top;
 - Time is represented vertically so models are read top to bottom;
 - Interactions are represented by labelled arrows, Different styles of arrow represent different types of interaction;
 - A thin rectangle in an object lifeline represents the time when the object is the controlling object in the system.

Sequence diagram describing data collection



information system

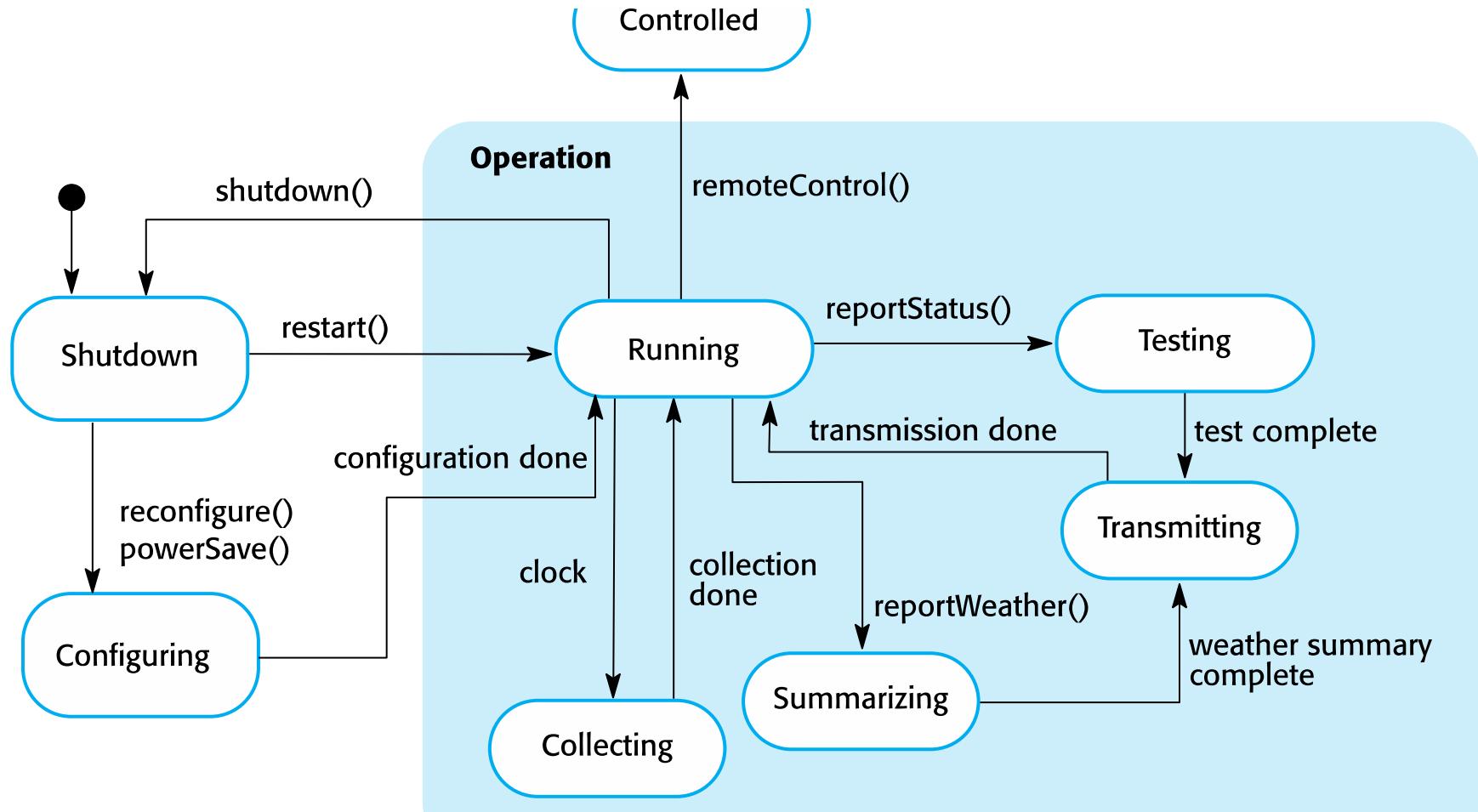


State diagrams



- ✧ State diagrams are used to show how objects respond to different service requests and the state transitions triggered by these requests.
- ✧ State diagrams are useful high-level models of a system or an object's run-time behavior.
- ✧ You don't usually need a state diagram for all of the objects in the system. Many of the objects in a system are relatively simple and a state model adds unnecessary detail to the design.

Weather station state diagram



Interface specification



- ✧ Object interfaces have to be specified so that the objects and other components can be designed in parallel.
- ✧ Designers should avoid designing the interface representation but should hide this in the object itself.
- ✧ Objects may have several interfaces which are viewpoints on the methods provided.
- ✧ The UML uses class diagrams for interface specification but Java may also be used.

Weather station interfaces



«interface» Reporting

weatherReport (WS-Ident): Wreport
statusReport (WS-Ident): Sreport

«interface» Remote Control

startInstrument(instrument): iStatus
stopInstrument (instrument): iStatus
collectData (instrument): iStatus
provideData (instrument): string



Design patterns

Design patterns



- ✧ A design pattern is a way of reusing abstract knowledge about a problem and its solution.
- ✧ A pattern is a description of the problem and the essence of its solution.
- ✧ It should be sufficiently abstract to be reused in different settings.
- ✧ Pattern descriptions usually make use of object-oriented characteristics such as inheritance and polymorphism.

Patterns



- ✧ *Patterns and Pattern Languages are ways to describe best practices, good designs, and capture experience in a way that it is possible for others to reuse this experience.*

Pattern elements



✧ Name

- A meaningful pattern identifier.

✧ Problem description.

✧ Solution description.

- Not a concrete design but a template for a design solution that can be instantiated in different ways.

✧ Consequences

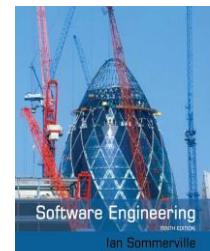
- The results and trade-offs of applying the pattern.



The Observer pattern

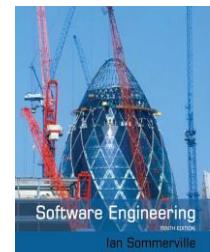
- ✧ Name
 - Observer.
- ✧ Description
 - Separates the display of object state from the object itself.
- ✧ Problem description
 - Used when multiple displays of state are needed.
- ✧ Solution description
 - See slide with UML description.
- ✧ Consequences
 - Optimisations to enhance display performance are impractical.

The Observer pattern (1)



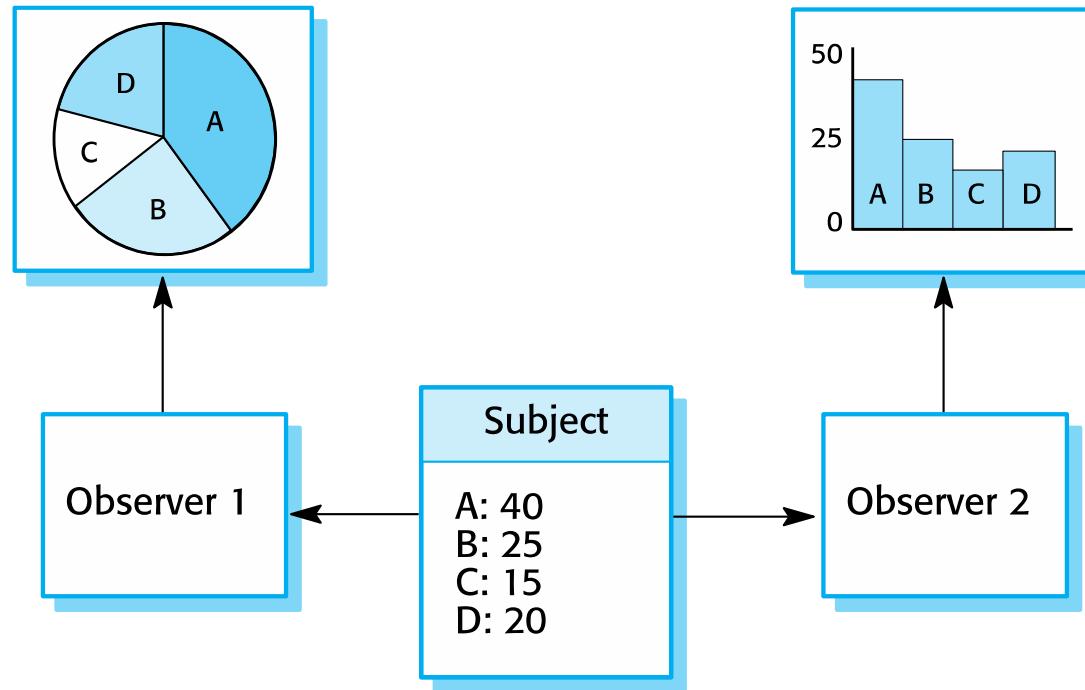
Pattern name	Observer
Description	<p>Separates the display of the state of an object from the object itself and allows alternative displays to be provided. When the object state changes, all displays are automatically notified and updated to reflect the change.</p>
Problem description	<p>In many situations, you have to provide multiple displays of state information, such as a graphical display and a tabular display. Not all of these may be known when the information is specified. All alternative presentations should support interaction and, when the state is changed, all displays must be updated.</p> <p>This pattern may be used in all situations where more than one display format for state information is required and where it is not necessary for the object that maintains the state information to know about the specific display formats used.</p>

The Observer pattern (2)



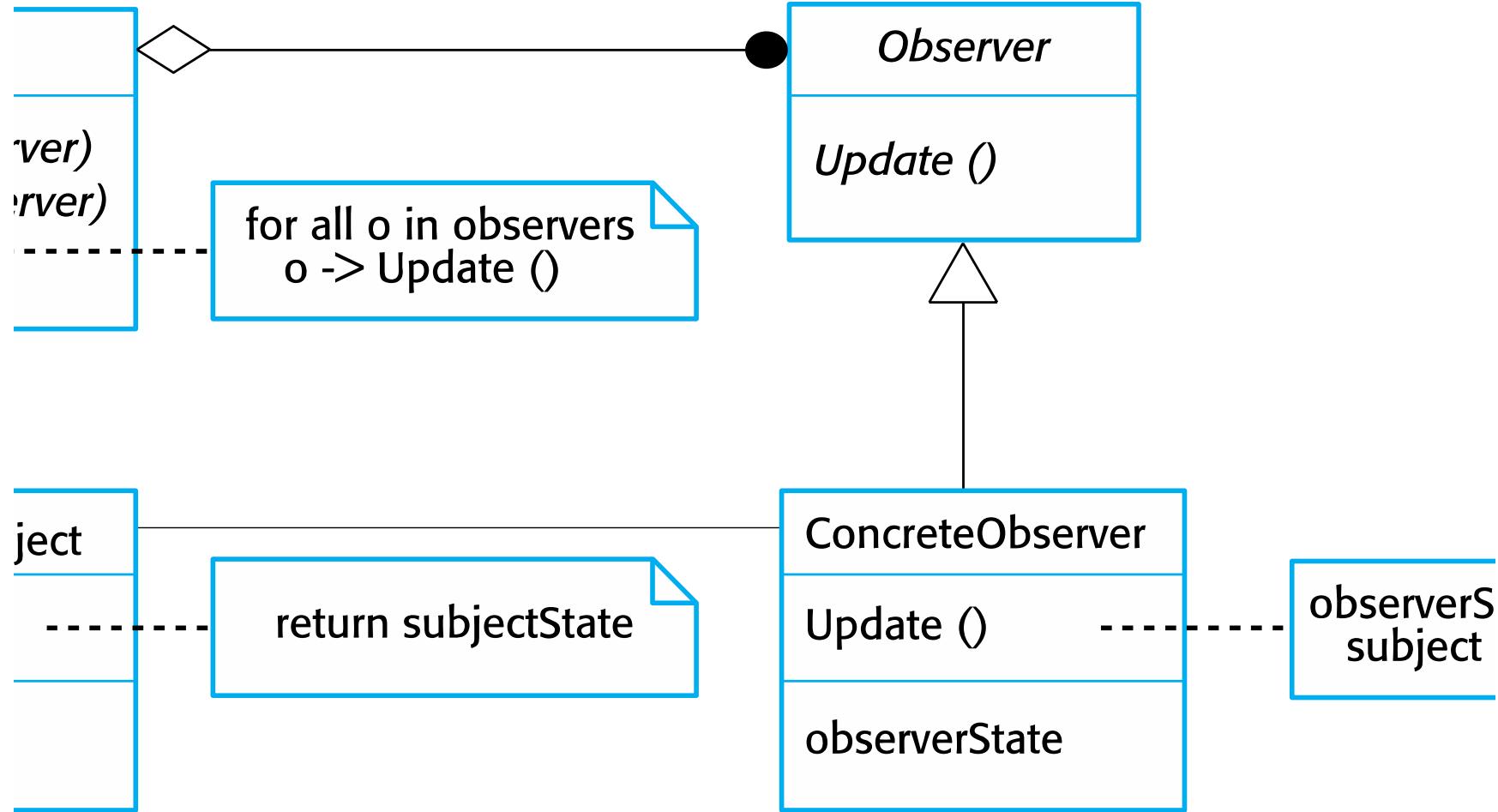
Pattern name	Observer
Solution description	<p>This involves two abstract objects, Subject and Observer, and two concrete objects, ConcreteSubject and ConcreteObject, which inherit the attributes of the related abstract objects. The abstract objects include general operations that are applicable in all situations. The state to be displayed is maintained in ConcreteSubject, which inherits operations from Subject allowing it to add and remove Observers (each observer corresponds to a display) and to issue a notification when the state has changed.</p> <p>The ConcreteObserver maintains a copy of the state of ConcreteSubject and implements the Update() interface of Observer that allows these copies to be kept in step. The ConcreteObserver automatically displays the state and reflects changes whenever the state is updated.</p>
Consequences	<p>The subject only knows the abstract Observer and does not know details of the concrete class. Therefore there is minimal coupling between these objects. Because of this lack of knowledge, optimizations that enhance display performance are impractical. Changes to the subject may cause a set of linked updates to observers to be generated, some of which may not be necessary.</p>

Multiple displays using the Observer pattern





A UML model of the Observer pattern





Design problems

- ✧ To use patterns in your design, you need to recognize that any design problem you are facing may have an associated pattern that can be applied.
 - Tell several objects that the state of some other object has changed (Observer pattern).
 - Tidy up the interfaces to a number of related objects that have often been developed incrementally (Façade pattern).
 - Provide a standard way of accessing the elements in a collection, irrespective of how that collection is implemented (Iterator pattern).
 - Allow for the possibility of extending the functionality of an existing class at run-time (Decorator pattern).



Implementation issues



Implementation issues

- ✧ Focus here is not on programming, although this is obviously important, but on other implementation issues that are often not covered in programming texts:
 - **Reuse** Most modern software is constructed by reusing existing components or systems. When you are developing software, you should make as much use as possible of existing code.
 - **Configuration management** During the development process, you have to keep track of the many different versions of each software component in a configuration management system.
 - **Host-target development** Production software does not usually execute on the same computer as the software development environment. Rather, you develop it on one computer (the host system) and execute it on a separate computer (the target system).

Reuse



- ✧ From the 1960s to the 1990s, most new software was developed from scratch, by writing all code in a high-level programming language.
 - The only significant reuse or software was the reuse of functions and objects in programming language libraries.
- ✧ Costs and schedule pressure mean that this approach became increasingly unviable, especially for commercial and Internet-based systems.
- ✧ An approach to development based around the reuse of existing software emerged and is now generally used for business and scientific software.

Reuse levels



✧ The abstraction level

- At this level, you don't reuse software directly but use knowledge of successful abstractions in the design of your software.

✧ The object level

- At this level, you directly reuse objects from a library rather than writing the code yourself.

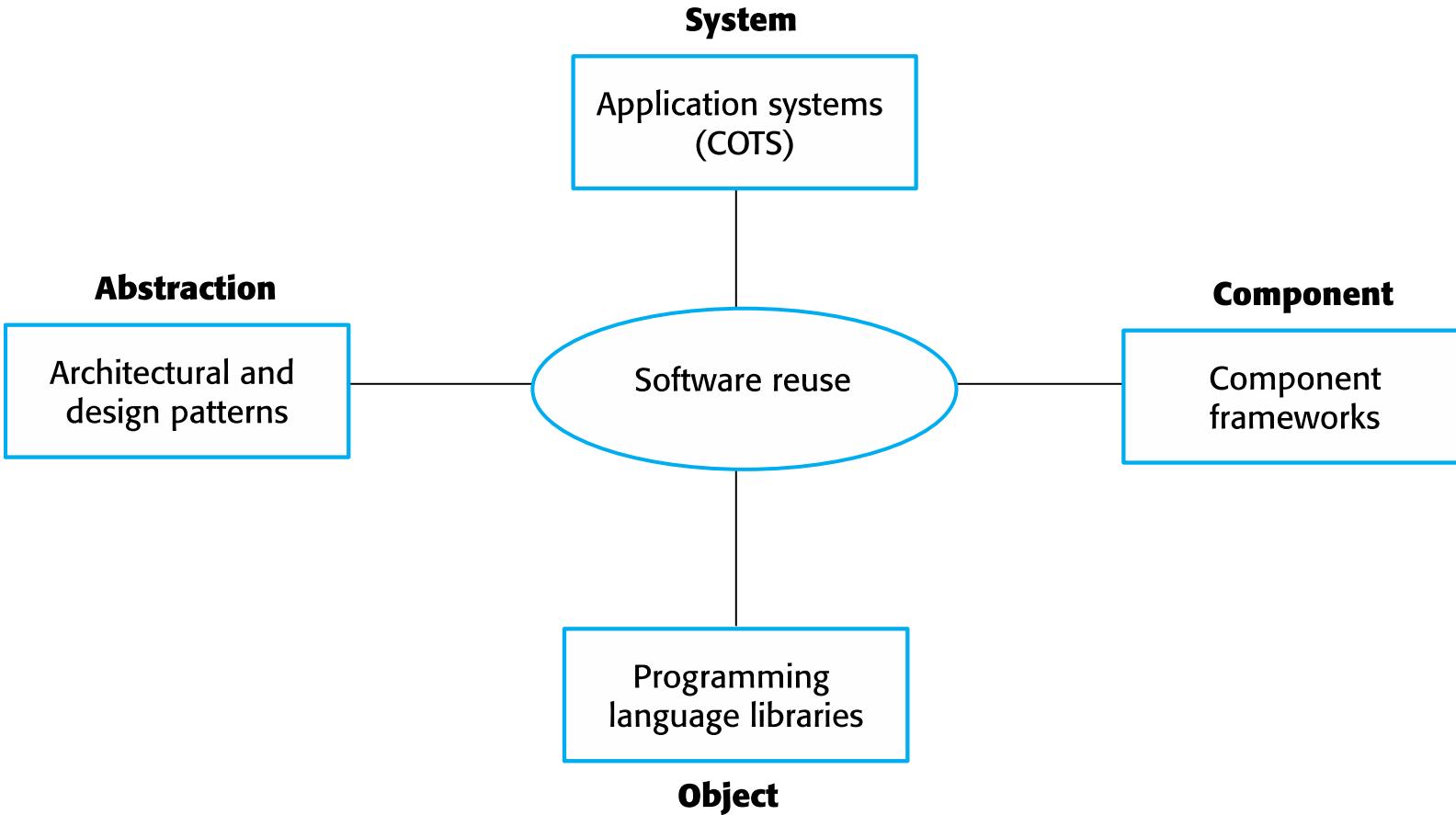
✧ The component level

- Components are collections of objects and object classes that you reuse in application systems.

✧ The system level

- At this level, you reuse entire application systems.

Software reuse

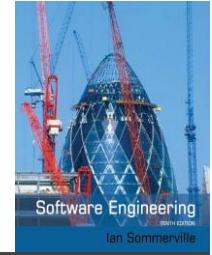


Reuse costs



- ✧ The costs of the time spent in looking for software to reuse and assessing whether or not it meets your needs.
- ✧ Where applicable, the costs of buying the reusable software. For large off-the-shelf systems, these costs can be very high.
- ✧ The costs of adapting and configuring the reusable software components or systems to reflect the requirements of the system that you are developing.
- ✧ The costs of integrating reusable software elements with each other (if you are using software from different sources) and with the new code that you have developed.

Configuration management



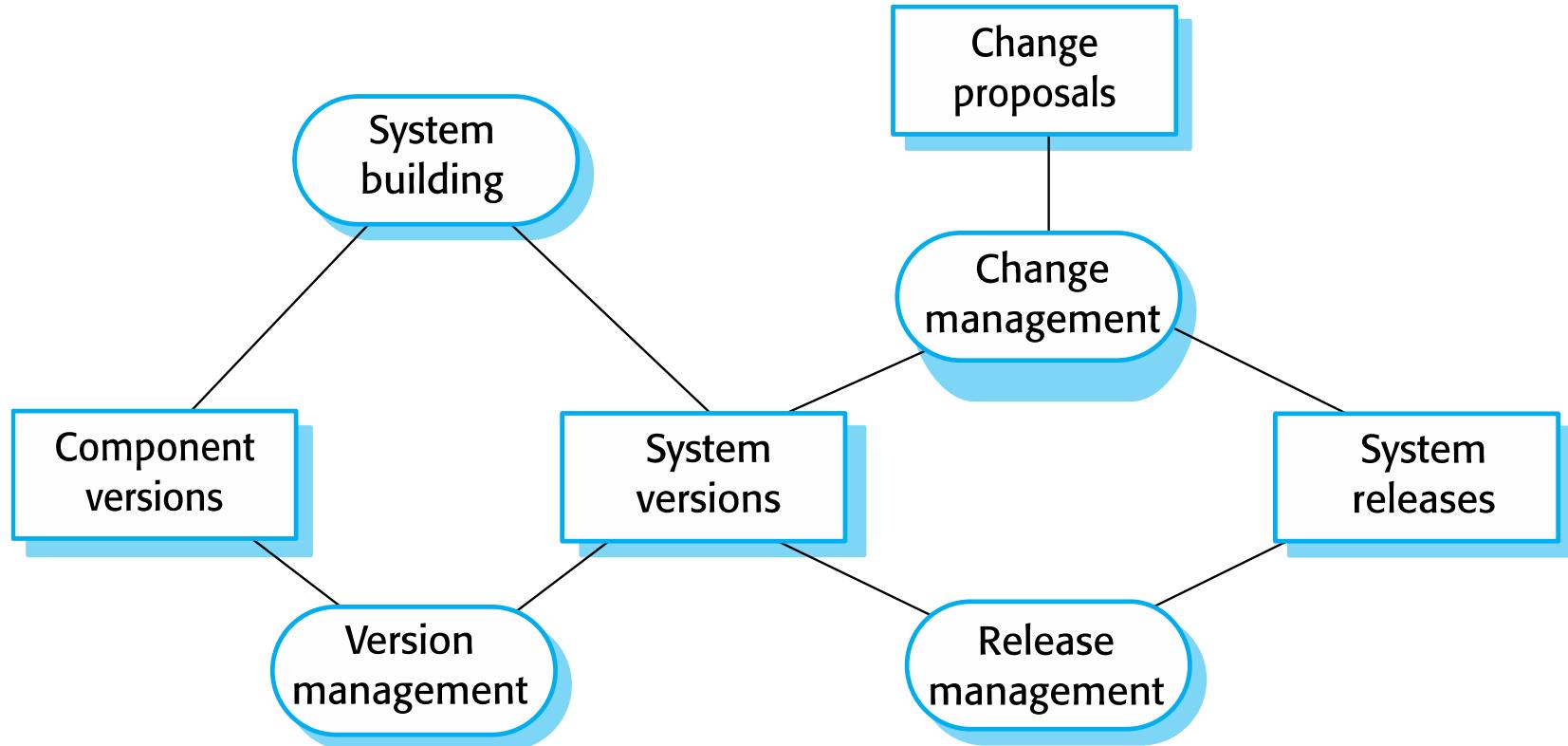
- ✧ Configuration management is the name given to the general process of managing a changing software system.
- ✧ The aim of configuration management is to support the system integration process so that all developers can access the project code and documents in a controlled way, find out what changes have been made, and compile and link components to create a system.
- ✧ See also Chapter 25.

Configuration management activities



- ✧ Version management, where support is provided to keep track of the different versions of software components. Version management systems include facilities to coordinate development by several programmers.
- ✧ System integration, where support is provided to help developers define what versions of components are used to create each version of a system. This description is then used to build a system automatically by compiling and linking the required components.
- ✧ Problem tracking, where support is provided to allow users to report bugs and other problems, and to allow all developers to see who is working on these problems and when they are fixed.

Configuration management tool interaction

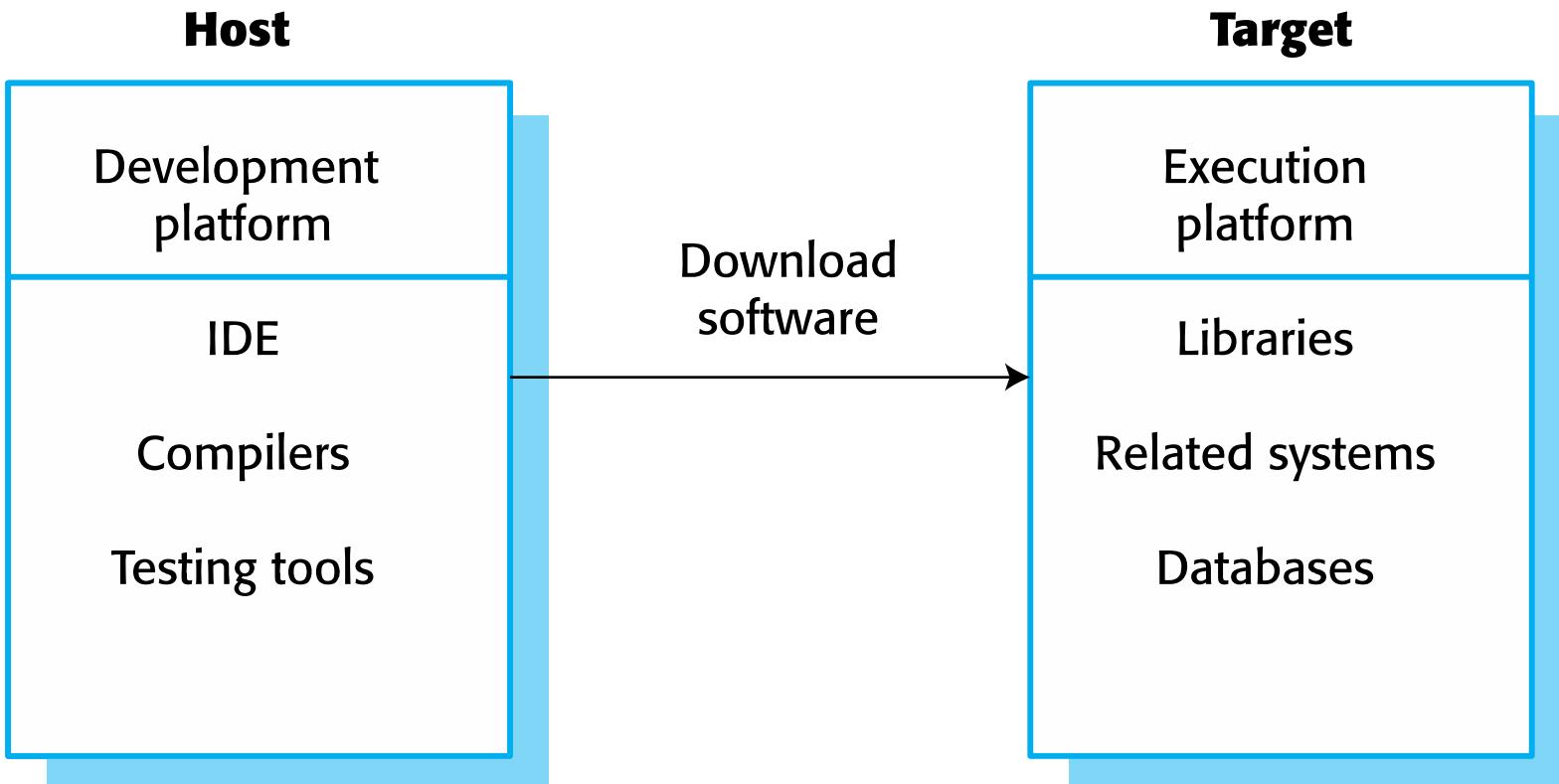


Host-target development



- ✧ Most software is developed on one computer (the host), but runs on a separate machine (the target).
- ✧ More generally, we can talk about a development platform and an execution platform.
 - A platform is more than just hardware.
 - It includes the installed operating system plus other supporting software such as a database management system or, for development platforms, an interactive development environment.
- ✧ Development platform usually has different installed software than execution platform; these platforms may have different architectures.

Host-target development



Development platform tools



- ✧ An integrated compiler and syntax-directed editing system that allows you to create, edit and compile code.
- ✧ A language debugging system.
- ✧ Graphical editing tools, such as tools to edit UML models.
- ✧ Testing tools, such as Junit that can automatically run a set of tests on a new version of a program.
- ✧ Project support tools that help you organize the code for different development projects.

Integrated development environments (IDEs)



- ✧ Software development tools are often grouped to create an integrated development environment (IDE).
- ✧ An IDE is a set of software tools that supports different aspects of software development, within some common framework and user interface.
- ✧ IDEs are created to support development in a specific programming language such as Java. The language IDE may be developed specially, or may be an instantiation of a general-purpose IDE, with specific language-support tools.

Component/system deployment factors



- ✧ If a component is designed for a specific hardware architecture, or relies on some other software system, it must obviously be deployed on a platform that provides the required hardware and software support.
- ✧ High availability systems may require components to be deployed on more than one platform. This means that, in the event of platform failure, an alternative implementation of the component is available.
- ✧ If there is a high level of communications traffic between components, it usually makes sense to deploy them on the same platform or on platforms that are physically close to one other. This reduces the delay between the time a message is sent by one component and received by another.



Open source development

Open source development



- ✧ Open source development is an approach to software development in which the source code of a software system is published and volunteers are invited to participate in the development process
- ✧ Its roots are in the Free Software Foundation (www.fsf.org), which advocates that source code should not be proprietary but rather should always be available for users to examine and modify as they wish.
- ✧ Open source software extended this idea by using the Internet to recruit a much larger population of volunteer developers. Many of them are also users of the code.

Open source systems



- ✧ The best-known open source product is, of course, the Linux operating system which is widely used as a server system and, increasingly, as a desktop environment.
- ✧ Other important open source products are Java, the Apache web server and the mySQL database management system.

Open source issues



- ✧ Should the product that is being developed make use of open source components?
- ✧ Should an open source approach be used for the software's development?

Open source business



- ✧ More and more product companies are using an open source approach to development.
- ✧ Their business model is not reliant on selling a software product but on selling support for that product.
- ✧ They believe that involving the open source community will allow software to be developed more cheaply, more quickly and will create a community of users for the software.



Open source licensing

- ✧ A fundamental principle of open-source development is that source code should be freely available, this does not mean that anyone can do as they wish with that code.
 - Legally, the developer of the code (either a company or an individual) still owns the code. They can place restrictions on how it is used by including legally binding conditions in an open source software license.
 - Some open source developers believe that if an open source component is used to develop a new system, then that system should also be open source.
 - Others are willing to allow their code to be used without this restriction. The developed systems may be proprietary and sold as closed source systems.

License models



- ✧ The GNU General Public License (GPL). This is a so-called ‘reciprocal’ license that means that if you use open source software that is licensed under the GPL license, then you must make that software open source.
- ✧ The GNU Lesser General Public License (LGPL) is a variant of the GPL license where you can write components that link to open source code without having to publish the source of these components.
- ✧ The Berkley Standard Distribution (BSD) License. This is a non-reciprocal license, which means you are not obliged to re-publish any changes or modifications made to open source code. You can include the code in proprietary systems that are sold.

License management



- ✧ Establish a system for maintaining information about open-source components that are downloaded and used.
- ✧ Be aware of the different types of licenses and understand how a component is licensed before it is used.
- ✧ Be aware of evolution pathways for components.
- ✧ Educate people about open source.
- ✧ Have auditing systems in place.
- ✧ Participate in the open source community.



Key points

- ✧ Software design and implementation are inter-leaved activities. The level of detail in the design depends on the type of system and whether you are using a plan-driven or agile approach.
- ✧ The process of object-oriented design includes activities to design the system architecture, identify objects in the system, describe the design using different object models and document the component interfaces.
- ✧ A range of different models may be produced during an object-oriented design process. These include static models (class models, generalization models, association models) and dynamic models (sequence models, state machine models).
- ✧ Component interfaces must be defined precisely so that other objects can use them. A UML interface stereotype may be used to define interfaces.

Key points



- ✧ When developing software, you should always consider the possibility of reusing existing software, either as components, services or complete systems.
- ✧ Configuration management is the process of managing changes to an evolving software system. It is essential when a team of people are cooperating to develop software.
- ✧ Most software development is host-target development. You use an IDE on a host machine to develop the software, which is transferred to a target machine for execution.
- ✧ Open source development involves making the source code of a system publicly available. This means that many people can propose changes and improvements to the software.

Software Engineering at Google

Originally published 6 Feb 2017

Revised 19 Feb 2019.

Fergus Henderson

<fergus@google.com> (work) or

<fergus.henderson@gmail.com> (personal)

Abstract

We catalog and describe Google's key software engineering practices.

Biography

Fergus Henderson has been a software engineer at Google for over 10 years. He started programming as a kid in 1979, and went on to academic research in programming language design and implementation. With his PhD supervisor, he co-founded a research group at the University of Melbourne that developed the programming language Mercury. He has been a program committee member for eight international conferences, and has released over 500,000 lines of open-source code. He was a former moderator of the Usenet newsgroup comp.std.c++ and was an officially accredited “Technical Expert” to the ISO C and C++ committees. He has over 15 years of commercial software industry experience. At Google, he was one of the original developers of Blaze, a build tool now used across Google, and worked on the server-side software behind speech recognition and voice actions (before Siri!) and speech synthesis. He currently manages Google's text-to-speech engineering team, but still writes and reviews plenty of code. Software that he has written is installed on over a billion devices, and gets used over a billion times per day.

Contents

[Abstract](#)

[Biography](#)

[Contents](#)

[1. Introduction](#)

[2. Software development](#)

[2.1. The Source Repository](#)

[2.2. The Build System](#)

[2.3. Code Review](#)

[2.4. Testing](#)

[2.5. Bug tracking](#)

[2.6. Programming languages](#)

[2.7. Debugging and Profiling tools](#)

[2.8. Release engineering](#)

[2.9. Launch approval](#)

[2.10. Post-mortems](#)

[2.11. Frequent rewrites](#)

[3. Project management](#)

[3.1. 20% time](#)

[3.2. Objectives and Key Results \(OKRs\)](#)

[3.3. Project approval](#)

[3.4. Corporate reorganizations](#)

[3.5. Annual hack weeks](#)

[4. People management](#)

[4.1. Roles](#)

[4.2. Facilities](#)

[4.3. Training](#)

[4.4. Transfers](#)

[4.5. Performance appraisal and rewards](#)

[5. Conclusions](#)

[Acknowledgements](#)

[References](#)

1. Introduction

Google has been a phenomenally successful company. As well as the success of Google Search and AdWords, Google has delivered many other stand-out products, including Google Maps, Google News, Google Translate, Google speech recognition, Chrome, and Android. Google has also greatly enhanced and scaled many products that were acquired by purchasing smaller companies, such as YouTube, and has made significant contributions to a wide variety of open-source projects. And Google has demonstrated some amazing products that are yet to launch, such as self-driving cars.

There are many reasons for Google's success, including enlightened leadership, great people, a high hiring bar, and the financial strength that comes from successfully taking advantage of an early lead in a very rapidly growing market. But one of these reasons is that **Google has developed excellent software engineering practices**, which have helped it to succeed. These practices have evolved over time based on the accumulated and distilled wisdom of many of the most talented software engineers on the planet. We would like to share knowledge of our practices with the world, and to share some of the lessons that we have learned from our mistakes along the way.

The aim of this paper is to catalogue and briefly describe Google's key software engineering practices. Other organizations and individuals can then compare and contrast these with their own software engineering practices, and consider whether to apply some of these practices themselves.

Many authors (e.g. [9], [10], [11]) have written books or articles analyzing Google's success and history. But most of those have dealt mainly with business, management, and culture; only a fraction of those (e.g. [1, 2, 3, 4, 5, 6, 7, 13, 14, 16, 21]) have explored the software engineering side of things, and most explore only a single aspect; and none of them provide a brief written overview of software engineering practices at Google as a whole, as this paper aims to do.

2. Software development

2.1. The Source Repository

Most of Google's code is stored in a single unified source-code repository, and is accessible to all software engineers at Google. (Customer data is tightly secured, it's only source code that's accessible.) There are some notable exceptions to the use of this single widely accessible repository, particularly the two large open-source projects Chrome and Android, which use separate open-source repositories, and some high-value or security-critical pieces of code for which read access is locked down more tightly. But most Google projects

share the same repository. As of January 2015, this 86 terabyte repository contained a billion files, including over 9 million source code files containing a total of **2 billion lines of source code**, with a history of 35 million commits and a change rate of 40 thousand commits per work day [18]. Write access to the repository is controlled: only the listed owners of each subtree of the repository can approve changes to that subtree. But generally any engineer can access any piece of code, can check it out and build it, can make local modifications, can test them, and can send changes for review by the code owners, and if an owner approves, can check in (commit) those changes. Culturally, engineers are encouraged to fix anything that they see is broken and know how to fix, regardless of project boundaries. This empowers engineers and leads to higher-quality infrastructure that better meets the needs of those using it.

Almost all development occurs at the “head” of the repository, not on branches. This helps identify integration problems early and minimizes the amount of merging work needed. It also makes it much easier and faster to push out security fixes.

Automated systems run tests frequently, often after every change to any file in the transitive dependencies of the test, although this is not always feasible. These systems automatically notify the author and reviewers of any change for which the tests failed, typically within a few minutes. Most teams make the current status of their build very conspicuous by installing prominent displays or even sculptures with color-coded lights (green for building successfully and all tests passing, red for some tests failing, black for broken build). This helps to focus engineers’ attention on keeping the build green. Most larger teams also have a “build cop” who is responsible for ensuring that the tests continue to pass at head, by working with the authors of the offending changes to quickly fix any problems or to roll back the offending change. (The build cop role is typically rotated among the team or among its more experienced members.) This focus on keeping the build green makes development at head practical, even for very large teams.

Code ownership. Each subtree of the repository can have a file listing the user ids of the “owners” of that subtree. Subdirectories also inherit owners from their parent directories, although that can be optionally suppressed. The owners of each subtree control write access to that subtree, as described in the code review section below. Each subtree is required to have at least two owners, although typically there are more, especially in geographically distributed teams. It is common for the whole team to be listed in the owners file. Changes to a subtree can be made by anyone at Google, not just the owners, but must be approved by an owner. This ensures that every change is reviewed by an engineer who understands the software being modified.

For more on the source code repository at Google, see [17, 18, 21]; and for how another large company deals with the same challenge, see [19].

2.2. The Build System

Google uses a distributed build system known as Blaze, which is responsible for compiling and linking software and for running tests. It provides standard commands for building and testing software that work across the whole repository. These standard commands and the highly optimized implementation mean that **it is typically very simple and quick for any Google engineer to build and test any software in the repository**. This consistency is a key enabler which helps to make it practical for engineers to make changes across project boundaries.

Programmers write “BUILD” files that Blaze uses to determine how to build their software. Build entities such as libraries, programs, and tests are declared using fairly high-level **declarative build specifications** that specify, for each entity, its name, its source files, and the libraries or other build entities that it depends on. These build specifications are comprised of declarations called “build rules” that each specify high-level concepts like “here is a C++ library with these source files which depends on these other libraries”, and it is up to the build system to map each build rule to a set of build steps, e.g. steps for compiling each source file and steps for linking, and for determining which compiler and compilation flags to use.

In some cases, notably Go programs, build files can be generated (and updated) automatically, since the dependency information in the BUILD files is (often) an abstraction of the dependency information in the source files. But they are nevertheless checked in to the repository. This ensures that the build system can quickly determine dependencies by analyzing only the build files rather than the source files, and it avoids excessive coupling between the build system and compilers or analysis tools for the many different programming languages supported.

The build system’s implementation uses Google’s distributed computing infrastructure. The work of each build is typically **distributed across hundreds or even thousands of machines**. This makes it possible to build extremely large programs quickly or to run thousands of tests in parallel.

Individual build steps must be “hermetic”: they depend only on their declared inputs. Enforcing that all dependencies be correctly declared is a consequence of distributing the build: only the declared inputs are sent to the machine on which the build step is run. As a result the build system can be relied on to know the true dependencies. Even the compilers that the build system invokes are treated as inputs.

Individual build steps are deterministic. As a consequence, the build system can cache build results. Software engineers can sync their workspace back to an old change number and can rebuild and will get exactly the same binary. Furthermore, this cache can be safely shared between different users. (To make this work properly, we had to eliminate non-determinism in the tools invoked by the build, for example by scrubbing out timestamps in the generated output files.)

The build system is reliable. The build system tracks dependencies on changes to the build rules themselves, and knows to rebuild targets if the action to produce them changed, even if the inputs to that action didn't, for example when only the compiler options changed. It also deals properly with interrupting the build part way, or modifying source files during the build: in such cases, you need only rerun the build command. There is never any need to run the equivalent of "make clean".

Build results are cached "in the cloud". This includes intermediate results. If another build request needs the same results, the build system will automatically reuse them rather than rebuilding, even if the request comes from a different user.

Incremental rebuilds are fast. The build system stays resident in memory so that for rebuilds it can incrementally analyze just the files that have changed since the last build.

Presubmit checks. Google has tools for automatically running a suite of tests when initiating a code review and/or preparing to commit a change to the repository. Each subtree of the repository can contain a configuration file which determines which tests to run, and whether to run them at code review time, or immediately before submitting, or both. The tests can be either synchronous, i.e. run before sending the change for review and/or before committing the change to the repository (good for fast-running tests); or asynchronous, with the results emailed to the review discussion thread. [The review thread is the email thread on which the code review takes place; all the information in that thread is also displayed in the web-based code review tool.]

2.3. Code Review

Google has built excellent web-based code review tools, integrated with email, that allow authors to request a review, and allows reviewers to view side-by-side diffs (with nice color coding) and comment on them. When the author of a change initiates a code review, the reviewers are notified by e-mail, with a link to the web review tool's page for that change. Email notifications are sent when reviewers submit their review comments. In addition, automated tools can send notifications, containing for example the results of automated tests or the findings of static analysis tools.

All changes to the main source code repository MUST be reviewed by at least one other engineer. In addition, if the author of a change is not one of the owners of the files being modified, then at least one of the owners must review and approve the change.

In exceptional cases, an owner of a subtree can check in (commit) an urgent change to that subtree *before* it is reviewed, but a reviewer must still be named, and the change author and reviewer will get automatically nagged about it until the change has been reviewed and approved. In such cases, any modifications needed to address review comments must be done

in a separate change, since the original change will have already been committed.

Google has tools for automatically suggesting reviewer(s) for a given change, by looking at the ownership and authorship of the code being modified, the history of recent reviewers, and the number of pending code reviews for each potential reviewer. At least one of the owners of each subtree which a change affects must review and approve that change. But apart from that, the author is free to choose reviewer(s) as they see fit.

One potential issue with code review is that if the reviewers are too slow to respond or are overly reluctant to approve changes, this could potentially slow down development. The fact that the code author chooses their reviewers helps avoid such problems, allowing engineers to avoid reviewers that might be overly possessive about their code, or to send reviews for simple changes to less thorough reviewers and to send reviews for more complex changes to more experienced reviewers or to several reviewers.

Code review discussions for each project are automatically copied to a mailing list designated by the project maintainers. Anyone is free to comment on any change, regardless of whether they were named as a reviewer of that change, both before and after the change is committed. If a bug is discovered, it's common to track down the change that introduced it and to comment on the original code review thread to point out the mistake so that the original author and reviewers are aware of it.

It is also possible to send code reviews to several reviewers and then to commit the change as soon as one of them has approved (provided either the author or the first responding reviewer is an owner, of course), before the other reviewers have commented, with any subsequent review comments being dealt with in follow-up changes. This can reduce the turnaround time for reviews.

In addition to the main section of the repository, **there is an “experimental” section of the repository where the normal code review requirements are not enforced.** However, code running in production must be in the main section of the repository, and engineers are very strongly encouraged to develop code in the main section of the repository, rather than developing in experimental and then moving it to the main section, since code review is most effective when done as the code is developed rather than afterwards. In practice engineers often request code reviews even for code in experimental.

Engineers are encouraged to keep each individual change small, with larger changes preferably broken into a series of smaller changes that a reviewer can easily review in one go. This also makes it easier for the author to respond to major changes suggested during the review of each piece; very large changes are often too rigid and resist reviewer-suggested

changes. One way in which keeping changes small is encouraged¹ is that the code review tools label each code review with a description of the size of the change, with changes of 30-99 lines added/deleted/removed being labelled “medium-size” and with changes of above 300 lines being labelled with increasingly disparaging labels, e.g. “large” (300-999), “freakin huge” (1000-1999), etc. (However, in a typically Googly way, this is kept fun by replacing these familiar descriptions with amusing alternatives on a few days each year, such as talk-like-a-pirate day. :)

2.4. Testing

Unit Testing is strongly encouraged and widely practiced at Google. All code used in production is expected to have unit tests, and the code review tool will highlight if source files are added without corresponding tests. Code reviewers usually require that any change which adds new functionality should also add new tests to cover the new functionality. Mocking frameworks (which allow construction of lightweight unit tests even for code with dependencies on heavyweight libraries) are quite popular.

Integration testing and regression testing are also widely practiced.

As discussed in [“Presubmit Checks”](#) above, testing can be automatically enforced as part of the code review and commit process.

Google also has automated tools for measuring test coverage. The results are also integrated as an optional layer in the source code browser.

Load testing prior to deployment is also de rigueur at Google. Teams are expected to produce a table or graph showing how key metrics, particularly latency and error rate, vary with the rate of incoming requests.

2.5. Bug tracking

Google uses a bug tracking system called Buganizer for tracking issues: bugs, feature requests, customer issues, and processes (such as releases or clean-up efforts). Bugs are categorized into hierarchical components and each component can have a default assignee and default email list to CC. When sending a source change for review, engineers are prompted to associate the change with a particular issue number.

It is common (though not universal) for teams at Google to regularly scan through open issues in their component(s), prioritizing them and where appropriate assigning them to particular engineers. Some teams have a particular individual responsible for bug triage, others do bug triage in their regular team meetings. Many teams at Google make use of labels on bugs to

¹ This has changed somewhat in recent years. More recent versions of the code review tools no longer use the more disparaging labels for large CLs, but they are still labelled with their size, e.g. “S”, “M”, “L”, “XL”.

indicate whether bugs have been triaged, and which release(s) each bug is targeted to be fixed in.

2.6. Programming languages

Software engineers at Google are strongly encouraged to program in one of five officially-approved programming languages at Google: **C++, Java, Python, Go, or JavaScript**. Minimizing the number of different programming languages used reduces obstacles to code reuse and programmer collaboration.

There are also Google **style guides** for each language, to ensure that code all across the company is written with similar style, layout, naming conventions, etc. In addition there is a company-wide **readability** training process, whereby experienced engineers who care about code readability train other engineers in how to write readable, idiomatic code in a particular language, by reviewing a substantial change or series of changes until the reviewer is satisfied that the author knows how to write readable code in that language. Each change that adds non-trivial new code in a particular language must be approved by someone who has passed this “readability” training process in that language.

In addition to these five languages, many **specialized domain-specific languages** are used for particular purposes (e.g. the build language used for specifying build targets and their dependencies).

Interoperation between these different programming languages is done mainly using **Protocol Buffers**. Protocol Buffers is a way of encoding structured data in an efficient yet extensible way. It includes a domain-specific language for specifying structured data, together with a compiler that takes in such descriptions and generates code in C++, Java, Python, for constructing, accessing, serializing, and deserializing these objects. Google’s version of Protocol Buffers is integrated with Google’s RPC libraries, enabling simple cross-language RPCs, with serialization and deserialization of requests and responses handled automatically by the RPC framework.

Commonality of process is a key to making development easy even with an enormous code base and a diversity of languages: there is a single set of commands to perform all the usual software engineering tasks (such as check out, edit, build, test, review, commit, file bug report, etc.) and the same commands can be used no matter what project or language. Developers don’t need to learn a new development process just because the code that they are editing happens to be part of a different project or written in a different language.

2.7. Debugging and Profiling tools

Google servers are linked with libraries that provide a number of tools for debugging running servers. In case of a server crash, a signal handler will automatically dump a stack trace to a

log file, as well as saving the core file. If the crash was due to running out of heap memory, the server will dump stack traces of the allocation sites of a sampled subset of the live heap objects. There are also web interfaces for debugging that allow examining incoming and outgoing RPCs (including timing, error rates, rate limiting, etc.), changing command-line flag values (e.g. to increase logging verbosity for a particular module), resource consumption, profiling, and more. These tools greatly increase the overall ease of debugging to the point where it is rare to fire up a traditional debugger such as gdb.

2.8. Release engineering

A few teams have dedicated release engineers, but for most teams at Google, the release engineering work is done by regular software engineers.

Releases are done frequently for most software; weekly or fortnightly releases are a common goal, and some teams even release daily. This is made possible by **automating most of the normal release engineering tasks**. Releasing frequently helps to keep engineers motivated (it's harder to get excited about something if it won't be released until many months or even years into the future) and increases overall velocity by allowing more iterations, and thus more opportunities for feedback and more chances to respond to feedback, in a given time.

A release typically starts in a fresh workspace, by syncing to the change number of the latest "green" build (i.e. the last change for which all the automatic tests passed), and making a release branch. The release engineer can select additional changes to be "cherry-picked", i.e. merged from the main branch onto the release branch. Then the software will be rebuilt from scratch and the tests are run. If any tests fail, additional changes are made to fix the failures and those additional changes are cherry-picked onto the release branch, after which the software will be rebuilt and the tests rerun. When the tests all pass, the built executable(s) and data file(s) are packaged up. All of these steps are automated so that the release engineer need only run some simple commands, or even just select some entries on a menu-driven UI, and choose which changes (if any) to cherry pick.

Once a candidate build has been packaged up, it is typically loaded onto a "**staging**" server for further **integration testing by small set of users** (sometimes just the development team).

A useful technique involves sending a copy of (a subset of) the requests from production traffic to the staging server, but also sending those same requests to the current production servers for actual processing. The responses from the staging server are discarded, and the responses from the live production servers are sent back to the users. This helps ensure that any issues that might cause serious problems (e.g. server crashes) can be detected before putting the server into production.

The next step is to usually roll out to one or more "**canary**" servers that are **processing a subset of the live production traffic**. Unlike the "staging" servers, these are processing and

responding to real users.

Finally the release can be rolled out to all servers in all data centers. For very high-traffic, high-reliability services, this is done with a **gradual roll-out** over a period of a couple of days, to help reduce the impact of any outages due to newly introduced bugs not caught by any of the previous steps.

For more information on release engineering at Google, see chapter 8 of the SRE book [7]. See also [15].

2.9. Launch approval

The launch of any user-visible change or significant design change requires approvals from a number of people outside of the core engineering team that implements the change. In particular approvals (often subject to detailed review) are required to ensure that code complies with legal requirements, privacy requirements, security requirements, reliability requirements (e.g. having appropriate automatic monitoring to detect server outages and automatically notify the appropriate engineers), business requirements, and so forth.

The launch process is also designed to ensure that appropriate people within the company are notified whenever any significant new product or feature launches.

Google has an internal launch approval tool that is used to track the required reviews and approvals and ensure compliance with the defined launch processes for each product. This tool is easily customizable, so that different products or product areas can have different sets of required reviews and approvals.

For more information about launch processes, see chapter 27 of the SRE book [7].

2.10. Post-mortems

Whenever there is a significant outage of any of our production systems, or similar mishap, the people involved are required to write a post-mortem document. This document describes the incident, including title, summary, impact, timeline, root cause(s), what worked/what didn't, and action items. **The focus is on the problems, and how to avoid them in future, not on the people or apportioning blame.** The impact section tries to quantify the effect of the incident, in terms of duration of outage, number of lost queries (or failed RPCs, etc.), and revenue. The timeline section gives a timeline of the events leading up to the outage and the steps taken to diagnose and rectify it. The what worked/what didn't section describes the lessons learnt -- which practices helped to quickly detect and resolve the issue, what went wrong, and what concrete actions (preferably filed as bugs assigned to specific people) can be take to reduce the likelihood and/or severity of similar problems in future.

For more information on post-mortem culture at Google, see chapter 15 of the SRE book [7].

2.11. Frequent rewrites

Most software at Google gets rewritten every few years.

This may seem incredibly costly. Indeed, it does consume a large fraction of Google's resources. However, it also has some crucial benefits that are key to Google's agility and long-term success. In a period of a few years, it is typical for the requirements for a product to change significantly, as the software environment and other technology around it change, and as changes in technology or in the marketplace affect user needs, desires, and expectations. Software that is a few years old was designed around an older set of requirements and is typically not designed in a way that is optimal for current requirements. Furthermore, it has typically accumulated a lot of complexity. Rewriting code cuts away all the unnecessary accumulated complexity that was addressing requirements which are no longer so important. In addition, rewriting code is a way of transferring knowledge and a sense of ownership to newer team members. This sense of ownership is crucial for productivity: engineers naturally put more effort into developing features and fixing problems in code that they feel is "theirs". Frequent rewrites also encourage mobility of engineers between different projects which helps to encourage cross-pollination of ideas. Frequent rewrites also help to ensure that code is written using modern technology and methodology.

3. Project management

3.1. 20% time

Engineers are permitted to spend up to 20% of their time working on any project of their choice, without needing approval from their manager or anyone else. This trust in engineers is extremely valuable, for several reasons. Firstly, it allows anyone with a good idea, even if it is an idea that others would not immediately recognize as being worthwhile, to have sufficient time to develop a prototype, demo, or presentation to show the value of their idea. Secondly, it provides management with visibility into activity that might otherwise be hidden. In other companies that don't have an official policy of allowing 20% time, engineers sometimes work on "skunkwork" projects without informing management. It's much better if engineers can be open about such projects, describing their work on such projects in their regular status updates, even in cases where their management may not agree on the value of the project. Having a company-wide official policy and a culture that supports it makes this possible. Thirdly, by allowing engineers to spend a small portion of their time working on more fun stuff, it keeps engineers motivated and excited by what they do, and stops them getting burnt out, which can easily happen if they feel compelled to spend 100% of their time working on more tedious tasks. The difference in productivity between engaged, motivated engineers and burnt out engineers is

a *lot* more than 20%. Fourthly, it encourages a culture of innovation. Seeing other engineers working on fun experimental 20% projects encourages everyone to do the same.

3.2. Objectives and Key Results (OKRs)

Individuals and teams at Google are required to explicitly document their goals and to assess their progress towards these goals. Teams set quarterly and annual objectives, with measurable key results that show progress towards these objectives. This is done at every level of the company, going all the way up to defining goals for the whole company. Goals for individuals and small teams should align with the higher-level goals for the broader teams that they are part of and with the overall company goals. At the end of each quarter, progress towards the measurable key results is recorded and each objective is given a score from 0.0 (no progress) to 1.0 (100% completion). OKRs and OKR scores are normally made visible across Google (with occasional exceptions for especially sensitive information such as highly confidential projects), but they *not* used directly as input to an individual's performance appraisal.

OKRs should be set high: the desired target overall average score is 65%, meaning that a team is encouraged to set as goals about 50% more tasks than they are likely to actually accomplish. If a team scores significantly higher than that, they are encouraged to set more ambitious OKRs for the following quarter (and conversely if they score significantly lower than that, they are encouraged to set their OKRs more conservatively the next quarter).

OKRs provide a key mechanism for communicating what each part of the company is working on, and for encouraging good performance from employees via social incentives... engineers know that their team will have a meeting where the OKRs will be scored, and have a natural drive to try to score well, even though OKRs have no direct impact on performance appraisals or compensation. Defining key results that are objective and measurable helps ensure that this human drive to perform well is channelled to doing things that have real concrete measurable impact on progress towards shared objectives.

3.3. Project approval

Although there is a well-defined process for launch approvals, Google does not have a well-defined process for project approval or cancellation. Despite having been at Google for over 10 years, and now having become a manager myself, I still don't fully understand how such decisions are made. In part this is because the approach to this is not uniform across the company. Managers at every level are responsible and accountable for what projects their teams work on, and exercise their discretion as they see fit. In some cases, this means that such decisions are made in a quite bottom-up fashion, with engineers being given freedom to choose which projects to work on, within their team's scope. In other cases, such decisions are made in a much more top-down fashion, with executives or managers making decisions about which projects will go ahead, which will get additional resources, and which will get cancelled.

3.4. Corporate reorganizations

Occasionally an executive decision is made to cancel a large project, and then the many engineers who had been working on that project may have to find new projects on new teams. Similarly there have been occasional “defragmentation” efforts, where projects that are split across multiple geographic locations are consolidated into a smaller number of locations, with engineers in some locations being required to change team and/or project in order to achieve this. In such cases, engineers are generally given freedom to choose their new team and role from within the positions available in their geographic location, or in the case of defragmentation, they may also be given the option of staying on the same team and project by moving to a different location.

In addition, other kinds of corporate reorganizations, such as merging or splitting teams and changes in reporting chains, seem to be fairly frequent occurrences, although I don’t know how Google compares with other large companies on that. In a large, technology-driven organization, somewhat frequent reorganization may be necessary to avoid organizational inefficiencies as the technology and requirements change. They can help to avoid or mitigate the common problem in large organizations of “shipping the org chart”, where the software architecture reflects the organization’s reporting structure.

3.5. Annual hack weeks

Another way in which Google encourages innovation is by holding “hackathons” [23]. In many Google offices, this takes the form of an annual week-long “hackathon” in which software engineers can take time out of their regular schedule to work on new innovative projects. After a kick-off meeting in which people can pitch ideas, they form small teams that spend a week building a demo or prototype based on a new idea, and at the end of the week present their demos to an audience and a panel of judges, who award (small) prizes for the best projects. The biggest rewards, though, are the recognition — and the chance to have a successful hackathon project graduate to become a full-time real project.

Despite all engineers being given permission from their site lead to participate in such hackathons, many engineers do find it difficult to detach from their day-to-day responsibilities, and so participation rates are generally fairly low (e.g. 5-20%), although many more will attend the initial kick-off and/or the final presentations and may get inspired by the ideas presented.

4. People management

4.1. Roles

As we’ll explain in more detail below, Google separates the engineering and management

career progression ladders, separates the tech lead role from management, embeds research within engineering, and supports engineers with product managers, project managers, and site reliability engineers (SREs). It seems likely that at least some of these practices are important to sustaining the culture of innovation that has developed at Google.

Google has a small number of different roles within engineering. Within each role, there is a career progression possible, with a sequence of levels, and the possibility of promotion (with associated improvement to compensation, e.g. salary) to recognize performance at the next level.

The main roles are these:

- **Engineering Manager**

This is the only people management role in this list. Individuals in other roles such as Software Engineer *may* manage people, but Engineering Managers *always* manage people. Engineering Managers are often former Software Engineers, and invariably have considerable technical expertise, as well as people skills.

There is a distinction between technical leadership and people management.

Engineering Managers do not necessarily lead projects; projects are led by a Tech Lead, who can be an Engineering Manager, but who is more often a Software Engineer. A project's Tech Lead has the final say for technical decisions in that project.

Managers are responsible for selecting Tech Leads, and for the performance of their teams. They perform coaching and assisting with career development, do performance evaluation (using input from peer feedback, see below), and are responsible for some aspects of compensation. They are also responsible for some parts of the hiring process.

Engineering Managers normally directly manage anywhere between 3 and 30 people, although 8 to 12 is most common.

- **Software Engineer (SWE)**

Most people doing software development work have this role. The hiring bar for software engineers at Google is very high; by hiring only exceptionally good software engineers, a lot of the software problems that plague other organizations are avoided or minimized.

Like many modern software companies, **Google has separate career progression sequences for engineering and management**. Although it is possible for a Software Engineer to manage people, or to transfer to the Engineering Manager role, managing

people is *not* a requirement for promotion, even at the highest levels. At the higher levels, showing leadership is required, but that can come in many forms. For example creating great software that has a huge impact or is used by very many other engineers is sufficient. This is important, because it means that people who have great technical skills but lack the desire or skills to manage people still have a good career progression path that does not require them to take a management track. This avoids the problem that some organizations suffer where people end up in management positions for reasons of career advancement but neglect the people management of the people in their team.

- **Research Scientist**

The hiring criteria for this role are very strict, and the bar is extremely high, requiring demonstrated exceptional research ability evidenced by a great publication record *and* ability to write code. Many very talented people in academia who would be able to qualify for a Software Engineer role would not qualify for a Research Scientist role at Google; most of the people with PhDs at Google are Software Engineers rather than Research Scientists. Research scientists are evaluated on their research contributions, including their publications, but apart from that and the different title, there is not really that much difference between the Software Engineer and Research Scientist role at Google. Both can do original research and publish papers, both can develop new product ideas and new technologies, and both can and do write code and develop products. Research Scientists at Google usually work alongside Software Engineers, in the same teams and working on the same products or the same research. This practice of embedding research within engineering contributes greatly to the ease with which new research can be incorporated into shipping products.

- **Site Reliability Engineer (SRE)**

The maintenance of operational systems is done by software engineering teams, rather than traditional sysadmin types, but the hiring requirements for SREs are slightly different than the requirements for the Software Engineer position (software engineering skills requirements can be slightly lower, if compensated for by expertise in other skills such as networking or unix system internals). The nature and purpose of the SRE role is explained very well and in detail in the SRE book [7], so we won't discuss it further here.

- **Product Manager**

Product Managers are responsible for the management of a product; as advocates for the product users, they coordinate the work of software engineers, evangelizing features of importance to those users, coordinating with other teams, tracking bugs and schedules, and ensuring that everything needed is in place to produce a high quality

product. Product Managers usually do NOT write code themselves, but work with software engineers to ensure that the right code gets written.

- **Program Manager / Technical Program Manager**

Program Managers have a role that is broadly similar to Product Manager, but rather than managing a product, they manage projects, processes, or operations (e.g. data collection). Technical Program Managers are similar, but also require specific technical expertise relating to their work, e.g. linguistics for dealing with speech data.

The ratio of Software Engineers to Product Managers and Program Managers varies across the organization, but is generally high, e.g. in the range 4:1 to 30:1.

4.2. Facilities

Google is famous for its fun facilities, with features like slides, ball pits, and games rooms. That helps attract and retain good talent. Google's excellent cafes, which are free to employees, provide that function too, and also subtly encourage Googlers to stay in the office; hunger is never a reason to leave. The frequent placement of "microkitchens" where employees can grab snacks and drinks serves the same function too, but also acts as an important source of informal idea exchange, as many conversations start up there. Gyms, sports, and on-site massage help keep employees fit, healthy, and happy, which improves productivity and retention.

The seating at Google is open-plan, and often fairly dense. While controversial [20], this encourages communication, sometimes at the expense of individual concentration, and is economical.

Employees are assigned an individual seat, but seats are re-assigned fairly frequently (e.g. every 6-12 months, often as a consequence of the organization expanding), with seating chosen by managers to facilitate and encourage communication, which is always easier between adjacent or nearly adjacent individuals.

Google's facilities all have meeting rooms fitted with state-of-the-art video conference facilities, where connecting to the other party for a prescheduled calendar invite is just a single tap on the screen.

4.3. Training

Google encourages employee education in many ways:

- New Googlers ("Nooglers") have a mandatory initial training course.
- Technical staff (SWEs and research scientists) start by doing "Codelabs": short online training courses in individual technologies, with coding exercises.

- Google offers employees a variety of online and in-person training courses.
- Google also offers support for studying at external institutions.

In addition, each Noogler is usually appointed an official “Mentor” and a separate “Buddy” to help get them up to speed. Unofficial mentoring also occurs via regular meetings with their manager, team meetings, code reviews, design reviews, and informal processes.

4.4. Transfers

Transfers between different parts of the company are encouraged, to help spread knowledge and technology across the organization and improve cross-organization communication. Transfers between projects and/or offices are allowed for employees in good standing after 12 months in a position. Software engineers are also encouraged to do temporary assignments in other parts of the organization, e.g. a six-month “rotation” (temporary assignment) in SRE (Site Reliability Engineering).

4.5. Performance appraisal and rewards

Feedback is strongly encouraged at Google. Engineers can give each other explicit positive feedback via “peer bonuses” and “kudos”. Any employee can nominate any other employee for a “peer bonus” -- a cash bonus of \$100 -- up to twice per year, for going beyond the normal call of duty, just by filling in a web form to describe the reason. Team-mates are also typically notified when a peer bonus is awarded. Employees can also give “kudos”, formalized statements of praise which provide explicit social recognition for good work, but with no financial reward; for “kudos” there is no requirement that the work be beyond the normal call of duty, and no limit on the number of times that they can be bestowed.

Managers can also award bonuses, including spot bonuses, e.g. for project completion. And as with many companies, Google employees get annual performance bonuses and equity awards based on their performance.

Google has a very careful and detailed promotion process, which involves nomination by self or manager, self-review, peer reviews, manager appraisals; the actual decisions are then made by promotion committees based on that input, and the results can be subject to further review by promotion appeals committees. Ensuring that the right people get promoted is critical to maintaining the right incentives for employees.

Poor performance, on the other hand, is handled with manager feedback, and if necessary with performance improvement plans, which involve setting very explicit concrete performance targets and assessing progress towards those targets. If that fails, termination for poor performance is possible, but in practice this is *extremely* rare at Google.

Manager performance is assessed with feedback surveys; every employee is asked to fill in an survey about the performance of their manager twice a year, and the results are anonymized and aggregated and then made available to managers. This kind of upward feedback is very important for maintaining and improving the quality of management throughout the organization.

5. Conclusions

We have briefly described most of the key software engineering practices used at Google. Of course Google is now a large and diverse organization, and some parts of the organization have different practices. But the practices described here are generally followed by most teams at Google.

With so many different software engineering practices involved, and with so many other reasons for Google's success that are not related to our software engineering practices, it is extremely difficult to give any quantitative or objective evidence connecting individual practices with improved outcomes. However, these practices are the ones that have stood the test of time at Google, where they have been subject to the collective subjective judgement of many thousands of excellent software engineers.

Acknowledgements

Special thanks to Alan Donovan for his extremely detailed and constructive feedback, and thanks also to Yaroslav Volovich, Urs Hözle, Brian Strope, Alexander Gutkin, Alex Gruenstein, Hameed Husaini, Glen Shires, Niall Murphy, Ranjit Mathew, Emma Soederberg, and Rob Siemborski for their very helpful comments on earlier drafts of this paper.

References

- [1] *Build in the Cloud: Accessing Source Code*, Nathan York,
<http://google-engtools.blogspot.com/2011/06/build-in-cloud-accessing-source-code.html>
- [2] *Build in the Cloud: How the Build System works*, Christian Kemper,
<http://google-engtools.blogspot.com/2011/08/build-in-cloud-how-build-system-works.htm>
- [3] *Build in the Cloud: Distributing Build Steps*, Nathan York
<http://google-engtools.blogspot.com/2011/09/build-in-cloud-distributing-build-steps.html>
- [4] *Build in the Cloud: Distributing Build Outputs*, Milos Besta, Yevgeniy Miretskiy and Jeff Cox
<http://google-engtools.blogspot.com/2011/10/build-in-cloud-distributing-build.html>
- [5] *Testing at the speed and scale of Google*, Pooja Gupta, Mark Ivey, and John Penix, Google engineering tools blog, June 2011.
<http://google-engtools.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html>

- [6] *Building Software at Google Scale Tech Talk*, Michael Barnathan, Greg Estren, Pepper Lebeck-Jone, Google tech talk.
<http://www.youtube.com/watch?v=2qv3fcXW1mg>
- [7] *Site Reliability Engineering*, Betsy Beyer, Chris Jones, Jennifer Petoff, Niall Richard Murphy, O'Reilly Media, April 2016, ISBN 978-1-4919-2909-4.
<https://landing.google.com/sre/book.html>
- [8] *How Google Works*, Eric Schmidt, Jonathan Rosenberg.
<http://www.howgoogleworks.net>
- [9] *What would Google Do?: Reverse-Engineering the Fastest Growing Company in the History of the World*, Jeff Jarvis, Harper Business, 2011.
https://books.google.co.uk/books/about/What_Would_Google_Do.html?id=GvkEcAAACAAJ&redir_esc=y
- [10] *The Search: How Google and Its Rivals Rewrote the Rules of Business and Transformed Our Culture*, John Battelle, 8 September 2005.
https://books.google.co.uk/books/about/The_Search.html?id=4MY8PgAACAAJ&redir_esc=y
- [11] *The Google Story*, David A. Vise, Pan Books, 2008.
<http://www.thegooglestory.com/>
- [12] *Searching for Build Debt: Experiences Managing Technical Debt at Google*, J. David Morgenthaler, Misha Gridnev, Raluca Sauciuc, and Sanjay Bhansali.
<http://static.googleusercontent.com/media/research.google.com/en//pubs/archive/37755.pdf>
- [13] *Development at the speed and scale of Google*, A. Kumar, December 2010, presentation, QCon.
<http://www.infoq.com/presentations/Development-at-Google>
- [14] *How Google Tests Software*, J. A. Whittaker, J. Arbon, and J. Carollo, Addison-Wesley, 2012.
- [15] *Release Engineering Practices and Pitfalls*, H. K. Wright and D. E. Perry, in *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*, IEEE, 2012, pp. 1281–1284.
<http://www.hyrumwright.org/papers/icse2012.pdf>
- [16] *Large-Scale Automated Refactoring Using ClangMR*, H. K. Wright, D. Jasper, M. Klimek, C. Carruth, Z. Wan, in *Proceedings of the 29th International Conference on Software Maintenance (ICSM '13)*, IEEE, 2013, pp. 548–551.
- [17] *Why Google Stores Billions of Lines of Code in a Single Repository*, Rachel Potvin, presentation.
<https://www.youtube.com/watch?v=W71BTkUbdqE>
- [18] *The Motivation for a Monolithic Codebase*, Rachel Potvin, Josh Levenberg, Communications of the ACM, July 2016.
<http://cacm.acm.org/magazines/2016/7/204032-why-google-stores-billions-of-lines-of-code-in-a-single-repository/fulltext>
- [19] *Scaling Mercurial at Facebook*, Durham Goode, Siddharth P. Agarwa, Facebook blog post, January 7th, 2014.
<https://code.facebook.com/posts/218678814984400/scaling-mercurial-at-facebook/>

[20] *Why We (Still) Believe In Private Offices*, David Fullerton, Stack Overflow blog post, January 16th, 2015.

<https://blog.stackoverflow.com/2015/01/why-we-still-believe-in-private-offices/>

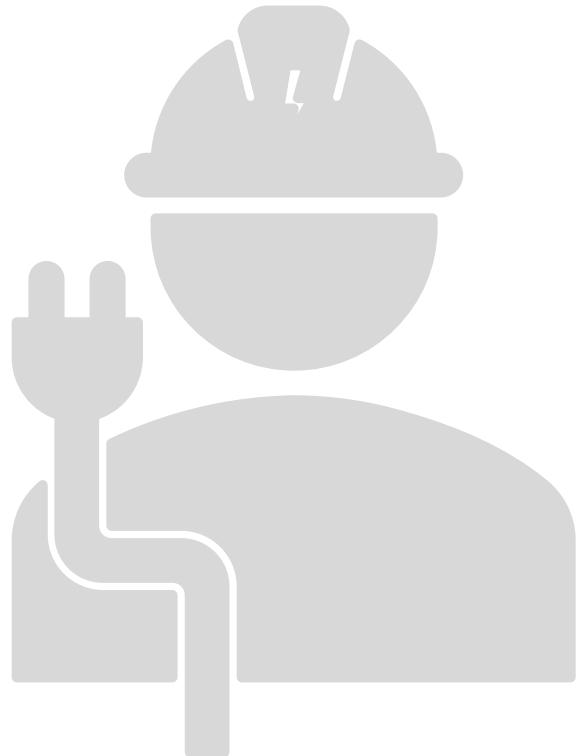
[21] *Continuous Integration at Google Scale*, John Micco, presentation, EclipseCon, 2013.

<http://eclipsecon.org/2013/sites/eclipsecon.org.2013/files/2013-03-24%20Continuous%20Integration%20at%20Google%20Scale.pdf>

[22] Bazel web site. <https://bazel.build/>

[23] *Unlock your team's creativity: running great hackathons*, Max Saltonstall, Google blog post, Aug 28, 2018.

<https://www.blog.google/inside-google/working-google/unlock-your-teams-creativity-running-great-hackathons/>



Software Construction – Case Study of Google

Dr. Gayashan Amarasinghe

Dept. of Computer Science and Engineering

Your experiences – how do you begin a software construction project?

What are the steps that you follow?

Case Study – Software development process at Google

- Based on “Software Engineering at Google” by Fergus Henderson, Originally published 6 Feb 2017 Revised 19 Feb 2019.[1]
- Overview
 - Source Repository
 - Build System
 - Code review
 - Testing
 - Bug tracking
 - Programming languages
 - Debugging and Profiling
 - Release Engineering
 - Launch
 - ...
- Project management
- People management

[1] <https://arxiv.org/ftp/arxiv/papers/1702/1702.01715.pdf>

Source Repository

- Most of Google's code is stored in a single unified source-code repository and is accessible to all software engineers at Google.
 - 86TB, 2 billion LOC (2015)
 - 35 million commits
 - 40000 commits per workday
- Write access to the repository is controlled:
 - only the listed owners of each subtree of the repository can approve changes to that subtree.
- Generally, any engineer can access any piece of code, can check it out and build it, can make local modifications, can test them, and can send changes for review by the code owners, and if an owner approves, can check in (commit) those changes.
- Culturally, engineers are encouraged to fix anything that they see is broken and know how to fix, regardless of project boundaries.

Source Repository

- Automated systems run tests frequently, often after every change to any file in the transitive dependencies of the test, although this is not always feasible.
- These systems automatically notify the author and reviewers of any change for which the tests failed, typically within a few minutes.
- Most teams make the current status of their build very conspicuous.

Source Repository – code ownership

- Each subtree of the repository can have a file listing the user ids of the “owners” of that subtree.
 - Subdirectories also inherit owners from their parent directories, although that can be optionally suppressed.
- The owners of each subtree control write access to that subtree.
- Each subtree is required to have at least two owners, although typically there are more, especially in geographically distributed teams.
 - It is common for the whole team to be listed in the owners-file.
- Changes to a subtree can be made by anyone at Google, not just the owners, but must be approved by an owner.
 - This ensures that every change is reviewed by an engineer who understands the software being modified.

The Build System

- Google uses a distributed build system known as Blaze, which is responsible for compiling and linking software and for running tests.
 - <https://bazel.build/> - open source
- Provides standard commands for building and testing software that work across the whole repository.
 - These standard commands and the highly optimized implementation mean that it is typically very simple and quick for any Google engineer to build and test any software in the repository.
- Programmers write “BUILD” files that Blaze uses to determine how to build their software.
- The build system’s implementation uses Google’s distributed computing infrastructure.

The Build System

- Individual build steps must be “hermetic”: they depend only on their declared inputs.
- Individual build steps are deterministic.
- The build system is reliable.
 - The build system tracks dependencies on changes to the build rules themselves
- Build results are cached “in the cloud”
- Incremental rebuilds are fast.
- Presubmit checks/tests
 - Google has tools for automatically running a suite of tests when initiating a code review and/or preparing to commit a change to the repository.
 - The tests can be either synchronous, i.e. run before sending the change for review and/or before committing the change to the repository (good for fast-running tests);
 - or asynchronous, with the results emailed to the review discussion thread.

Code Review

- Google has built excellent web-based code review tools, integrated with email, that allow authors to request a review, and allows reviewers to view side-by-side diffs (with nice color coding) and comment on them.
 - When the author of a change initiates a code review, the reviewers are notified by e-mail, with a link to the web review tool's page for that change.
- All changes to the main source code repository MUST be reviewed by at least one other engineer.
 - if the author of a change is not one of the owners of the files being modified, then at least one of the owners must review and approve the change.
- This process can be slow? How to avoid that for rapid development?

Code Review

- Code review discussions for each project are automatically copied to a mailing list designated by the project maintainers.
 - Anyone is free to comment on any change, regardless of whether they were named as a reviewer of that change, both before and after the change is committed.
 - If a bug is discovered, it's common to track down the change that introduced it and to comment on the original code review thread to point out the mistake so that the original author and reviewers are aware of it.
-
- Engineers are encouraged to keep each individual change small.
 - larger changes preferably broken into a series of smaller changes that a reviewer can easily review in one go.
 - This also makes it easier for the author to respond to major changes suggested during the review of each piece; very large changes are often too rigid and resist reviewer-suggested changes.

Testing

- Unit Testing is strongly encouraged and widely practiced at Google.
- All code used in production is expected to have unit tests, and the code review tool will highlight if source files are added without corresponding tests.
- Code reviewers usually require that any change which adds new functionality should also add new tests to cover the new functionality.
- Mocking frameworks (which allow construction of lightweight unit tests even for code with dependencies on heavyweight libraries) are quite popular.
- Integration testing and regression testing are also widely practiced.
- Load testing prior to deployment is also strictly followed at Google.
 - Teams are expected to produce a table or graph showing how key metrics, particularly latency and error rate, vary with the rate of incoming requests.

Bug Tracking

- Google uses a bug tracking system called Buganizer for tracking issues:
 - bugs, feature requests, customer issues, and processes (such as releases or clean-up efforts).
- Bugs are categorized into hierarchical components and each component can have a default assignee and default email list to CC.
- When sending a source change for review, engineers are prompted to associate the change with a particular issue number.
- Bug triaging: common for teams at Google to regularly scan through open issues in their component(s), prioritizing them and where appropriate assigning them to particular engineers.

Programming Languages

- Software engineers at Google are strongly encouraged to program in one of five officially-approved programming languages at Google:
 - C++, Java, Python, Go, or JavaScript.
- Minimizing the number of different programming languages used reduces obstacles to code reuse and programmer collaboration.
- Google style guides for each language, ensure that code all across the company is written with similar style, layout, naming conventions, etc.
- In addition to these five languages, many specialized domain-specific languages are used for particular purposes.
 - e.g. the build language used for specifying build targets and their dependencies.
- Interoperation between these different programming languages is done mainly using Protocol Buffers.
 - Protocol Buffers is a way of encoding structured data in an efficient yet extensible way.

Google style guides

Eg: Python style guide

Use default iterators and operators for types that support them, like lists, dictionaries, and files. The built-in types define iterator methods, too. Prefer these methods to methods that return lists, except that you should not mutate a container while iterating over it.

```
Yes:  for key in adict: ...
      if key not in adict: ...
      if obj in alist: ...
      for line in afile: ...
      for k, v in adict.items(): ...
      for k, v in six.iteritems(adict): ...
```

```
No:   for key in adict.keys(): ...
      if not adict.has_key(key): ...
      for line in afile.readlines(): ...
      for k, v in dict.iteritems(): ...
```

Debugging and Profiling tools

- Google servers are linked with libraries that provide a number of tools for debugging running servers.
- In case of a server crash, a signal handler will automatically dump a stack trace to a log file, as well as saving the core file.
- If the crash was due to running out of heap memory, the server will dump stack traces of the allocation sites of a sampled subset of the live heap objects.
- There are also web interfaces for debugging that allow examining incoming and outgoing RPCs (including timing, error rates, rate limiting, etc.), changing command-line flag values (e.g. to increase logging verbosity for a particular module), resource consumption, profiling, and more.
- These tools greatly increase the overall ease of debugging to the point where it is rare to fire up a traditional debugger such as gdb.

Release Engineering

- For most teams at Google, the release engineering work is done by regular software engineers.
- Releases are done frequently for most software;
 - weekly or fortnightly releases are a common goal
 - some teams even release daily
- Made possible by automating most of the normal release engineering tasks.
- Releasing frequently helps to keep engineers motivated (it's harder to get excited about something if it won't be released until many months or even years into the future)
 - increases overall velocity by allowing more iterations, and thus more opportunities for feedback and more chances to respond to feedback, in a given time.

Release Engineering – steps of a release

1. Starts in a fresh workspace, by syncing to the change number of the latest “green” build (i.e. the last change for which all the automatic tests passed), and making a release branch.
2. The release engineer can select additional changes to be “cherry-picked”, i.e. merged from the main branch onto the release branch.
3. Then the software will be rebuilt from scratch and the tests are run.
 - a) If any tests fail, additional changes are made to fix the failures and those additional changes are cherry-picked onto the release branch, after which the software will be rebuilt and the tests rerun.
4. When the all tests pass, the built executable(s) and data file(s) are packaged up.
5. Once a candidate build has been packaged up, it is typically loaded onto a “staging” server for further integration testing by small set of users.
6. The next step is to usually roll out to one or more “canary” servers that are processing a subset of the live production traffic.
7. Finally the release can be rolled out to all servers in all data centers.
8. For very high-traffic, high-reliability services, this is done with a gradual roll-out over a period of a couple of days, to help reduce the impact of any outages due to newly introduced bugs not caught by any of the previous steps.

Launch Approval

- The launch of any user-visible change or significant design change requires approvals from a number of people outside of the core engineering team that implements the change.
- In particular approvals, are required to ensure that code complies with legal requirements, privacy requirements, security requirements, reliability requirements, business requirements, and so forth.
- Google has an internal launch approval tool that is used to track the required reviews and approvals and ensure compliance with the defined launch processes for each product.

Post-mortems

- Whenever there is a significant outage of any of our production systems, or similar mishap, the people involved are required to write a post-mortem document.
- This document describes the incident, including title, summary, impact, timeline, root cause(s), what worked/what didn't, and action items.
- The focus is on the problems, and how to avoid them in future, not on the people or apportioning blame.
- The impact section tries to quantify the effect of the incident, in terms of duration of outage, number of lost queries (or failed RPCs, etc.), and revenue.
- The timeline section gives a timeline of the events leading up to the outage and the steps taken to diagnose and rectify it.
- The what worked/what didn't section describes the lessons learnt.

Frequent Rewrites

- Most software at Google gets rewritten every few years.
- It consumes a large fraction of Google's resources.
- It also has some crucial benefits that are key to Google's agility and long-term success.
- It is typical for the requirements for a product to change significantly, as the software environment and other technology around it change, and as changes in technology or in the marketplace affect user needs, desires, and expectations.
- Software that is a few years old was designed around an older set of requirements and is typically not designed in a way that is optimal for current requirements.
- It has typically accumulated a lot of complexity.
- Rewriting code cuts away all the unnecessary accumulated complexity that was addressing requirements which may no longer be so important.
- Rewriting code is also a way of transferring knowledge and a sense of ownership to newer team members.

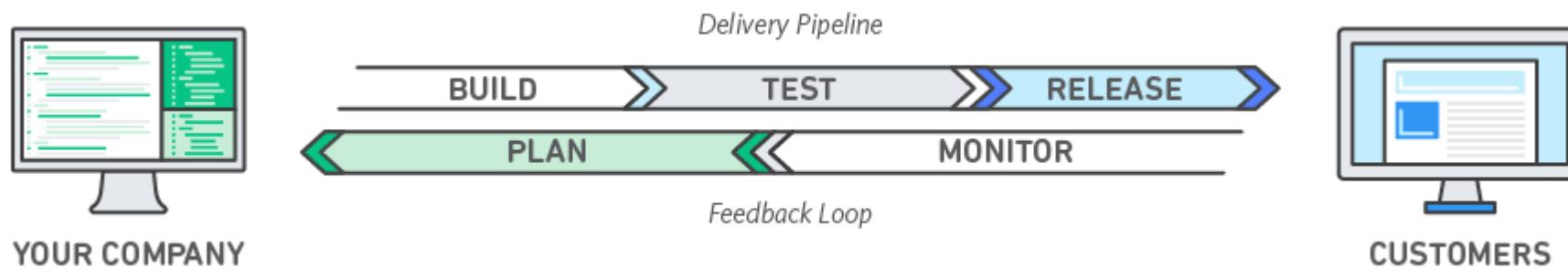
What else?

- Project management
 - Time management
 - Objectives
 - Project approval
 - Corporate reorganizations
 - ...
- People management
 - Roles
 - Facilities
 - Training
 - Transfers
 - Performance appraisals
 - ...

Rapid software development with DevOps

GAYASHAN AMARASINGHE

DevOps Model



Benefits of DevOps



SPEED



SCALE



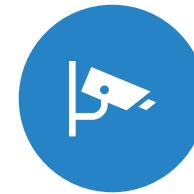
IMPROVED
COLLABORATION



RAPID DELIVERY



RELIABILITY



SECURITY

DevOps Practices

Continuous Integration (CI)

- Commit often
- Run tests and identify bugs quicker

Continuous Delivery (CD)

- Deploy to testing/production faster
- Always have deployment ready artifacts

Microservices

- Individually manageable decoupled components

Infrastructure as Code (IaC)

- Infrastructure provisioning using code
- Chef, Puppet, Terraform,...

Configuration Management

- Make configuration changes repeatable and standardized

Policy as Code

Monitoring and Logging

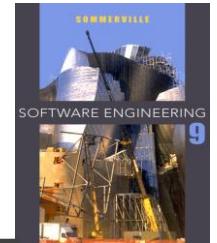
Communication and Collaboration



Assignment – Set up your own CI/CD pipeline

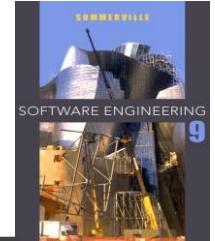
This assignment is for setting up a CI/CD pipeline for your SimpleExpenseManager project.

1. Add unit test cases that cover adding and retrieving dummy data from the used database in your SimpleExpenseManager project (given as part of the Embedded Databases assignment)
2. Follow the video guideline and set up a CI/CD pipeline for the project using GitHub actions.
3. Submit the link for your project.



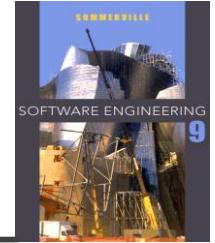
Chapter 8 – Software Testing

Lecture 1



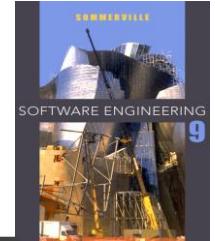
Topics covered

- ✧ Development testing
- ✧ Test-driven development
- ✧ Release testing
- ✧ User testing



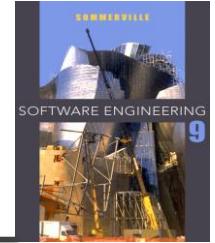
Program testing

- ✧ Testing is intended to show that a program does what it is intended to do and to discover program defects before it is put into use.
- ✧ When you test software, you execute a program using artificial data.
- ✧ You check the results of the test run for errors, anomalies or information about the program's non-functional attributes.
- ✧ Can reveal the presence of errors NOT their absence.
- ✧ Testing is part of a more general verification and validation process, which also includes static validation techniques.



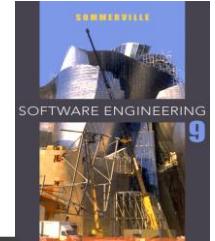
Program testing goals

- ✧ To demonstrate to the developer and the customer that the software meets its requirements.
 - For custom software, this means that there should be at least one test for every requirement in the requirements document. For generic software products, it means that there should be tests for all of the system features, plus combinations of these features, that will be incorporated in the product release.
- ✧ To discover situations in which the behavior of the software is incorrect, undesirable or does not conform to its specification.
 - Defect testing is concerned with rooting out undesirable system behavior such as system crashes, unwanted interactions with other systems, incorrect computations and data corruption.



Validation and defect testing

- ✧ The first goal leads to **validation testing**
 - You expect the system to perform correctly using a given set of test cases that reflect the system's expected use.
- ✧ The second goal leads to **defect testing**
 - The test cases are designed to expose defects. The test cases in defect testing can be deliberately obscure and need not reflect how the system is normally used.



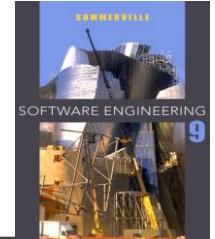
Testing process goals

✧ Validation testing

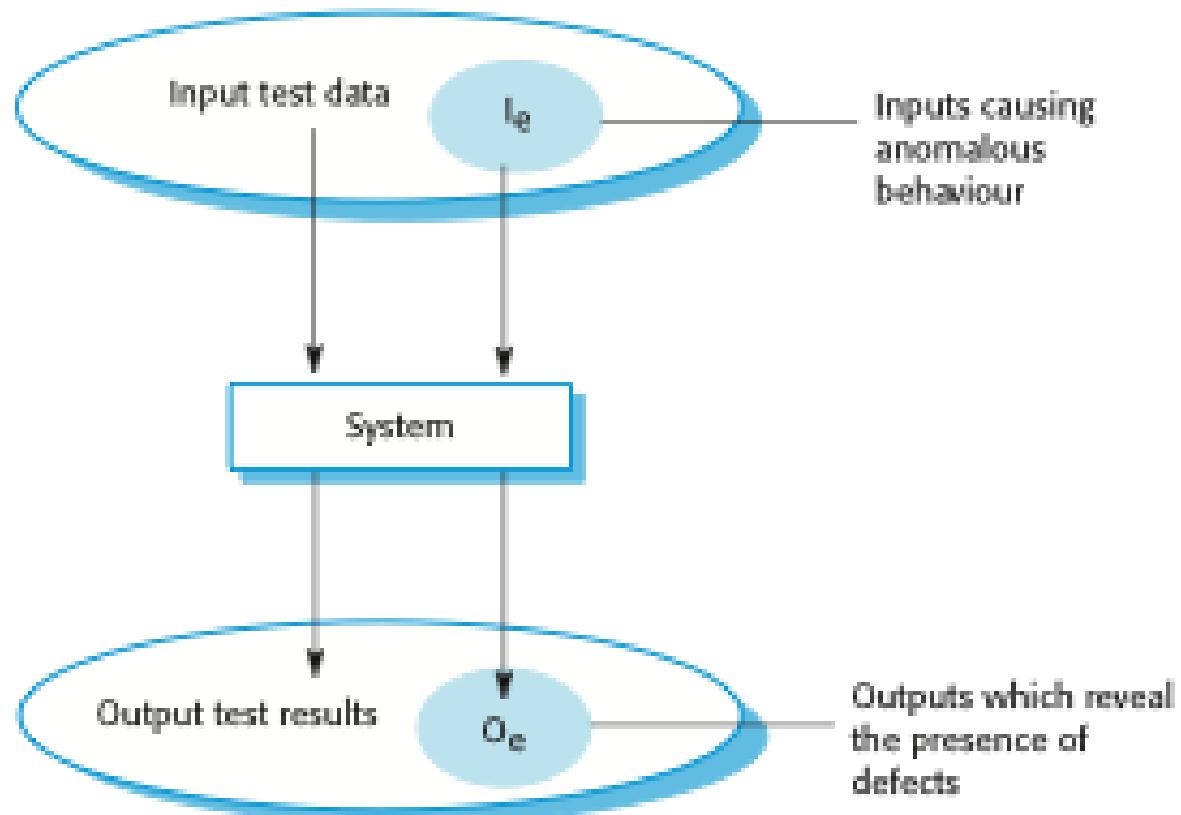
- To demonstrate to the developer and the system customer that the software meets its requirements
- A successful test shows that the system operates as intended.

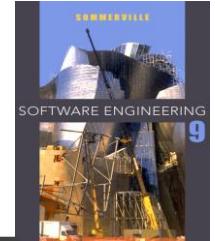
✧ Defect testing

- To discover faults or defects in the software where its behaviour is incorrect or not in conformance with its specification
- A successful test is a test that makes the system perform incorrectly and so exposes a defect in the system.



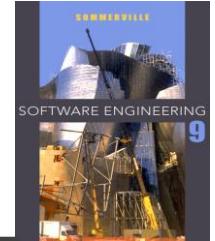
An input-output model of program testing





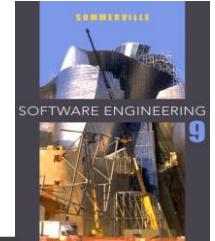
Verification vs validation

- ✧ Verification:
"Are we building the product right".
- ✧ The software should conform to its specification.
- ✧ Validation:
"Are we building the right product".
- ✧ The software should do what the user really requires.



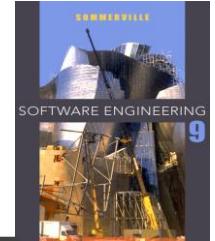
V & V confidence

- ✧ Aim of V & V is to establish confidence that the system is 'fit for purpose'.
- ✧ Depends on system's purpose, user expectations and marketing environment
 - Software purpose
 - The level of confidence depends on how critical the software is to an organisation.
 - User expectations
 - Users may have low expectations of certain kinds of software.
 - Marketing environment
 - Getting a product to market early may be more important than finding defects in the program.

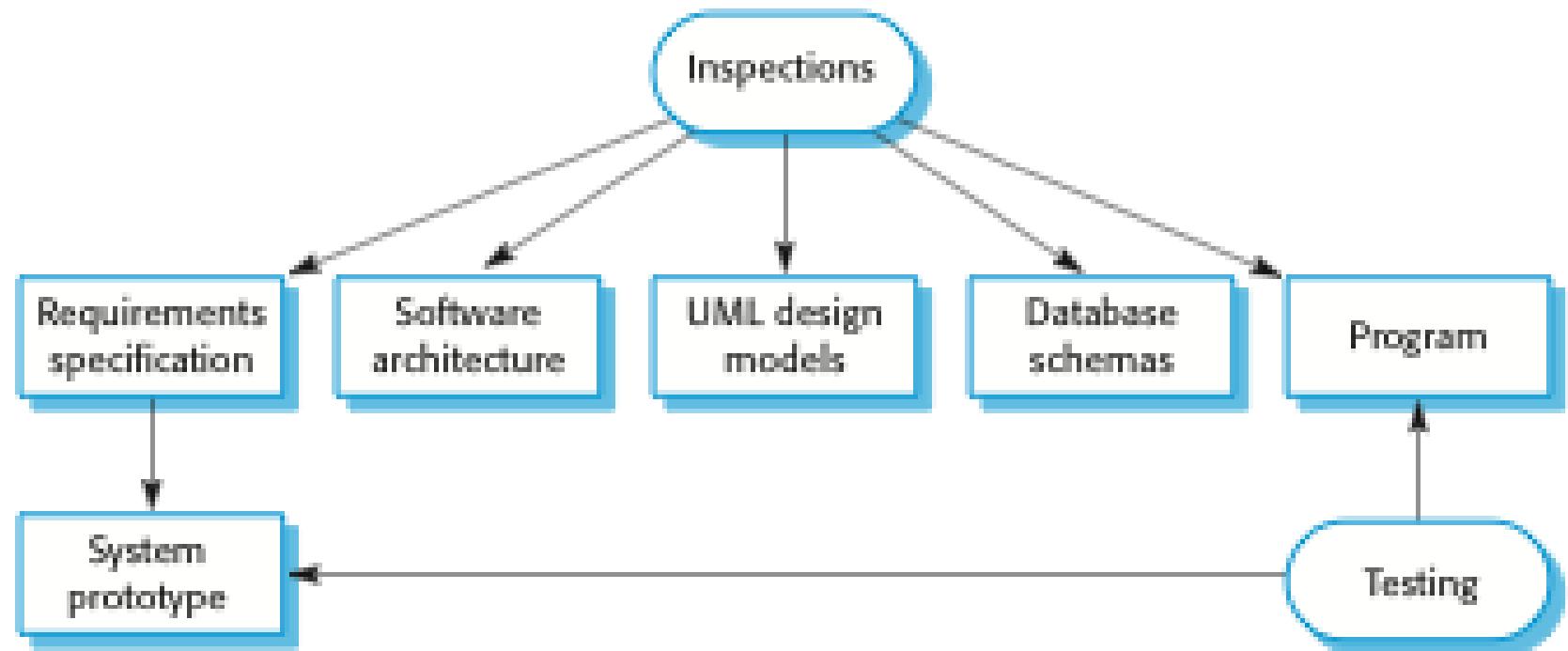


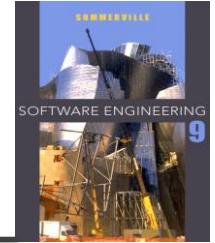
Inspections and testing

- ❖ **Software inspections** Concerned with analysis of the static system representation to discover problems (static verification)
 - May be supplement by tool-based document and code analysis.
 - Discussed in Chapter 15.
- ❖ **Software testing** Concerned with exercising and observing product behaviour (dynamic verification)
 - The system is executed with test data and its operational behaviour is observed.



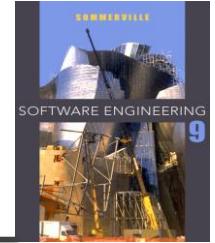
Inspections and testing





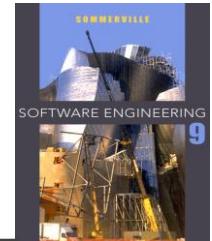
Software inspections

- ✧ These involve people examining the source representation with the aim of discovering anomalies and defects.
- ✧ Inspections not require execution of a system so may be used before implementation.
- ✧ They may be applied to any representation of the system (requirements, design, configuration data, test data, etc.).
- ✧ They have been shown to be an effective technique for discovering program errors.



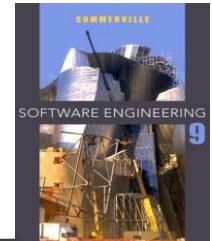
Advantages of inspections

- ✧ During testing, errors can mask (hide) other errors.
Because inspection is a static process, you don't have to be concerned with interactions between errors.
- ✧ Incomplete versions of a system can be inspected without additional costs. If a program is incomplete, then you need to develop specialized test harnesses to test the parts that are available.
- ✧ As well as searching for program defects, an inspection can also consider broader quality attributes of a program, such as compliance with standards, portability and maintainability.

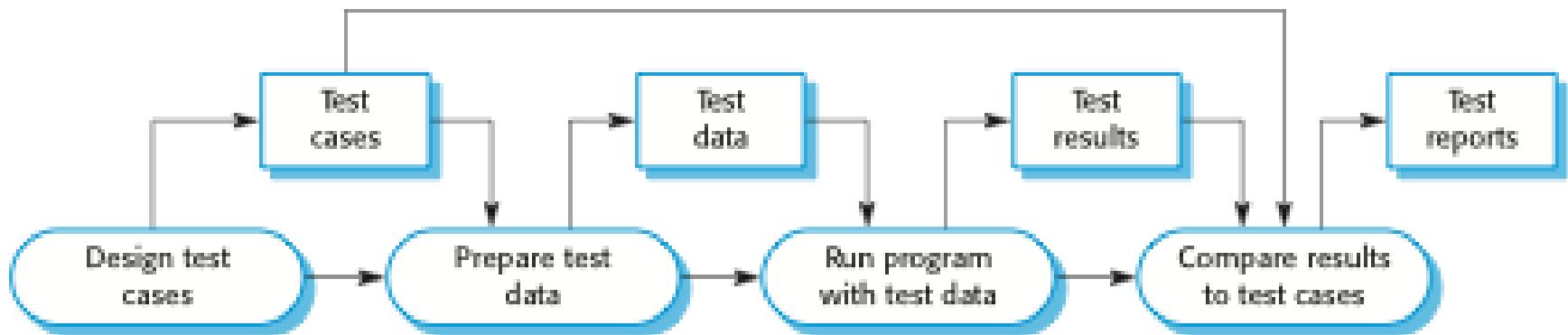


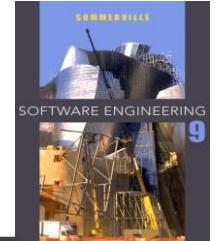
Inspections and testing

- ✧ Inspections and testing are complementary and not opposing verification techniques.
- ✧ Both should be used during the V & V process.
- ✧ Inspections can check conformance with a specification but not conformance with the customer's real requirements.
- ✧ Inspections cannot check non-functional characteristics such as performance, usability, etc.



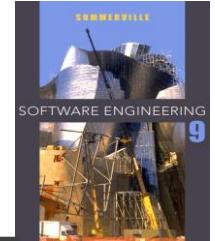
A model of the software testing process





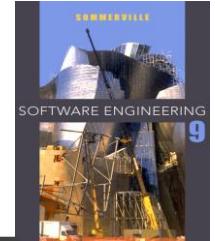
Stages of testing

- ✧ Development testing, where the system is tested during development to discover bugs and defects.
- ✧ Release testing, where a separate testing team test a complete version of the system before it is released to users.
- ✧ User testing, where users or potential users of a system test the system in their own environment.



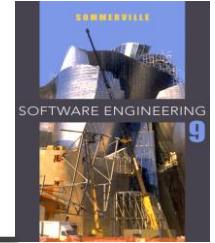
Development testing

- ✧ Development testing includes all testing activities that are carried out by the team developing the system.
 - Unit testing, where individual program units or object classes are tested. Unit testing should focus on testing the functionality of objects or methods.
 - Component testing, where several individual units are integrated to create composite components. Component testing should focus on testing component interfaces.
 - System testing, where some or all of the components in a system are integrated and the system is tested as a whole. System testing should focus on testing component interactions.



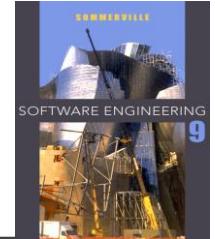
Unit testing

- ✧ Unit testing is the process of testing individual components in isolation.
- ✧ It is a defect testing process.
- ✧ Units may be:
 - Individual functions or methods within an object
 - Object classes with several attributes and methods
 - Composite components with defined interfaces used to access their functionality.



Object class testing

- ✧ Complete test coverage of a class involves
 - Testing all operations associated with an object
 - Setting and interrogating all object attributes
 - Exercising the object in all possible states.
- ✧ Inheritance makes it more difficult to design object class tests as the information to be tested is not localised.

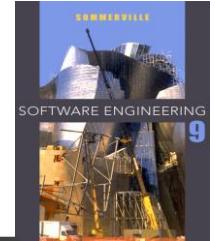


The weather station object interface

WeatherStation

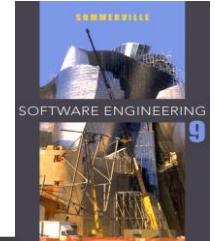
identifier

reportWeather ()
reportStatus ()
powerSave (instruments)
remoteControl (commands)
reconfigure (commands)
restart (instruments)
shutdown (instruments)



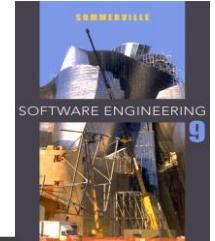
Weather station testing

- ✧ Need to define test cases for reportWeather, calibrate, test, startup and shutdown.
- ✧ Using a state model, identify sequences of state transitions to be tested and the event sequences to cause these transitions
- ✧ For example:
 - Shutdown -> Running-> Shutdown
 - Configuring-> Running-> Testing -> Transmitting -> Running
 - Running-> Collecting-> Running-> Summarizing -> Transmitting -> Running



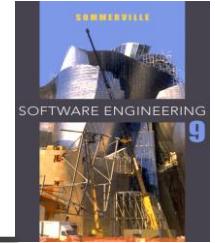
Automated testing

- ✧ Whenever possible, unit testing should be automated so that tests are run and checked without manual intervention.
- ✧ In automated unit testing, you make use of a test automation framework (such as JUnit) to write and run your program tests.
- ✧ Unit testing frameworks provide generic test classes that you extend to create specific test cases. They can then run all of the tests that you have implemented and report, often through some GUI, on the success or otherwise of the tests.



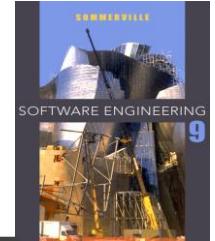
Automated test components

- ✧ A setup part, where you initialize the system with the test case, namely the inputs and expected outputs.
- ✧ A call part, where you call the object or method to be tested.
- ✧ An assertion part where you compare the result of the call with the expected result. If the assertion evaluates to true, the test has been successful if false, then it has failed.



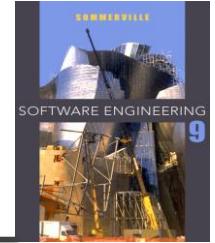
Unit test effectiveness

- ✧ The test cases should show that, when used as expected, the component that you are testing does what it is supposed to do.
- ✧ If there are defects in the component, these should be revealed by test cases.
- ✧ This leads to 2 types of unit test case:
 - The first of these should reflect normal operation of a program and should show that the component works as expected.
 - The other kind of test case should be based on testing experience of where common problems arise. It should use abnormal inputs to check that these are properly processed and do not crash the component.



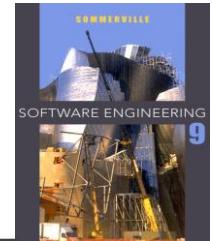
Testing strategies

- ✧ Partition testing, where you identify groups of inputs that have common characteristics and should be processed in the same way.
 - You should choose tests from within each of these groups.
- ✧ Guideline-based testing, where you use testing guidelines to choose test cases.
 - These guidelines reflect previous experience of the kinds of errors that programmers often make when developing components.

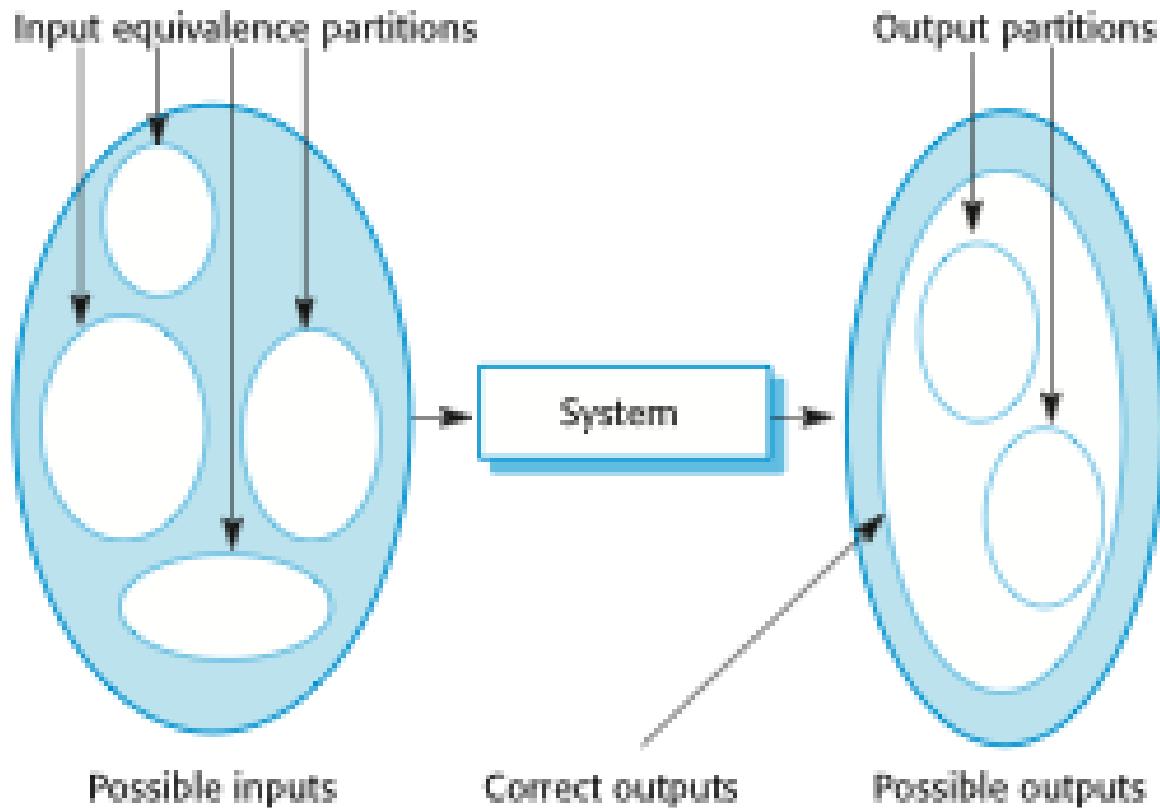


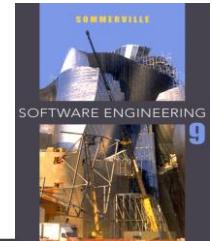
Partition testing

- ✧ Input data and output results often fall into different classes where all members of a class are related.
- ✧ Each of these classes is an **equivalence partition** or domain where the program behaves in an equivalent way for each class member.
- ✧ Test cases should be chosen from each partition.

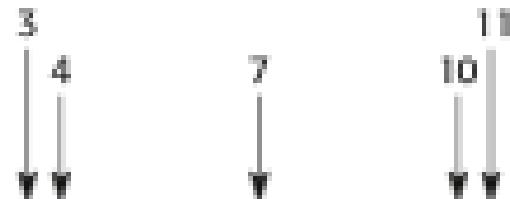


Equivalence partitioning





Equivalence partitions

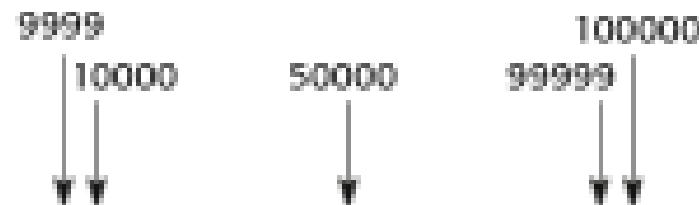


Less than 4

Between 4 and 10

More than 10

Number of input values

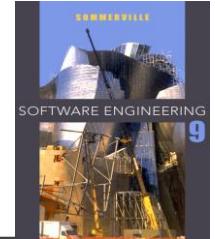


Less than 10000

Between 10000 and 99999

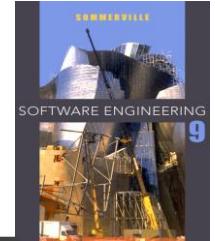
More than 99999

Input values



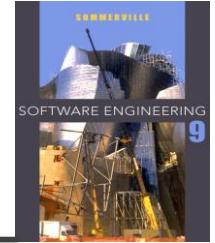
Testing guidelines (sequences)

- ✧ Test software with sequences which have only a single value.
- ✧ Use sequences of different sizes in different tests.
- ✧ Derive tests so that the first, middle and last elements of the sequence are accessed.
- ✧ Test with sequences of zero length.



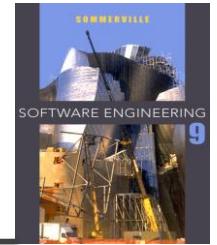
General testing guidelines

- ✧ Choose inputs that force the system to generate all error messages
- ✧ Design inputs that cause input buffers to overflow
- ✧ Repeat the same input or series of inputs numerous times
- ✧ Force invalid outputs to be generated
- ✧ Force computation results to be too large or too small.



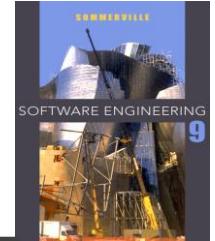
Key points

- ✧ Testing can only show the presence of errors in a program. It cannot demonstrate that there are no remaining faults.
- ✧ Development testing is the responsibility of the software development team. A separate team should be responsible for testing a system before it is released to customers.
- ✧ Development testing includes unit testing, in which you test individual objects and methods component testing in which you test related groups of objects and system testing, in which you test partial or complete systems.



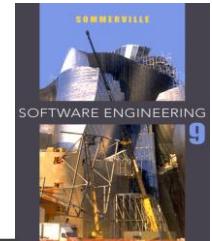
Chapter 8 – Software Testing

Lecture 2

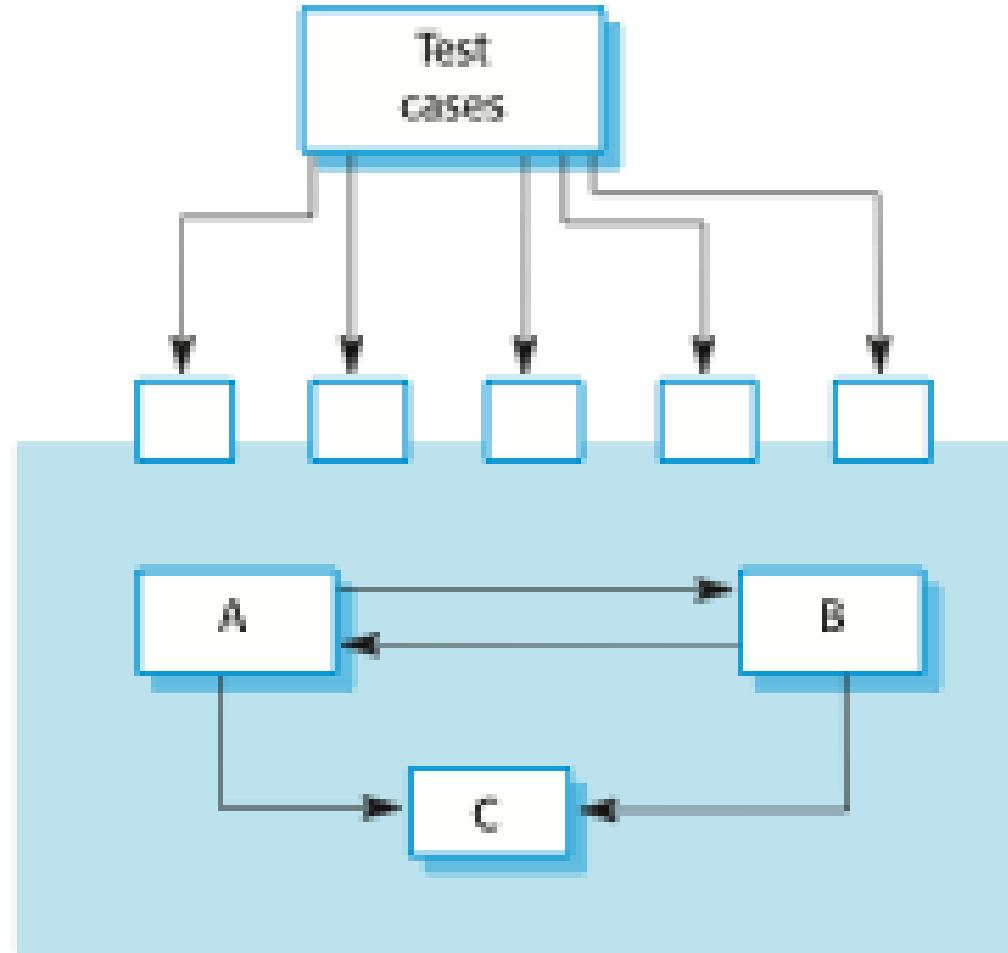


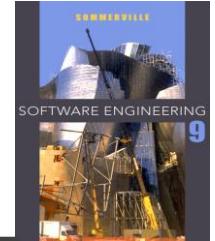
Component testing

- ✧ Software components are often composite components that are made up of several interacting objects.
 - For example, in the weather station system, the reconfiguration component includes objects that deal with each aspect of the reconfiguration.
- ✧ You access the functionality of these objects through the defined component interface.
- ✧ Testing composite components should therefore focus on showing that the component interface behaves according to its specification.
 - You can assume that unit tests on the individual objects within the component have been completed.



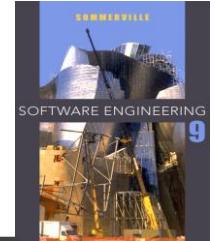
Interface testing





Interface testing

- ✧ Objectives are to detect faults due to interface errors or invalid assumptions about interfaces.
- ✧ Interface types
 - **Parameter interfaces** Data passed from one method or procedure to another.
 - **Shared memory interfaces** Block of memory is shared between procedures or functions.
 - **Procedural interfaces** Sub-system encapsulates a set of procedures to be called by other sub-systems.
 - **Message passing interfaces** Sub-systems request services from other sub-systems



Interface errors

✧ Interface misuse

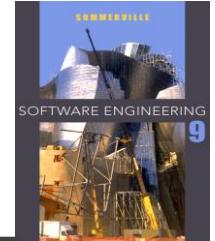
- A calling component calls another component and makes an error in its use of its interface e.g. parameters in the wrong order.

✧ Interface misunderstanding

- A calling component embeds assumptions about the behaviour of the called component which are incorrect.

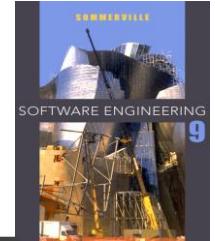
✧ Timing errors

- The called and the calling component operate at different speeds and out-of-date information is accessed.



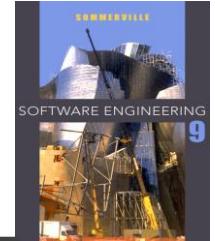
Interface testing guidelines

- ✧ Design tests so that parameters to a called procedure are at the extreme ends of their ranges.
- ✧ Always test pointer parameters with null pointers.
- ✧ Design tests which cause the component to fail.
- ✧ Use stress testing in message passing systems.
- ✧ In shared memory systems, vary the order in which components are activated.



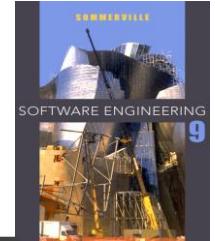
System testing

- ✧ System testing during development involves integrating components to create a version of the system and then testing the integrated system.
- ✧ The focus in system testing is testing the interactions between components.
- ✧ System testing checks that components are compatible, interact correctly and transfer the right data at the right time across their interfaces.
- ✧ System testing tests the emergent behaviour of a system.



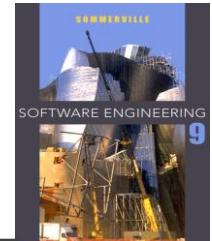
System and component testing

- ✧ During system testing, reusable components that have been separately developed and off-the-shelf systems may be integrated with newly developed components. The complete system is then tested.
- ✧ Components developed by different team members or sub-teams may be integrated at this stage. System testing is a collective rather than an individual process.
 - In some companies, system testing may involve a separate testing team with no involvement from designers and programmers.

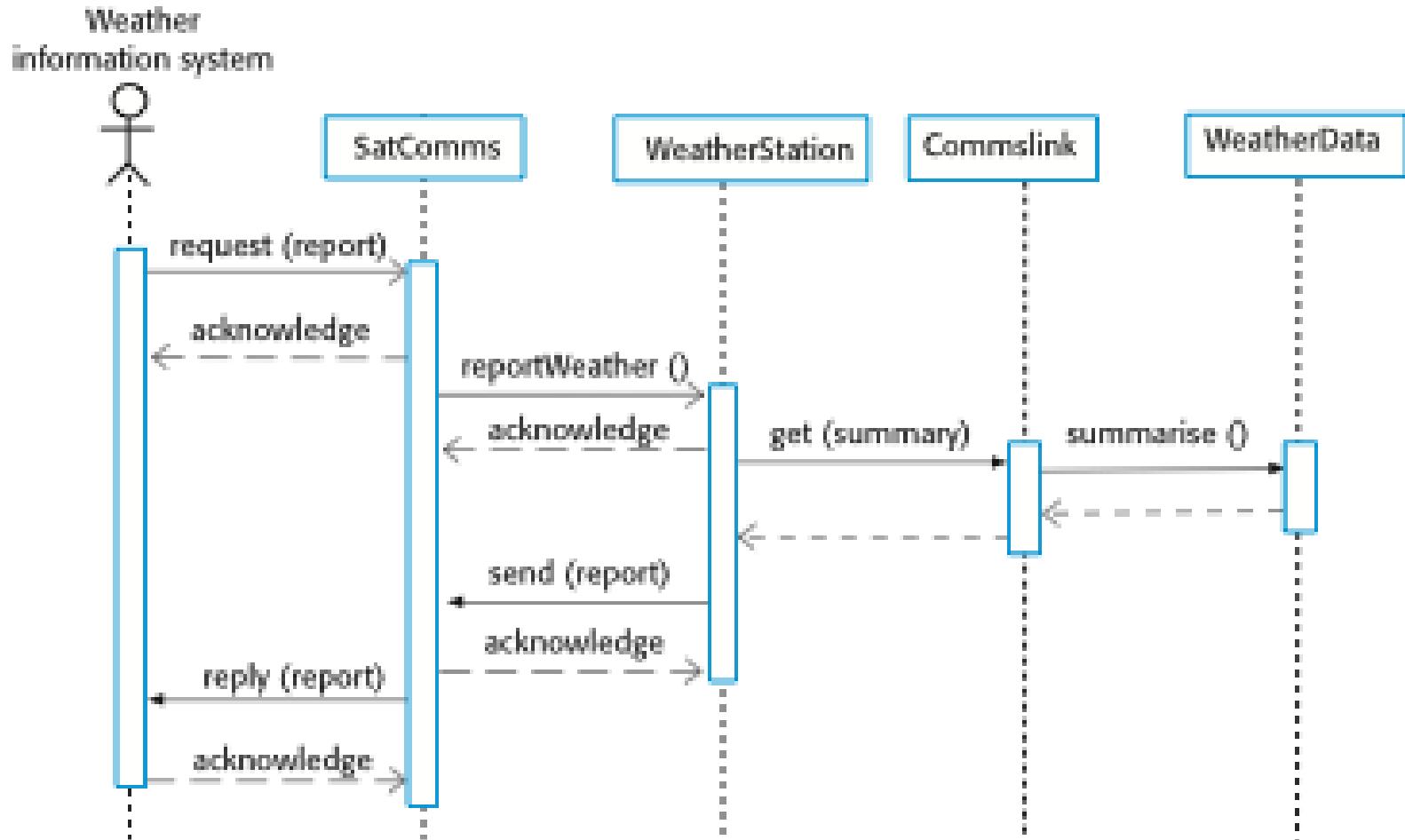


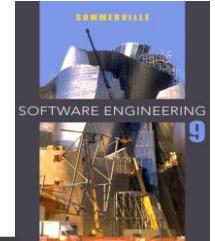
Use-case testing

- ✧ The use-cases developed to identify system interactions can be used as a basis for system testing.
- ✧ Each use case usually involves several system components so testing the use case forces these interactions to occur.
- ✧ The sequence diagrams associated with the use case documents the components and interactions that are being tested.



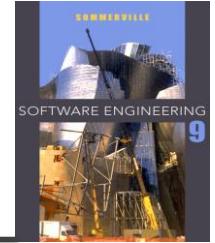
Collect weather data sequence chart





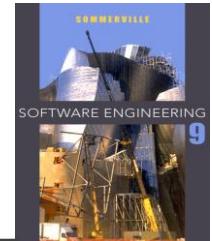
Testing policies

- ✧ Exhaustive system testing is impossible so testing policies which define the required system test coverage may be developed.
- ✧ Examples of testing policies:
 - All system functions that are accessed through menus should be tested.
 - Combinations of functions (e.g. text formatting) that are accessed through the same menu must be tested.
 - Where user input is provided, all functions must be tested with both correct and incorrect input.

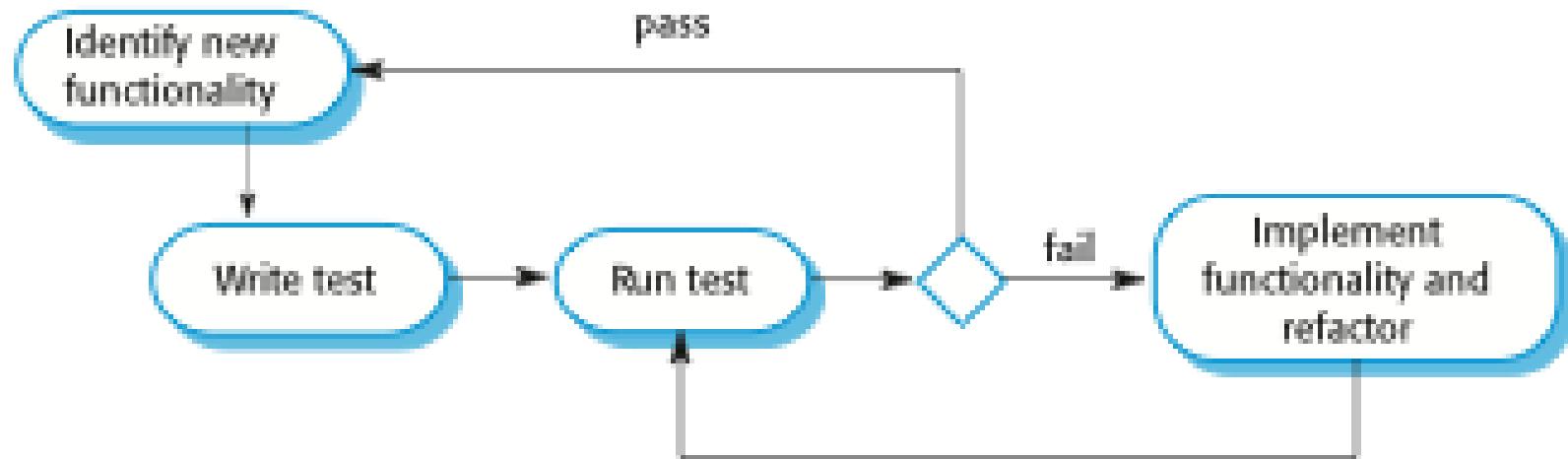


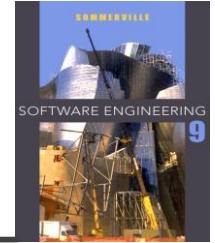
Test-driven development

- ✧ Test-driven development (TDD) is an approach to program development in which you inter-leave testing and code development.
- ✧ Tests are written before code and ‘passing’ the tests is the critical driver of development.
- ✧ You develop code incrementally, along with a test for that increment. You don’t move on to the next increment until the code that you have developed passes its test.
- ✧ TDD was introduced as part of agile methods such as Extreme Programming. However, it can also be used in plan-driven development processes.



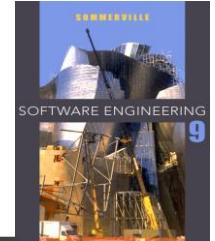
Test-driven development





TDD process activities

- ✧ Start by identifying the increment of functionality that is required. This should normally be small and implementable in a few lines of code.
- ✧ Write a test for this functionality and implement this as an automated test.
- ✧ Run the test, along with all other tests that have been implemented. Initially, you have not implemented the functionality so the new test will fail.
- ✧ Implement the functionality and re-run the test.
- ✧ Once all tests run successfully, you move on to implementing the next chunk of functionality.



Benefits of test-driven development

✧ Code coverage

- Every code segment that you write has at least one associated test so all code written has at least one test.

✧ Regression testing

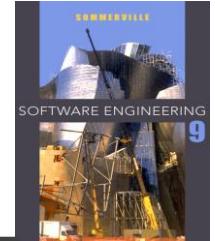
- A regression test suite is developed incrementally as a program is developed.

✧ Simplified debugging

- When a test fails, it should be obvious where the problem lies. The newly written code needs to be checked and modified.

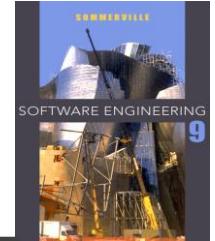
✧ System documentation

- The tests themselves are a form of documentation that describe what the code should be doing.



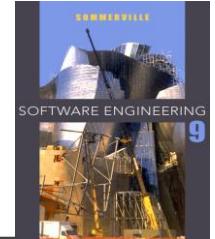
Regression testing

- ✧ Regression testing is testing the system to check that changes have not ‘broken’ previously working code.
- ✧ In a manual testing process, regression testing is expensive but, with automated testing, it is simple and straightforward. All tests are rerun every time a change is made to the program.
- ✧ Tests must run ‘successfully’ before the change is committed.



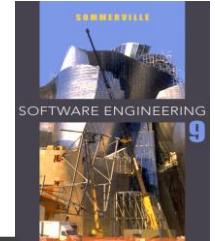
Release testing

- ✧ Release testing is the process of testing a particular release of a system that is intended for use outside of the development team.
- ✧ The primary goal of the release testing process is to convince the supplier of the system that it is good enough for use.
 - Release testing, therefore, has to show that the system delivers its specified functionality, performance and dependability, and that it does not fail during normal use.
- ✧ Release testing is usually a black-box testing process where tests are only derived from the system specification.



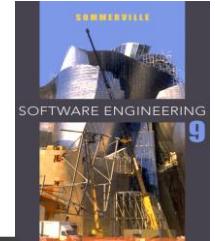
Release testing and system testing

- ✧ Release testing is a form of system testing.
- ✧ Important differences:
 - A separate team that has not been involved in the system development, should be responsible for release testing.
 - System testing by the development team should focus on discovering bugs in the system (defect testing). The objective of release testing is to check that the system meets its requirements and is good enough for external use (validation testing).



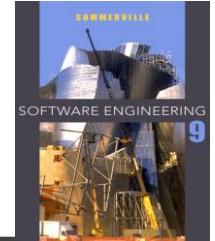
Requirements based testing

- ✧ Requirements-based testing involves examining each requirement and developing a test or tests for it.
- ✧ MHC-PMS requirements:
 - If a patient is known to be allergic to any particular medication, then prescription of that medication shall result in a warning message being issued to the system user.
 - If a prescriber chooses to ignore an allergy warning, they shall provide a reason why this has been ignored.



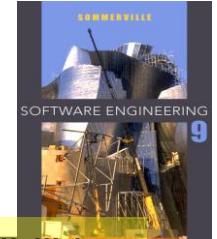
Requirements tests

- ✧ Set up a patient record with no known allergies. Prescribe medication for allergies that are known to exist. Check that a warning message is not issued by the system.
- ✧ Set up a patient record with a known allergy. Prescribe the medication to that the patient is allergic to, and check that the warning is issued by the system.
- ✧ Set up a patient record in which allergies to two or more drugs are recorded. Prescribe both of these drugs separately and check that the correct warning for each drug is issued.
- ✧ Prescribe two drugs that the patient is allergic to. Check that two warnings are correctly issued.
- ✧ Prescribe a drug that issues a warning and overrule that warning. Check that the system requires the user to provide information explaining why the warning was overruled.



Features tested by scenario

- ✧ Authentication by logging on to the system.
- ✧ Downloading and uploading of specified patient records to a laptop.
- ✧ Home visit scheduling.
- ✧ Encryption and decryption of patient records on a mobile device.
- ✧ Record retrieval and modification.
- ✧ Links with the drugs database that maintains side-effect information.
- ✧ The system for call prompting.



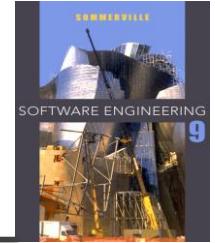
A usage scenario for the MHC-PMS

Kate is a nurse who specializes in mental health care. One of her responsibilities is to visit patients at home to check that their treatment is effective and that they are not suffering from medication side -effects.

On a day for home visits, Kate logs into the MHC-PMS and uses it to print her schedule of home visits for that day, along with summary information about the patients to be visited. She requests that the records for these patients be downloaded to her laptop. She is prompted for her key phrase to encrypt the records on the laptop.

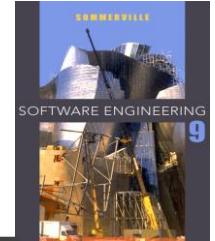
One of the patients that she visits is Jim, who is being treated with medication for depression. Jim feels that the medication is helping him but believes that it has the side -effect of keeping him awake at night. Kate looks up Jim's record and is prompted for her key phrase to decrypt the record. She checks the drug prescribed and queries its side effects. Sleeplessness is a known side effect so she notes the problem in Jim's record and suggests that he visits the clinic to have his medication changed. He agrees so Kate enters a prompt to call him when she gets back to the clinic to make an appointment with a physician. She ends the consultation and the system re-encrypts Jim's record.

After, finishing her consultations, Kate returns to the clinic and uploads the records of patients visited to the database. The system generates a call list for Kate of those patients who she has to contact for follow-up information and make clinic appointments.



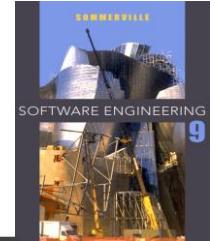
Performance testing

- ✧ Part of release testing may involve testing the emergent properties of a system, such as performance and reliability.
- ✧ Tests should reflect the profile of use of the system.
- ✧ Performance tests usually involve planning a series of tests where the load is steadily increased until the system performance becomes unacceptable.
- ✧ Stress testing is a form of performance testing where the system is deliberately overloaded to test its failure behaviour.



User testing

- ✧ User or customer testing is a stage in the testing process in which users or customers provide input and advice on system testing.
- ✧ User testing is essential, even when comprehensive system and release testing have been carried out.
 - The reason for this is that influences from the user's working environment have a major effect on the reliability, performance, usability and robustness of a system. These cannot be replicated in a testing environment.



Types of user testing

✧ Alpha testing

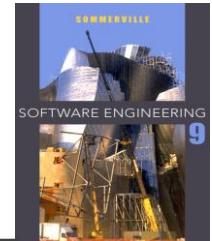
- Users of the software work with the development team to test the software at the developer's site.

✧ Beta testing

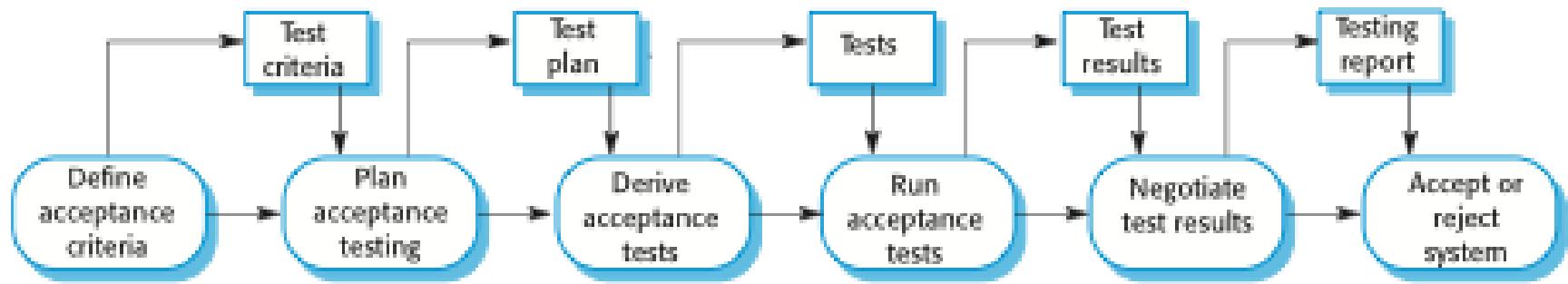
- A release of the software is made available to users to allow them to experiment and to raise problems that they discover with the system developers.

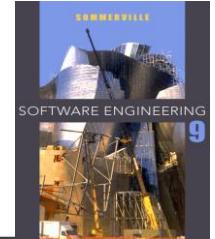
✧ Acceptance testing

- Customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment. Primarily for custom systems.



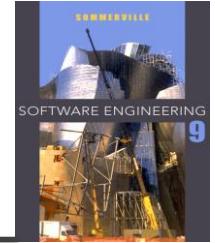
The acceptance testing process





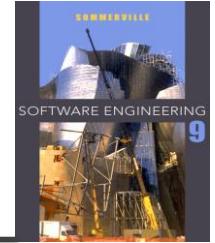
Stages in the acceptance testing process

- ✧ Define acceptance criteria
- ✧ Plan acceptance testing
- ✧ Derive acceptance tests
- ✧ Run acceptance tests
- ✧ Negotiate test results
- ✧ Reject/accept system



Agile methods and acceptance testing

- ✧ In agile methods, the user/customer is part of the development team and is responsible for making decisions on the acceptability of the system.
- ✧ Tests are defined by the user/customer and are integrated with other tests in that they are run automatically when changes are made.
- ✧ There is no separate acceptance testing process.
- ✧ Main problem here is whether or not the embedded user is 'typical' and can represent the interests of all system stakeholders.



Key points

- ✧ When testing software, you should try to ‘break’ the software by using experience and guidelines to choose types of test case that have been effective in discovering defects in other systems.
- ✧ Wherever possible, you should write automated tests. The tests are embedded in a program that can be run every time a change is made to a system.
- ✧ Test-first development is an approach to development where tests are written before the code to be tested.
- ✧ Scenario testing involves inventing a typical usage scenario and using this to derive test cases.
- ✧ Acceptance testing is a user testing process where the aim is to decide if the software is good enough to be deployed and used in its operational environment.

Application Deployment Strategies/Patterns

GAYASHAN AMARASINGHE

Why do you
need a
strategy?

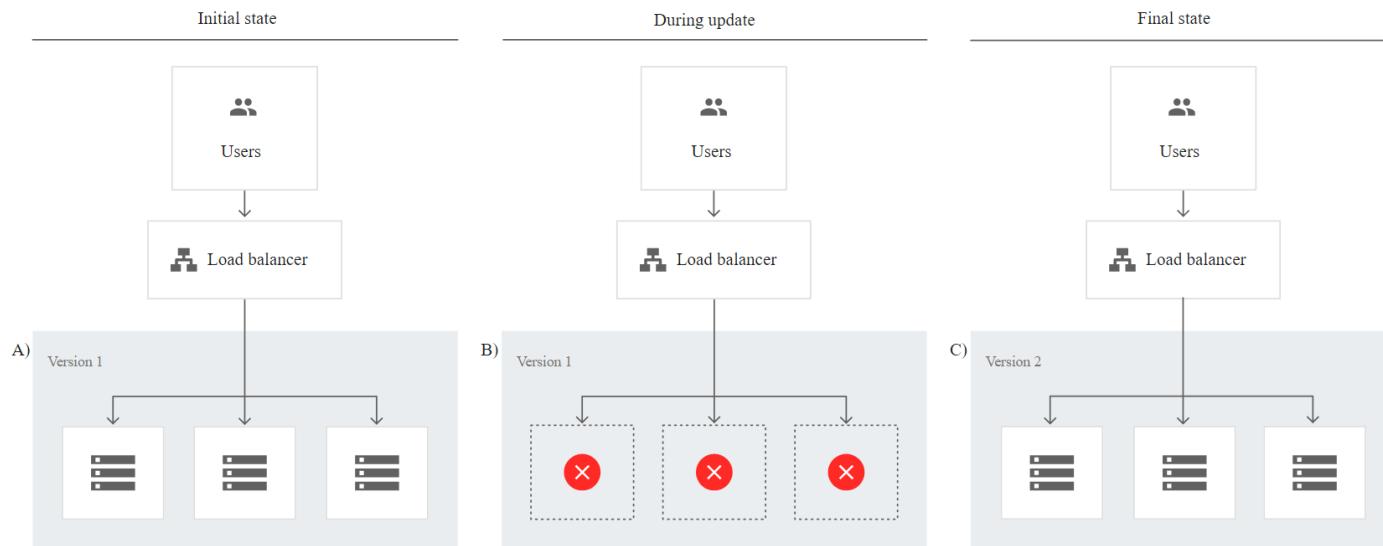


Deployment Strategies/Patterns

- Recreate
- Rolling update
- Blue/Green
- Canary
- A/B testing
- Shadow

<https://cloud.google.com/architecture/application-deployment-and-testing-strategies>

Recreate Pattern



Pros:

- Simplicity
- Application state is renewed

Cons:

- Downtime

Rolling update Pattern

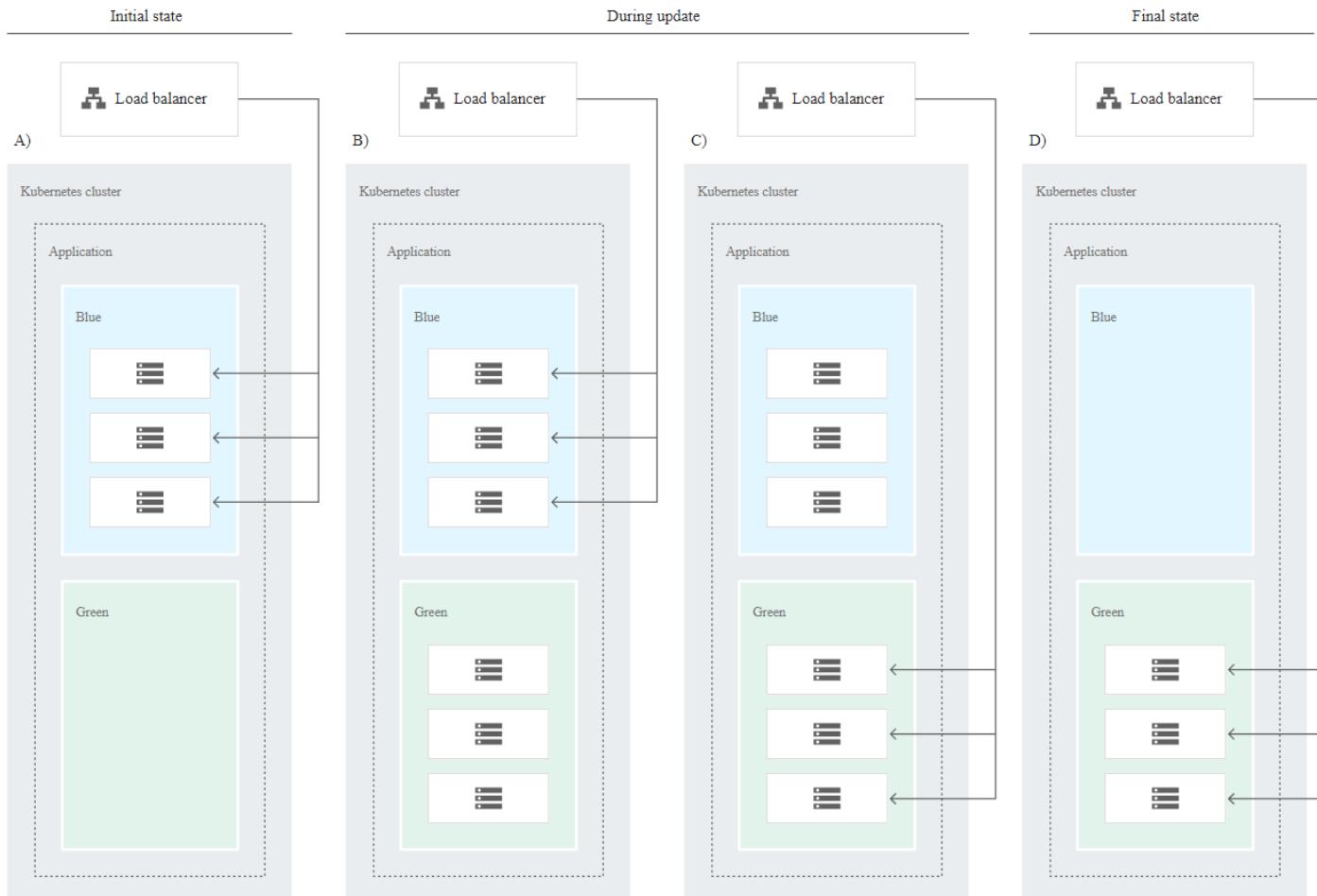


Pros:

- Easy to set up
- Versions are slowly released
- No downtime

Cons:

- Rollback is expensive
- No control over traffic



Blue/Green Deployment Pattern

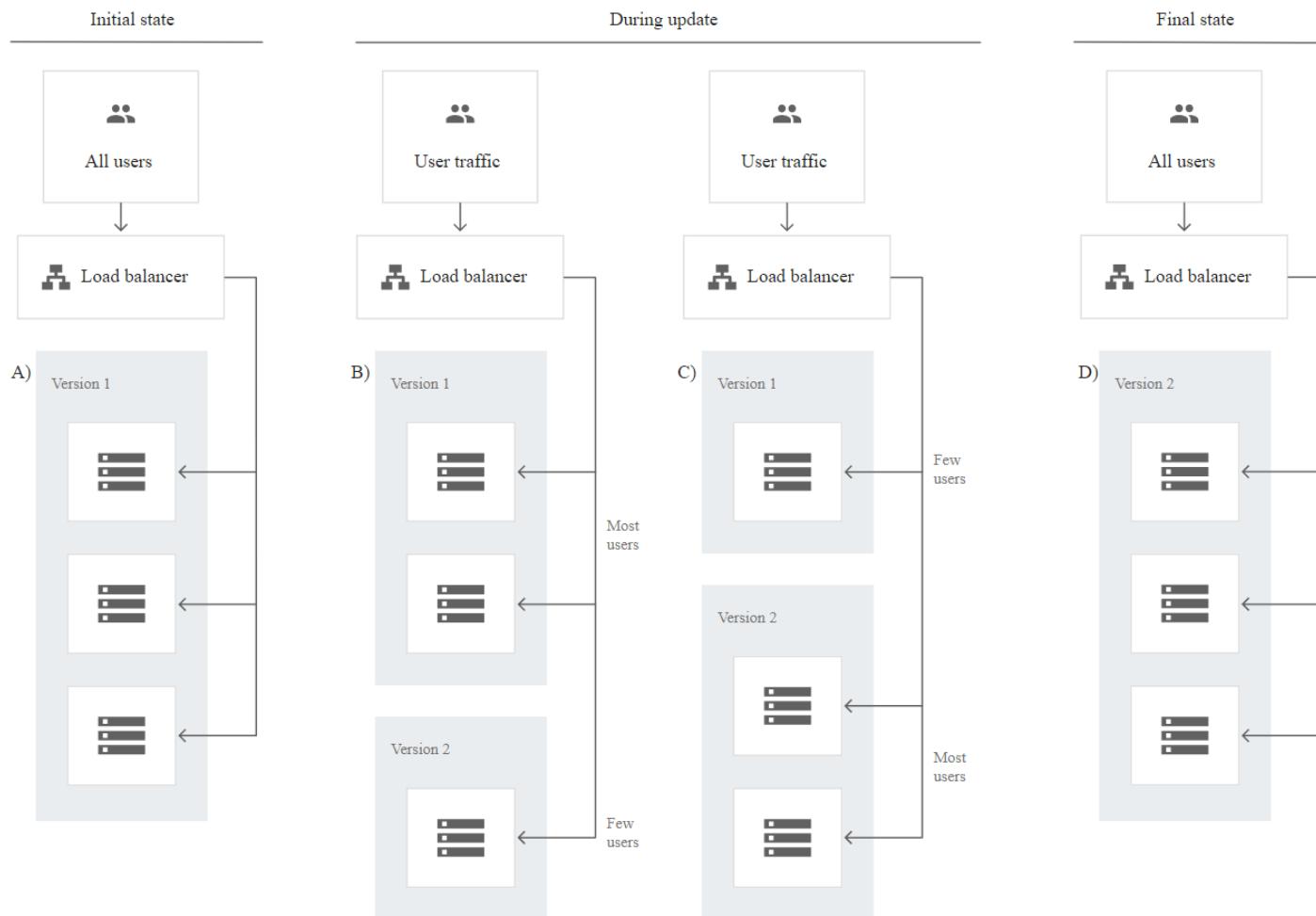
Pros:

- Zero downtime
- Instant rollback
- Avoid versioning mismatch issues

Cons:

- Double the resources are required
- Difficult to handle stateful applications

Canary Test Pattern



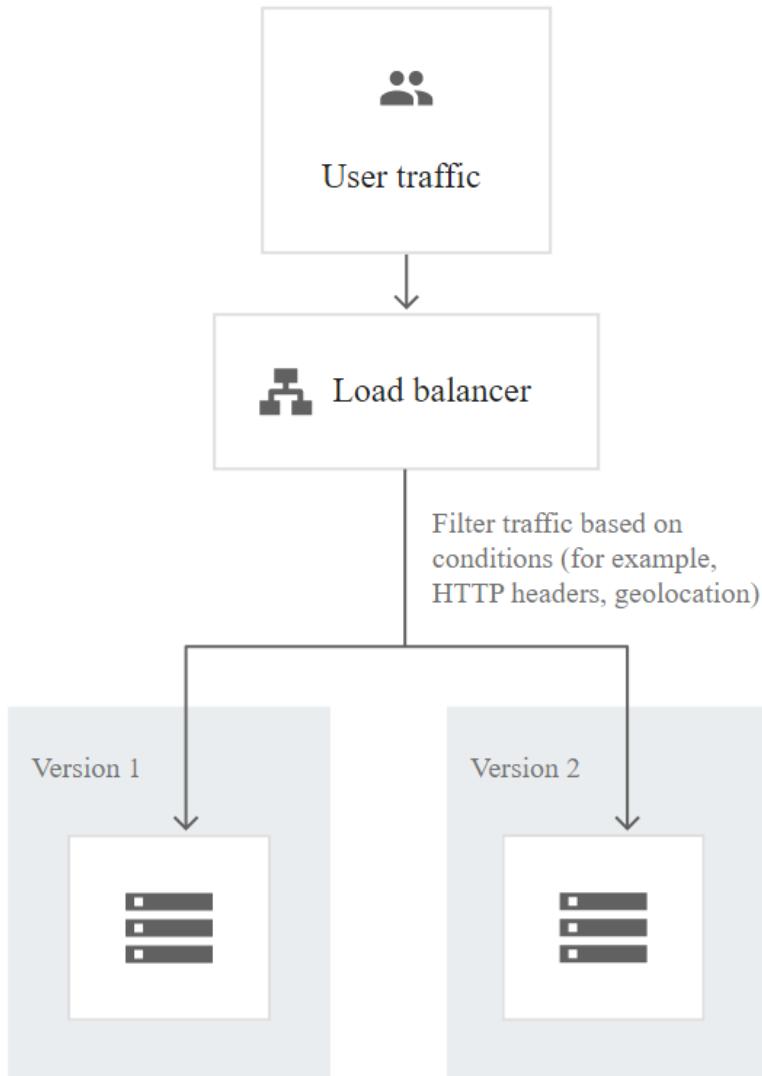
Pros:

- Fast rollback
- Convenient for performance/bug monitoring

Cons:

- Slow rollout

A/B Test Pattern

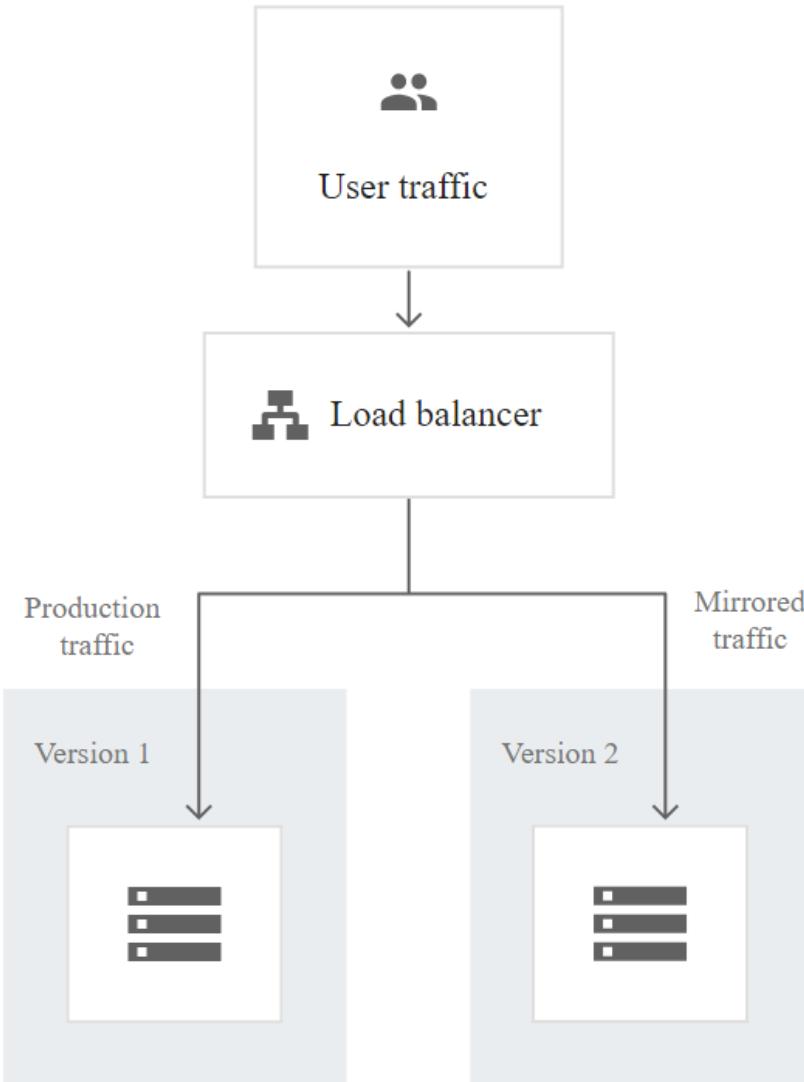


Pros:

- Full control over the application traffic

Cons:

- Requires infrastructure support
- Difficult to troubleshoot



Shadow Test Pattern

Pros:

- Performance testing with production traffic
- No impact on the user
- No need to rollout until stable

Cons:

- Expensive
- Can be misleading
- Complex to set up

Deployment or testing pattern	Zero downtime	Real production traffic testing	Releasing to users based on conditions	Rollback duration	Impact on hardware and cloud costs
Recreate Version 1 is terminated, and Version 2 is rolled out.	x	x	x	Fast but disruptive because of downtime	No extra setup required
Rolling update Version 2 is gradually rolled out and replaces Version 1.	✓	x	x	Slow	Can require extra setup for surge upgrades
Blue/green Version 2 is released alongside Version 1; the traffic is switched to Version 2 after it is tested.	✓	x	x	Instant	Need to maintain blue and green environments simultaneously
Canary Version 2 is released to a subset of users, followed by a full rollout.	✓	✓	x	Fast	No extra setup required
A/B Version 2 is released, under specific conditions, to a subset of users.	✓	✓	✓	Fast	No extra setup required
Shadow Version 2 receives real-world traffic without impacting user requests.	✓	✓	x	Does not apply	Need to maintain parallel environments in order to capture and replay user requests

Best practices?



Backward compatibility



Continuous integration/continuous deployment (CI/CD)



Automation



Operating environments and configuration management



Rollback strategy



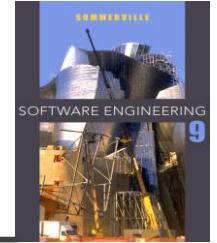
Post-deployment monitoring

Further reading/ watching

- Kubernetes and the challenges of continuous software delivery:
<https://cloud.google.com/architecture/addressing-continuous-delivery-challenges-in-a-kubernetes-world>

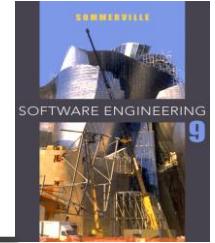
- Choosing the right modern development strategy:
<https://www.youtube.com/watch?v=-55YIDf0Z-E>

- Tutorial: <https://cloud.google.com/architecture/implementing-deployment-and-testing-strategies-on-gke>



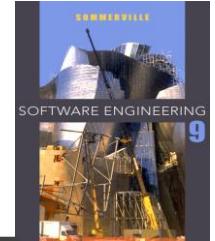
Chapter 22 – Project Management

Lecture 1



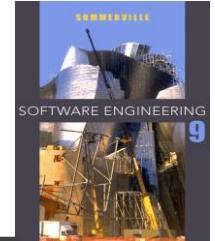
Topics covered

- ✧ Risk management
- ✧ Managing people
- ✧ Teamwork



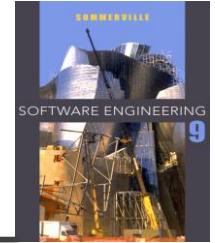
Software project management

- ✧ Concerned with activities involved in ensuring that software is delivered on time and on schedule and in accordance with the requirements of the organisations developing and procuring the software.
- ✧ Project management is needed because software development is always subject to budget and schedule constraints that are set by the organisation developing the software.



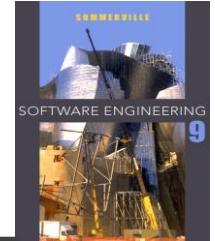
Success criteria

- ✧ Deliver the software to the customer at the agreed time.
- ✧ Keep overall costs within budget.
- ✧ Deliver software that meets the customer's expectations.
- ✧ Maintain a happy and well-functioning development team.



Software management distinctions

- ✧ The product is intangible.
 - Software cannot be seen or touched. Software project managers cannot see progress by simply looking at the artefact that is being constructed.
- ✧ Many software projects are 'one-off' projects.
 - Large software projects are usually different in some ways from previous projects. Even managers who have lots of previous experience may find it difficult to anticipate problems.
- ✧ Software processes are variable and organization specific.
 - We still cannot reliably predict when a particular software process is likely to lead to development problems.



Management activities

✧ *Project planning*

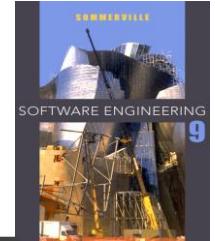
- Project managers are responsible for planning, estimating and scheduling project development and assigning people to tasks.

✧ *Reporting*

- Project managers are usually responsible for reporting on the progress of a project to customers and to the managers of the company developing the software.

✧ *Risk management*

- Project managers assess the risks that may affect a project, monitor these risks and take action when problems arise.



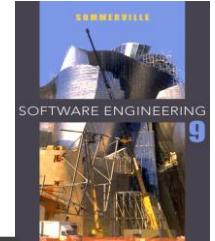
Management activities

✧ *People management*

- Project managers have to choose people for their team and establish ways of working that leads to effective team performance

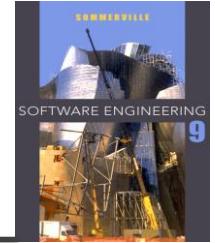
✧ *Proposal writing*

- The first stage in a software project may involve writing a proposal to win a contract to carry out an item of work. The proposal describes the objectives of the project and how it will be carried out.



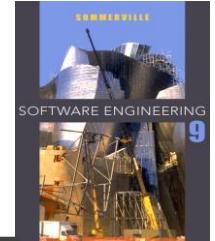
Risk management

- ✧ Risk management is concerned with identifying risks and drawing up plans to minimise their effect on a project.
- ✧ A risk is a probability that some adverse circumstance will occur
 - Project risks affect schedule or resources;
 - Product risks affect the quality or performance of the software being developed;
 - Business risks affect the organisation developing or procuring the software.



Examples of common project, product, and business risks

Risk	Affects	Description
Staff turnover	Project	Experienced staff will leave the project before it is finished.
Management change	Project	There will be a change of organizational management with different priorities.
Hardware unavailability	Project	Hardware that is essential for the project will not be delivered on schedule.
Requirements change	Project and product	There will be a larger number of changes to the requirements than anticipated.
Specification delays	Project and product	Specifications of essential interfaces are not available on schedule.
Size underestimate	Project and product	The size of the system has been underestimated.
CASE tool underperformance	Product	CASE tools, which support the project, do not perform as anticipated.
Technology change	Business	The underlying technology on which the system is built is superseded by new technology.
Product competition	Business	A competitive product is marketed before the system is completed.



The risk management process

✧ Risk identification

- Identify project, product and business risks;

✧ Risk analysis

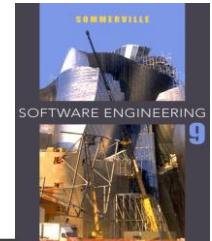
- Assess the likelihood and consequences of these risks;

✧ Risk planning

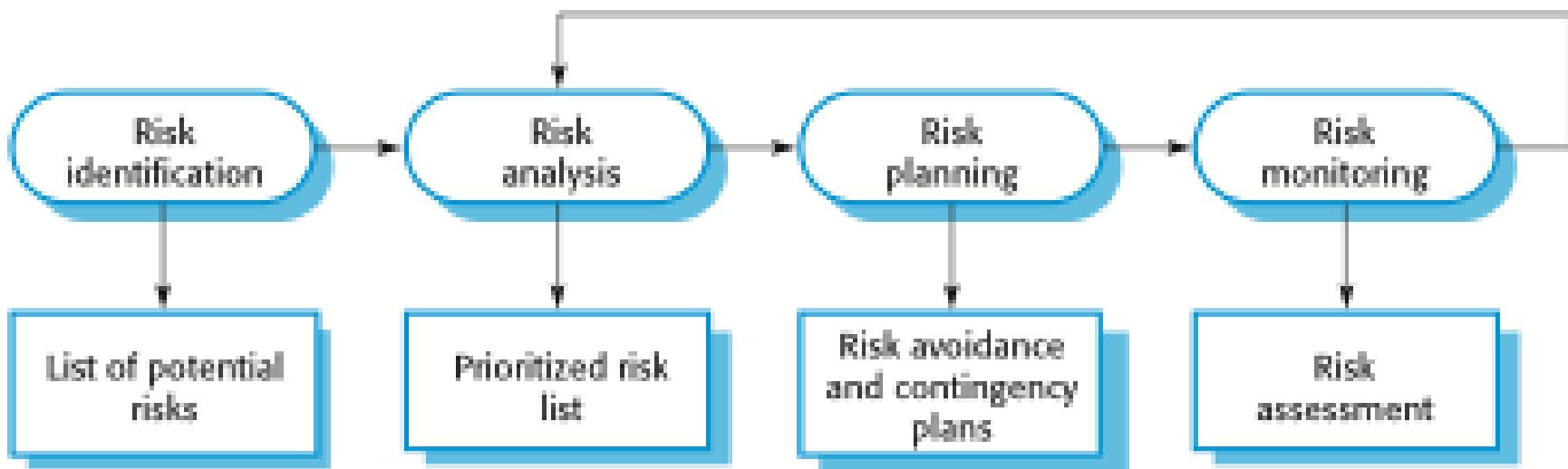
- Draw up plans to avoid or minimise the effects of the risk;

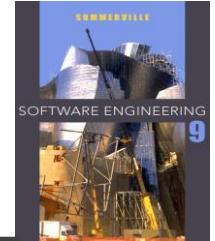
✧ Risk monitoring

- Monitor the risks throughout the project;



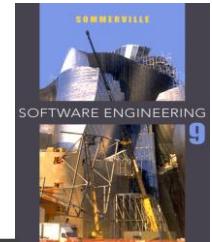
The risk management process





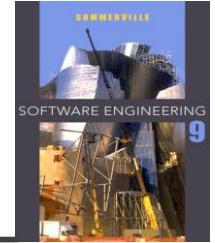
Risk identification

- ✧ May be a team activities or based on the individual project manager's experience.
- ✧ A checklist of common risks may be used to identify risks in a project
 - Technology risks.
 - People risks.
 - Organisational risks.
 - Requirements risks.
 - Estimation risks.



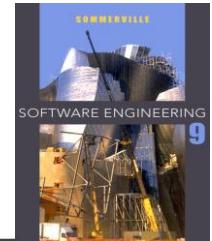
Examples of different risk types

Risk type	Possible risks
Technology	The database used in the system cannot process as many transactions per second as expected. (1) Reusable software components contain defects that mean they cannot be reused as planned. (2)
People	It is impossible to recruit staff with the skills required. (3) Key staff are ill and unavailable at critical times. (4) Required training for staff is not available. (5)
Organizational	The organization is restructured so that different management are responsible for the project. (6) Organizational financial problems force reductions in the project budget. (7)
Tools	The code generated by software code generation tools is inefficient. (8) Software tools cannot work together in an integrated way. (9)
Requirements	Changes to requirements that require major design rework are proposed. (10) Customers fail to understand the impact of requirements changes. (11)
Estimation	The time required to develop the software is underestimated. (12) The rate of defect repair is underestimated. (13) The size of the software is underestimated. (14)



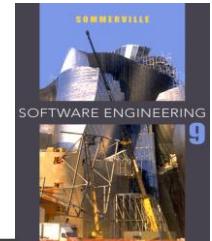
Risk analysis

- ✧ Assess probability and seriousness of each risk.
- ✧ Probability may be very low, low, moderate, high or very high.
- ✧ Risk consequences might be catastrophic, serious, tolerable or insignificant.



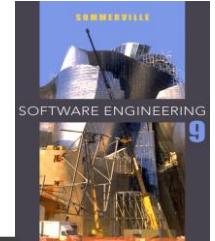
Risk types and examples

Risk	Probability	Effects
Organizational financial problems force reductions in the project budget (7).	Low	Catastrophic
It is impossible to recruit staff with the skills required for the project (3).	High	Catastrophic
Key staff are ill at critical times in the project (4).	Moderate	Serious
Faults in reusable software components have to be repaired before these components are reused. (2).	Moderate	Serious
Changes to requirements that require major design rework are proposed (10).	Moderate	Serious
The organization is restructured so that different management are responsible for the project (6).	High	Serious
The database used in the system cannot process as many transactions per second as expected (1).	Moderate	Serious



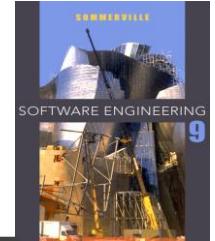
Risk types and examples

Risk	Probability	Effects
The time required to develop the software is underestimated (12).	High	Serious
Software tools cannot be integrated (9).	High	Tolerable
Customers fail to understand the impact of requirements changes (11).	Moderate	Tolerable
Required training for staff is not available (5).	Moderate	Tolerable
The rate of defect repair is underestimated (13).	Moderate	Tolerable
The size of the software is underestimated (14).	High	Tolerable
Code generated by code generation tools is inefficient (8).	Moderate	Insignificant



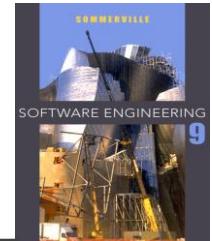
Risk planning

- ✧ Consider each risk and develop a strategy to manage that risk.
- ✧ Avoidance strategies
 - The probability that the risk will arise is reduced;
- ✧ Minimisation strategies
 - The impact of the risk on the project or product will be reduced;
- ✧ Contingency plans
 - If the risk arises, contingency plans are plans to deal with that risk;



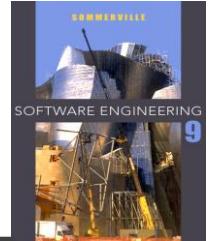
Strategies to help manage risk

Risk	Strategy
Organizational financial problems	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business and presenting reasons why cuts to the project budget would not be cost-effective.
Recruitment problems	Alert customer to potential difficulties and the possibility of delays; investigate buying-in components.
Staff illness	Reorganize team so that there is more overlap of work and people therefore understand each other's jobs.
Defective components	Replace potentially defective components with bought-in components of known reliability.
Requirements changes	Derive traceability information to assess requirements change impact; maximize information hiding in the design.



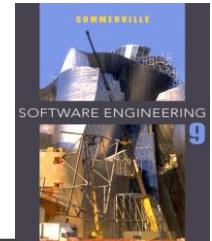
Strategies to help manage risk

Risk	Strategy
Organizational restructuring	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business.
Database performance	Investigate the possibility of buying a higher-performance database.
Underestimated development time	Investigate buying-in components; investigate use of a program generator.



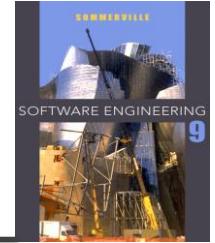
Risk monitoring

- ✧ Assess each identified risks regularly to decide whether or not it is becoming less or more probable.
- ✧ Also assess whether the effects of the risk have changed.
- ✧ Each key risk should be discussed at management progress meetings.



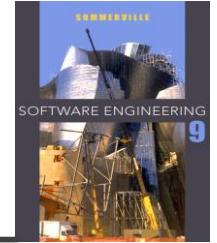
Risk indicators

Risk type	Potential indicators
Technology	Late delivery of hardware or support software; many reported technology problems.
People	Poor staff morale; poor relationships amongst team members; high staff turnover.
Organizational	Organizational gossip; lack of action by senior management.
Tools	Reluctance by team members to use tools; complaints about CASE tools; demands for higher-powered workstations.
Requirements	Many requirements change requests; customer complaints.
Estimation	Failure to meet agreed schedule; failure to clear reported defects.



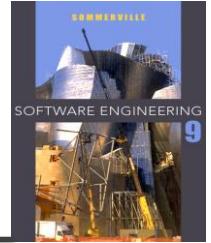
Key points

- ✧ Good project management is essential if software engineering projects are to be developed on schedule and within budget.
- ✧ Software management is distinct from other engineering management. Software is intangible. Projects may be novel or innovative with no body of experience to guide their management. Software processes are not as mature as traditional engineering processes.
- ✧ Risk management is now recognized as one of the most important project management tasks.
- ✧ Risk management involves identifying and assessing project risks to establish the probability that they will occur and the consequences for the project if that risk does arise. You should make plans to avoid, manage or deal with likely risks if or when they arise.



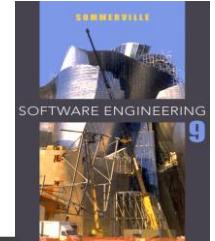
Chapter 22 – Project Management

Lecture 2



Managing people

- ✧ People are an organisation's most important assets.
- ✧ The tasks of a manager are essentially people-oriented.
Unless there is some understanding of people,
management will be unsuccessful.
- ✧ Poor people management is an important contributor to
project failure.



People management factors

✧ Consistency

- Team members should all be treated in a comparable way without favourites or discrimination.

✧ Respect

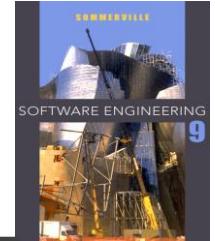
- Different team members have different skills and these differences should be respected.

✧ Inclusion

- Involve all team members and make sure that people's views are considered.

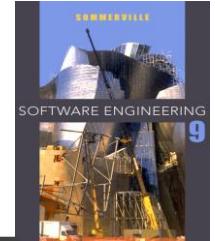
✧ Honesty

- You should always be honest about what is going well and what is going badly in a project.



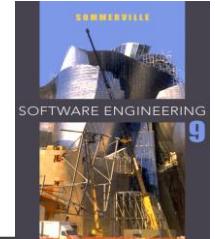
Motivating people

- ✧ An important role of a manager is to motivate the people working on a project.
- ✧ Motivation means organizing the work and the working environment to encourage people to work effectively.
 - If people are not motivated, they will not be interested in the work they are doing. They will work slowly, be more likely to make mistakes and will not contribute to the broader goals of the team or the organization.
- ✧ Motivation is a complex issue but it appears that there are different types of motivation based on:
 - Basic needs (e.g. food, sleep, etc.);
 - Personal needs (e.g. respect, self-esteem);
 - Social needs (e.g. to be accepted as part of a group).



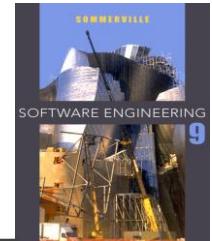
Human needs hierarchy





Need satisfaction

- ✧ In software development groups, basic physiological and safety needs are not an issue.
- ✧ Social
 - Provide communal facilities;
 - Allow informal communications e.g. via social networking
- ✧ Esteem
 - Recognition of achievements;
 - Appropriate rewards.
- ✧ Self-realization
 - Training - people want to learn more;
 - Responsibility.

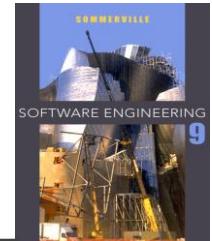


Individual motivation

Alice is a software project manager working in a company that develops alarm systems. This company wishes to enter the growing market of assistive technology to help elderly and disabled people live independently. Alice has been asked to lead a team of 6 developers than can develop new products based around the company's alarm technology.

Alice's assistive technology project starts well. Good working relationships develop within the team and creative new ideas are developed. The team decides to develop a peer-to-peer messaging system using digital televisions linked to the alarm network for communications. However, some months into the project, Alice notices that Dorothy, a hardware design expert, starts coming into work late, the quality of her work deteriorates and, increasingly, that she does not appear to be communicating with other members of the team.

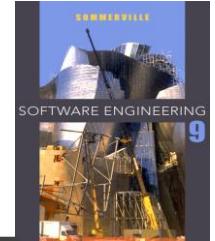
Alice talks about the problem informally with other team members to try to find out if Dorothy's personal circumstances have changed, and if this might be affecting her work. They don't know of anything, so Alice decides to talk with Dorothy to try to understand the problem.



Individual motivation

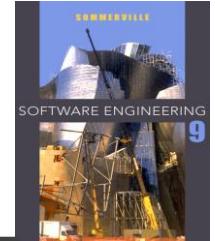
After some initial denials that there is a problem, Dorothy admits that she has lost interest in the job. She expected that she would be able to develop and use her hardware interfacing skills. However, because of the product direction that has been chosen, she has little opportunity for this. Basically, she is working as a C programmer with other team members.

Although she admits that the work is challenging, she is concerned that she is not developing her interfacing skills. She is worried that finding a job that involves hardware interfacing will be difficult after this project. Because she does not want to upset the team by revealing that she is thinking about the next project, she has decided that it is best to minimize conversation with them.



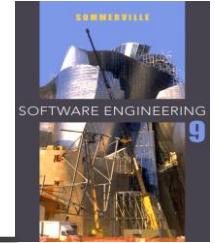
Personality types

- ✧ The needs hierarchy is almost certainly an over-simplification of motivation in practice.
- ✧ Motivation should also take into account different personality types:
 - Task-oriented;
 - Self-oriented;
 - Interaction-oriented.



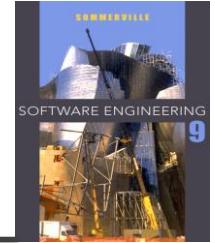
Personality types

- ✧ Task-oriented.
 - The motivation for doing the work is the work itself;
- ✧ Self-oriented.
 - The work is a means to an end which is the achievement of individual goals - e.g. to get rich, to play tennis, to travel etc.;
- ✧ Interaction-oriented
 - The principal motivation is the presence and actions of co-workers. People go to work because they like to go to work.



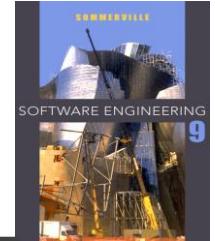
Motivation balance

- ✧ Individual motivations are made up of elements of each class.
- ✧ The balance can change depending on personal circumstances and external events.
- ✧ However, people are not just motivated by personal factors but also by being part of a group and culture.
- ✧ People go to work because they are motivated by the people that they work with.



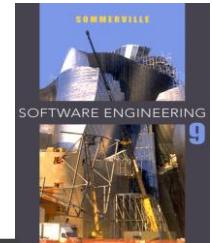
Teamwork

- ✧ Most software engineering is a group activity
 - The development schedule for most non-trivial software projects is such that they cannot be completed by one person working alone.
- ✧ A good group is cohesive and has a team spirit. The people involved are motivated by the success of the group as well as by their own personal goals.
- ✧ Group interaction is a key determinant of group performance.
- ✧ Flexibility in group composition is limited
 - Managers must do the best they can with available people.



Group cohesiveness

- ✧ In a cohesive group, members consider the group to be more important than any individual in it.
- ✧ The advantages of a cohesive group are:
 - Group quality standards can be developed by the group members.
 - Team members learn from each other and get to know each other's work; Inhibitions caused by ignorance are reduced.
 - Knowledge is shared. Continuity can be maintained if a group member leaves.
 - Refactoring and continual improvement is encouraged. Group members work collectively to deliver high quality results and fix problems, irrespective of the individuals who originally created the design or program.

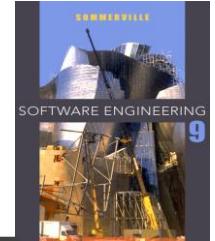


Team spirit

Alice, an experienced project manager, understands the importance of creating a cohesive group. As they are developing a new product, she takes the opportunity of involving all group members in the product specification and design by getting them to discuss possible technology with elderly members of their families. She also encourages them to bring these family members to meet other members of the development group.

Alice also arranges monthly lunches for everyone in the group. These lunches are an opportunity for all team members to meet informally, talk around issues of concern, and get to know each other. At the lunch, Alice tells the group what she knows about organizational news, policies, strategies, and so forth. Each team member then briefly summarizes what they have been doing and the group discusses a general topic, such as new product ideas from elderly relatives.

Every few months, Alice organizes an ‘away day’ for the group where the team spends two days on ‘technology updating’. Each team member prepares an update on a relevant technology and presents it to the group. This is an off-site meeting in a good hotel and plenty of time is scheduled for discussion and social interaction.



The effectiveness of a team

✧ The people in the group

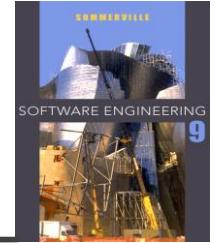
- You need a mix of people in a project group as software development involves diverse activities such as negotiating with clients, programming, testing and documentation.

✧ The group organization

- A group should be organized so that individuals can contribute to the best of their abilities and tasks can be completed as expected.

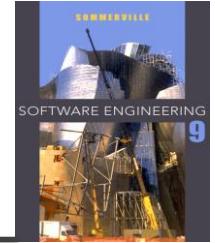
✧ Technical and managerial communications

- Good communications between group members, and between the software engineering team and other project stakeholders, is essential.



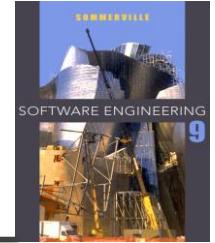
Selecting group members

- ✧ A manager or team leader's job is to create a cohesive group and organize their group so that they can work together effectively.
- ✧ This involves creating a group with the right balance of technical skills and personalities, and organizing that group so that the members work together effectively.



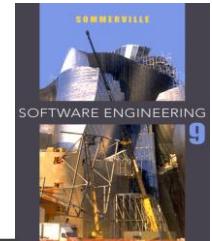
Assembling a team

- ✧ May not be possible to appoint the ideal people to work on a project
 - Project budget may not allow for the use of highly-paid staff;
 - Staff with the appropriate experience may not be available;
 - An organisation may wish to develop employee skills on a software project.
- ✧ Managers have to work within these constraints especially when there are shortages of trained staff.



Group composition

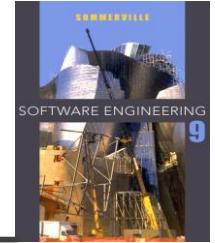
- ✧ Group composed of members who share the same motivation can be problematic
 - Task-oriented - everyone wants to do their own thing;
 - Self-oriented - everyone wants to be the boss;
 - Interaction-oriented - too much chatting, not enough work.
- ✧ An effective group has a balance of all types.
- ✧ This can be difficult to achieve software engineers are often task-oriented.
- ✧ Interaction-oriented people are very important as they can detect and defuse tensions that arise.



Group composition

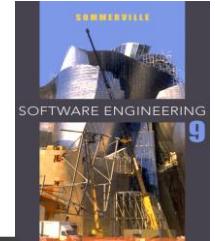
In creating a group for assistive technology development, Alice is aware of the importance of selecting members with complementary personalities. When interviewing potential group members, she tried to assess whether they were task-oriented, self-oriented, or interaction-oriented. She felt that she was primarily a self-oriented type because she considered the project to be a way of getting noticed by senior management and possibly promoted. She therefore looked for one or perhaps two interaction-oriented personalities, with task-oriented individuals to complete the team. The final assessment that she arrived at was:

- Alice—self-oriented
- Brian—task-oriented
- Bob—task-oriented
- Carol—interaction-oriented
- Dorothy—self-oriented
- Ed—interaction-oriented
- Fred—task-oriented



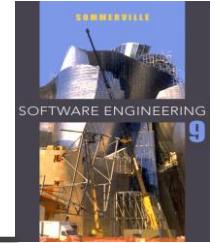
Group organization

- ✧ The way that a group is organized affects the decisions that are made by that group, the ways that information is exchanged and the interactions between the development group and external project stakeholders.
 - Key questions include:
 - Should the project manager be the technical leader of the group?
 - Who will be involved in making critical technical decisions, and how will these be made?
 - How will interactions with external stakeholders and senior company management be handled?
 - How can groups integrate people who are not co-located?
 - How can knowledge be shared across the group?



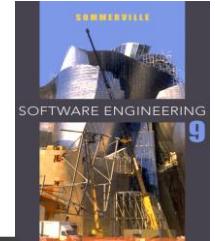
Group organization

- ✧ Small software engineering groups are usually organised informally without a rigid structure.
- ✧ For large projects, there may be a hierarchical structure where different groups are responsible for different sub-projects.
- ✧ Agile development is always based around an informal group on the principle that formal structure inhibits information exchange



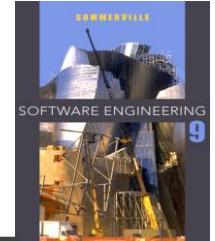
Informal groups

- ✧ The group acts as a whole and comes to a consensus on decisions affecting the system.
- ✧ The group leader serves as the external interface of the group but does not allocate specific work items.
- ✧ Rather, work is discussed by the group as a whole and tasks are allocated according to ability and experience.
- ✧ This approach is successful for groups where all members are experienced and competent.



Group communications

- ✧ Good communications are essential for effective group working.
- ✧ Information must be exchanged on the status of work, design decisions and changes to previous decisions.
- ✧ Good communications also strengthens group cohesion as it promotes understanding.



Group communications

✧ Group size

- The larger the group, the harder it is for people to communicate with other group members.

✧ Group structure

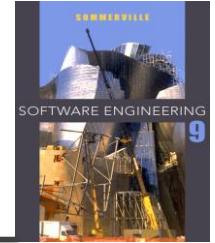
- Communication is better in informally structured groups than in hierarchically structured groups.

✧ Group composition

- Communication is better when there are different personality types in a group and when groups are mixed rather than single sex.

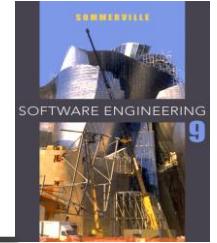
✧ The physical work environment

- Good workplace organisation can help encourage communications.



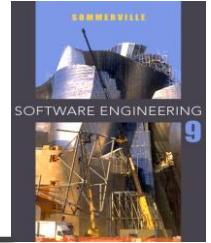
Key points

- ✧ People are motivated by interaction with other people, the recognition of management and their peers, and by being given opportunities for personal development.
- ✧ Software development groups should be fairly small and cohesive. The key factors that influence the effectiveness of a group are the people in that group, the way that it is organized and the communication between group members.
- ✧ Communications within a group are influenced by factors such as the status of group members, the size of the group, the gender composition of the group, personalities and available communication channels.



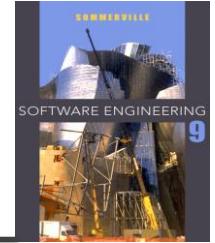
Chapter 23 – Project planning

Lecture 1



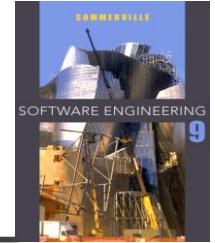
Topics covered

- ✧ Software pricing
- ✧ Plan-driven development
- ✧ Project scheduling
- ✧ Agile planning
- ✧ Estimation techniques



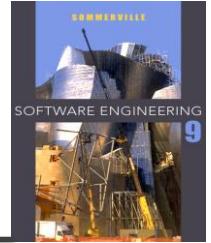
Project planning

- ✧ Project planning involves breaking down the work into parts and assign these to project team members, anticipate problems that might arise and prepare tentative solutions to those problems.
- ✧ The project plan, which is created at the start of a project, is used to communicate how the work will be done to the project team and customers, and to help assess progress on the project.



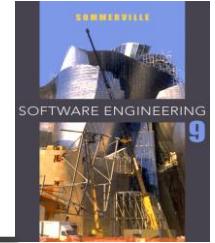
Planning stages

- ✧ At the proposal stage, when you are bidding for a contract to develop or provide a software system.
- ✧ During the project startup phase, when you have to plan who will work on the project, how the project will be broken down into increments, how resources will be allocated across your company, etc.
- ✧ Periodically throughout the project, when you modify your plan in the light of experience gained and information from monitoring the progress of the work.



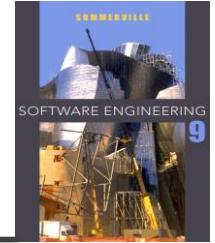
Proposal planning

- ✧ Planning may be necessary with only outline software requirements.
- ✧ The aim of planning at this stage is to provide information that will be used in setting a price for the system to customers.



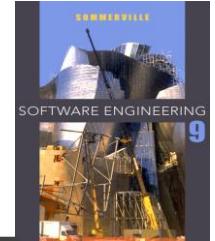
Software pricing

- ✧ Estimates are made to discover the cost, to the developer, of producing a software system.
 - You take into account, hardware, software, travel, training and effort costs.
- ✧ There is not a simple relationship between the development cost and the price charged to the customer.
- ✧ Broader organisational, economic, political and business considerations influence the price charged.



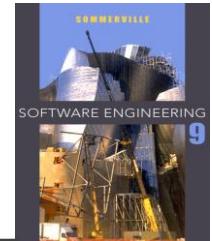
Factors affecting software pricing

Factor	Description
Market opportunity	A development organization may quote a low price because it wishes to move into a new segment of the software market. Accepting a low profit on one project may give the organization the opportunity to make a greater profit later. The experience gained may also help it develop new products.
Cost estimate uncertainty	If an organization is unsure of its cost estimate, it may increase its price by a contingency over and above its normal profit.
Contractual terms	A customer may be willing to allow the developer to retain ownership of the source code and reuse it in other projects. The price charged may then be less than if the software source code is handed over to the customer.



Factors affecting software pricing

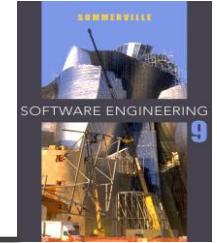
Factor	Description
Requirements volatility	If the requirements are likely to change, an organization may lower its price to win a contract. After the contract is awarded, high prices can be charged for changes to the requirements.
Financial health	Developers in financial difficulty may lower their price to gain a contract. It is better to make a smaller than normal profit or break even than to go out of business. Cash flow is more important than profit in difficult economic times.



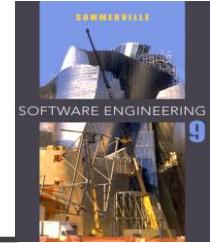
Plan-driven development

- ✧ Plan-driven or plan-based development is an approach to software engineering where the development process is planned in detail.
 - Plan-driven development is based on engineering project management techniques and is the ‘traditional’ way of managing large software development projects.
- ✧ A project plan is created that records the work to be done, who will do it, the development schedule and the work products.
- ✧ Managers use the plan to support project decision making and as a way of measuring progress.

Plan-driven development – pros and cons

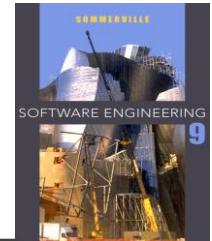


- ✧ The arguments in favor of a plan-driven approach are that early planning allows organizational issues (availability of staff, other projects, etc.) to be closely taken into account, and that potential problems and dependencies are discovered before the project starts, rather than once the project is underway.
- ✧ The principal argument against plan-driven development is that many early decisions have to be revised because of changes to the environment in which the software is to be developed and used.



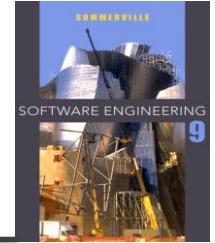
Project plans

- ✧ In a plan-driven development project, a project plan sets out the resources available to the project, the work breakdown and a schedule for carrying out the work.
- ✧ Plan sections
 - Introduction
 - Project organization
 - Risk analysis
 - Hardware and software resource requirements
 - Work breakdown
 - Project schedule
 - Monitoring and reporting mechanisms



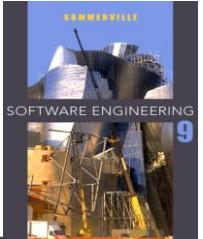
Project plan supplements

Plan	Description
Quality plan	Describes the quality procedures and standards that will be used in a project.
Validation plan	Describes the approach, resources, and schedule used for system validation.
Configuration management plan	Describes the configuration management procedures and structures to be used.
Maintenance plan	Predicts the maintenance requirements, costs, and effort.
Staff development plan	Describes how the skills and experience of the project team members will be developed.

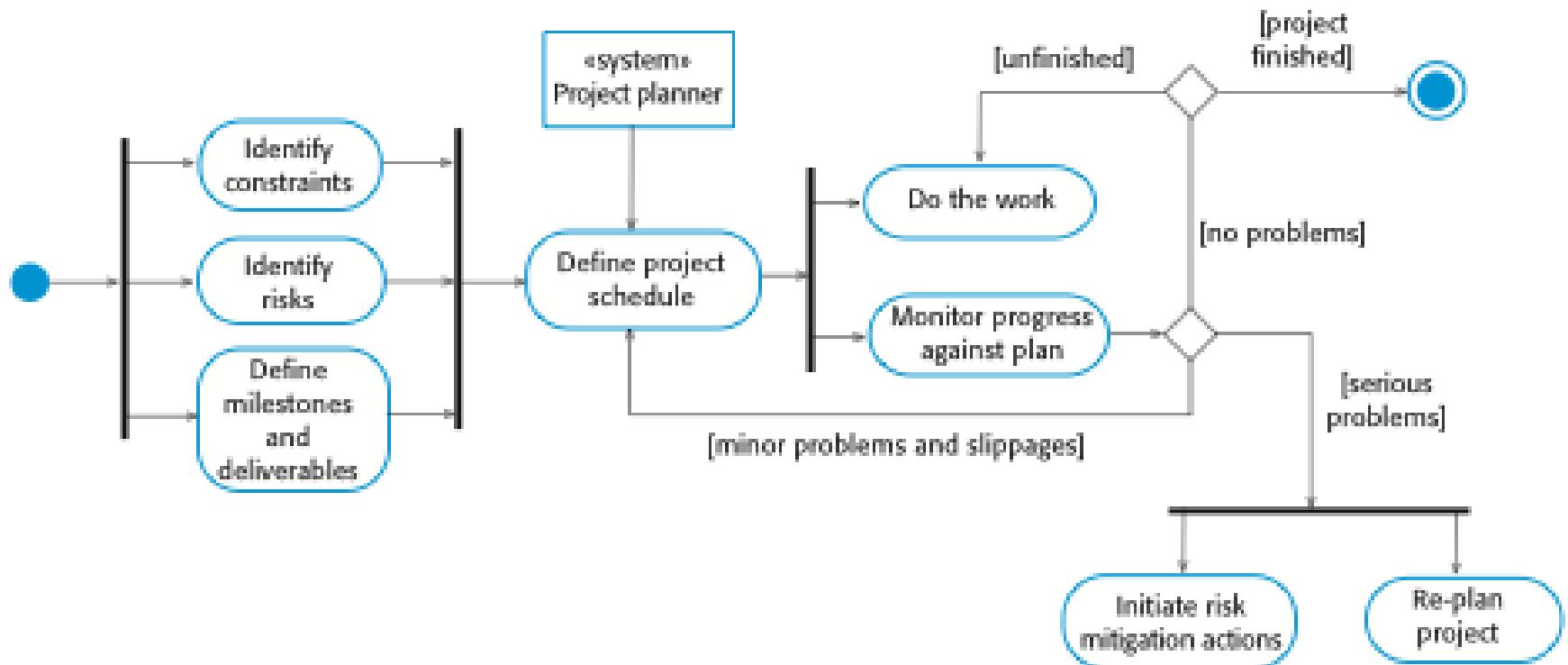


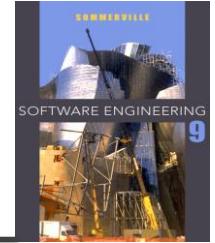
The planning process

- ✧ Project planning is an iterative process that starts when you create an initial project plan during the project startup phase.
- ✧ Plan changes are inevitable.
 - As more information about the system and the project team becomes available during the project, you should regularly revise the plan to reflect requirements, schedule and risk changes.
 - Changing business goals also leads to changes in project plans. As business goals change, this could affect all projects, which may then have to be re-planned.



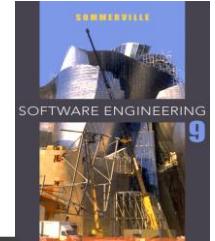
The project planning process





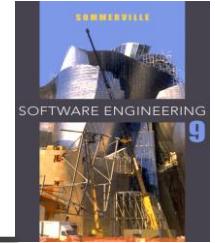
Project scheduling

- ✧ Project scheduling is the process of deciding how the work in a project will be organized as separate tasks, and when and how these tasks will be executed.
- ✧ You estimate the calendar time needed to complete each task, the effort required and who will work on the tasks that have been identified.
- ✧ You also have to estimate the resources needed to complete each task, such as the disk space required on a server, the time required on specialized hardware, such as a simulator, and what the travel budget will be.



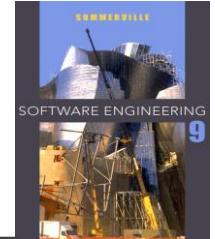
Project scheduling activities

- ✧ Split project into tasks and estimate time and resources required to complete each task.
- ✧ Organize tasks concurrently to make optimal use of workforce.
- ✧ Minimize task dependencies to avoid delays caused by one task waiting for another to complete.
- ✧ Dependent on project managers intuition and experience.

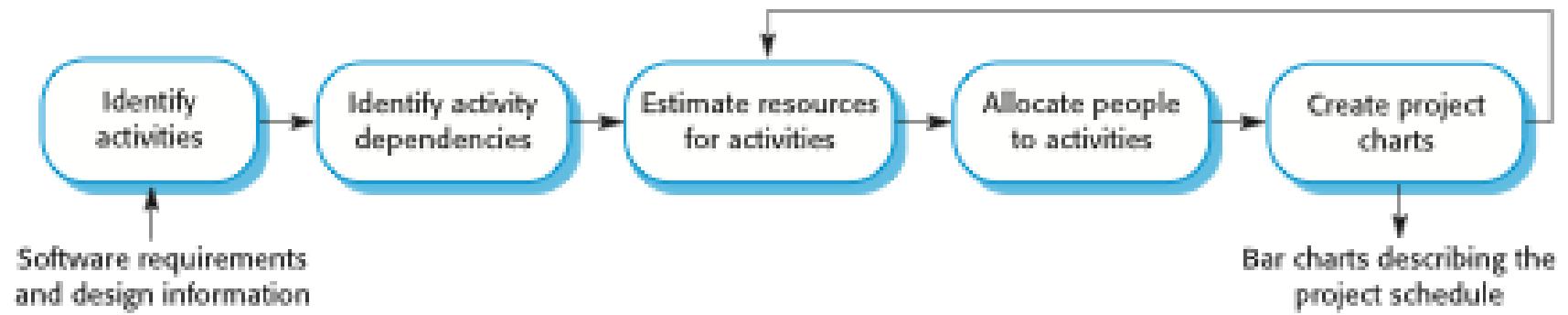


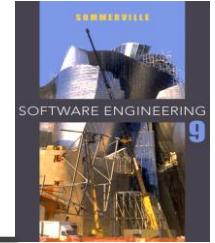
Milestones and deliverables

- ✧ Milestones are points in the schedule against which you can assess progress, for example, the handover of the system for testing.
- ✧ Deliverables are work products that are delivered to the customer, e.g. a requirements document for the system.



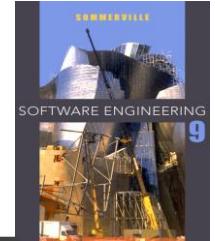
The project scheduling process





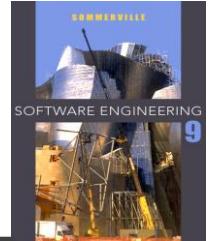
Scheduling problems

- ✧ Estimating the difficulty of problems and hence the cost of developing a solution is hard.
- ✧ Productivity is not proportional to the number of people working on a task.
- ✧ Adding people to a late project makes it later because of communication overheads.
- ✧ The unexpected always happens. Always allow contingency in planning.



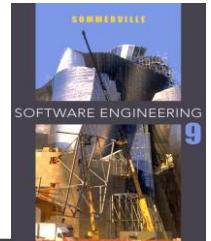
Schedule representation

- ✧ Graphical notations are normally used to illustrate the project schedule.
- ✧ These show the project breakdown into tasks. Tasks should not be too small. They should take about a week or two.
- ✧ Bar charts are the most commonly used representation for project schedules. They show the schedule as activities or resources against time.

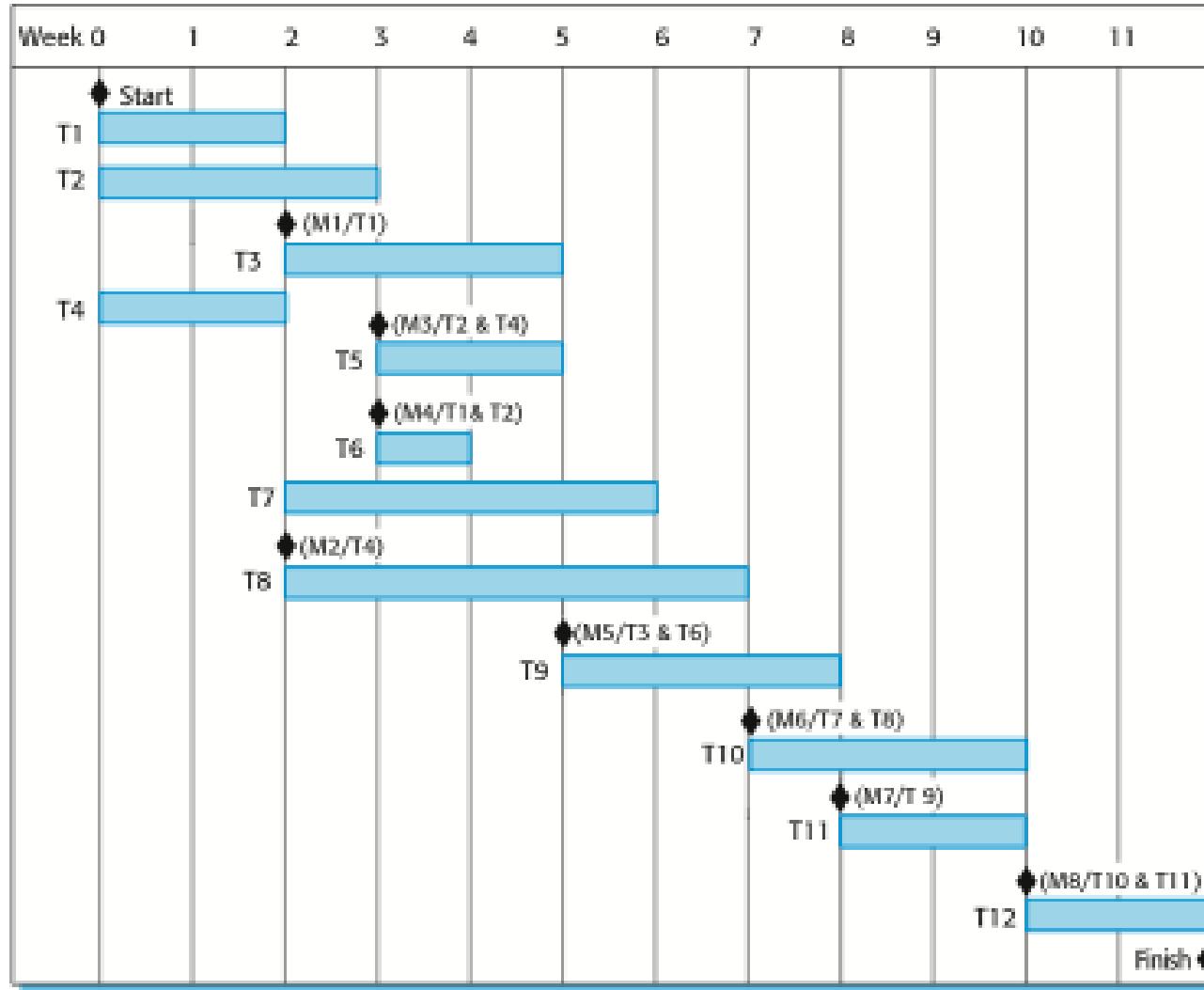


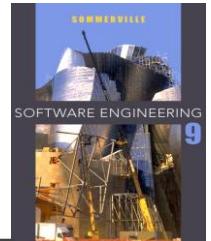
Tasks, durations, and dependencies

Task	Effort (person-days)	Duration (days)	Dependencies
T1	15	10	
T2	8	15	
T3	20	15	T1 (M1)
T4	5	10	
T5	5	10	T2, T4 (M3)
T6	10	5	T1, T2 (M4)
T7	25	20	T1 (M1)
T8	75	25	T4 (M2)
T9	10	15	T3, T6 (M5)
T10	20	15	T7, T8 (M6)
T11	10	10	T9 (M7)
T12	20	10	T10, T11 (M8)

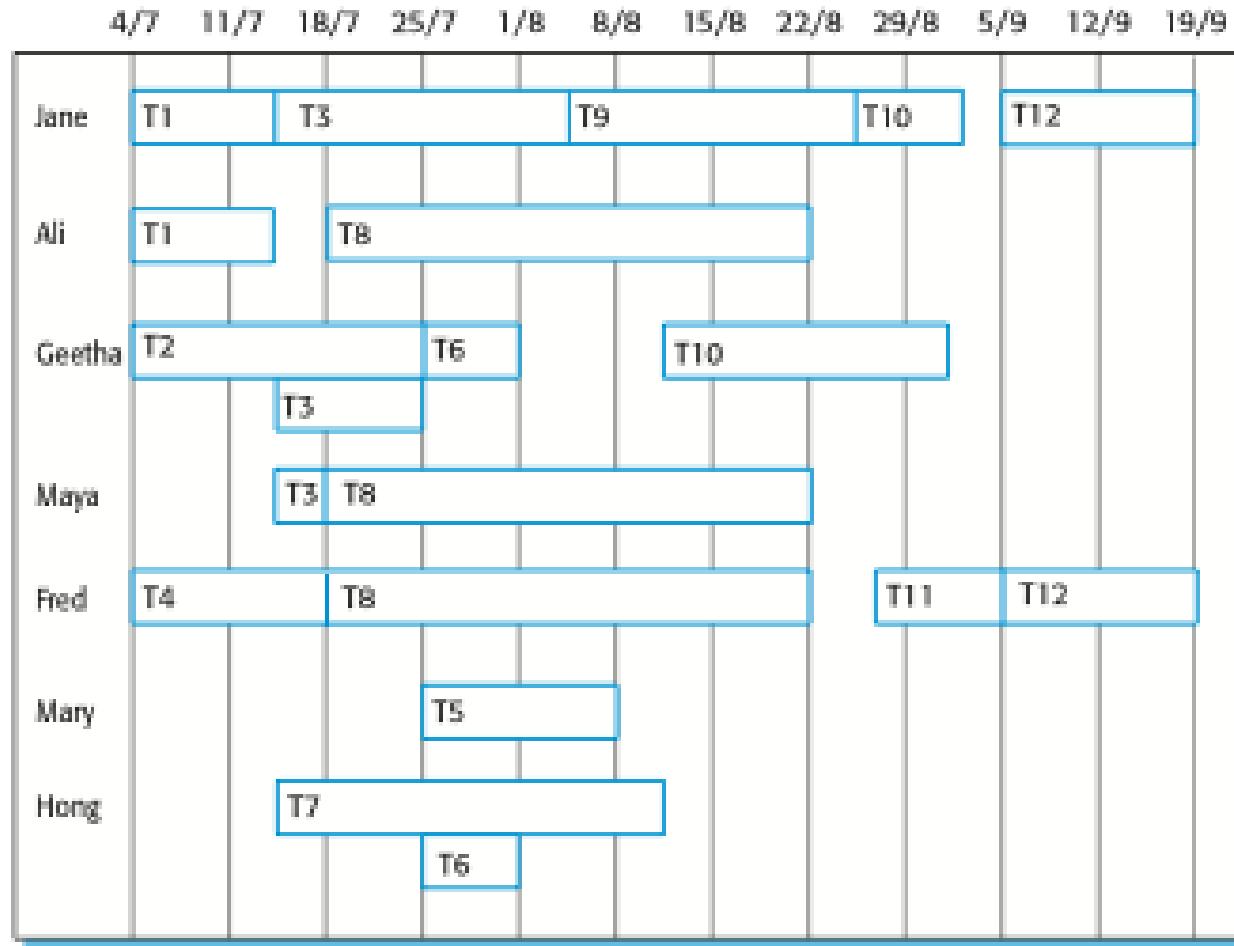


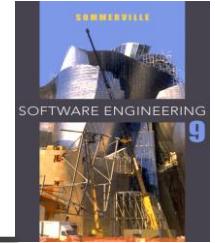
Activity bar chart





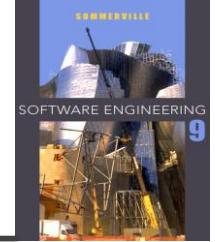
Staff allocation chart





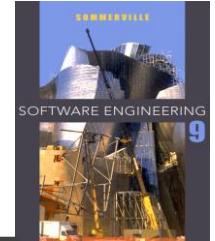
Agile planning

- ✧ Agile methods of software development are iterative approaches where the software is developed and delivered to customers in increments.
- ✧ Unlike plan-driven approaches, the functionality of these increments is not planned in advance but is decided during the development.
 - The decision on what to include in an increment depends on progress and on the customer's priorities.
- ✧ The customer's priorities and requirements change so it makes sense to have a flexible plan that can accommodate these changes.



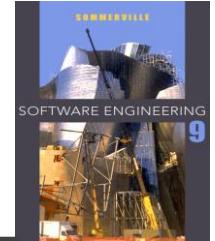
Agile planning stages

- ✧ Release planning, which looks ahead for several months and decides on the features that should be included in a release of a system.
- ✧ Iteration planning, which has a shorter term outlook, and focuses on planning the next increment of a system. This is typically 2-4 weeks of work for the team.



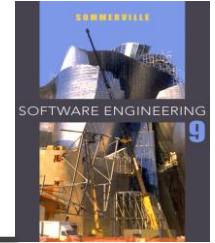
Planning in XP





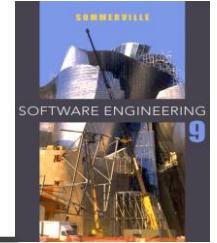
Story-based planning

- ✧ The system specification in XP is based on user stories that reflect the features that should be included in the system.
- ✧ The project team read and discuss the stories and rank them in order of the amount of time they think it will take to implement the story.
- ✧ Release planning involves selecting and refining the stories that will reflect the features to be implemented in a release of a system and the order in which the stories should be implemented.
- ✧ Stories to be implemented in each iteration are chosen, with the number of stories reflecting the time to deliver an iteration (usually 2 or 3 weeks).



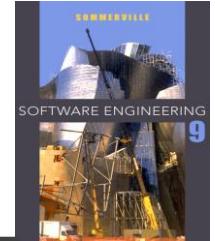
Key points

- ✧ The price charged for a system does not just depend on its estimated development costs; it may be adjusted depending on the market and organizational priorities.
- ✧ Plan-driven development is organized around a complete project plan that defines the project activities, the planned effort, the activity schedule and who is responsible for each activity.
- ✧ Project scheduling involves the creation of graphical representations of the project plan. Bar charts, which show the activity duration and staffing timelines, are the most commonly used schedule representations.
- ✧ The XP planning game involves the whole team in project planning. The plan is developed incrementally and, if problems arise, is adjusted. Software functionality is reduced instead of delaying delivery of an increment.



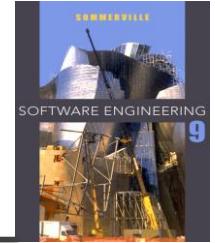
Chapter 23 – Project planning

Lecture 2



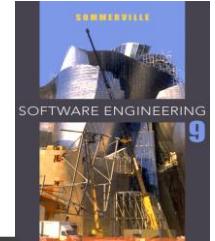
Estimation techniques

- ✧ Organizations need to make software effort and cost estimates. There are two types of technique that can be used to do this:
 - *Experience-based techniques* The estimate of future effort requirements is based on the manager's experience of past projects and the application domain. Essentially, the manager makes an informed judgment of what the effort requirements are likely to be.
 - *Algorithmic cost modeling* In this approach, a formulaic approach is used to compute the project effort based on estimates of product attributes, such as size, and process characteristics, such as experience of staff involved.



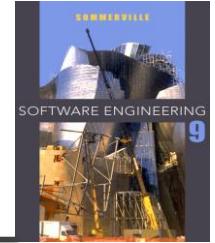
Experience-based approaches

- ✧ Experience-based techniques rely on judgments based on experience of past projects and the effort expended in these projects on software development activities.
- ✧ Typically, you identify the deliverables to be produced in a project and the different software components or systems that are to be developed.
- ✧ You document these in a spreadsheet, estimate them individually and compute the total effort required.
- ✧ It usually helps to get a group of people involved in the effort estimation and to ask each member of the group to explain their estimate.



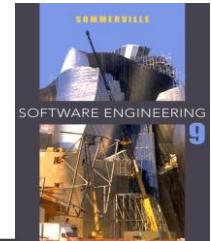
Algorithmic cost modelling

- ✧ Cost is estimated as a mathematical function of product, project and process attributes whose values are estimated by project managers:
 - $\text{Effort} = A \cdot \text{Size}^B \cdot M$
 - A is an organisation-dependent constant, B reflects the disproportionate effort for large projects and M is a multiplier reflecting product, process and people attributes.
- ✧ The most commonly used product attribute for cost estimation is code size.
- ✧ Most models are similar but they use different values for A, B and M.

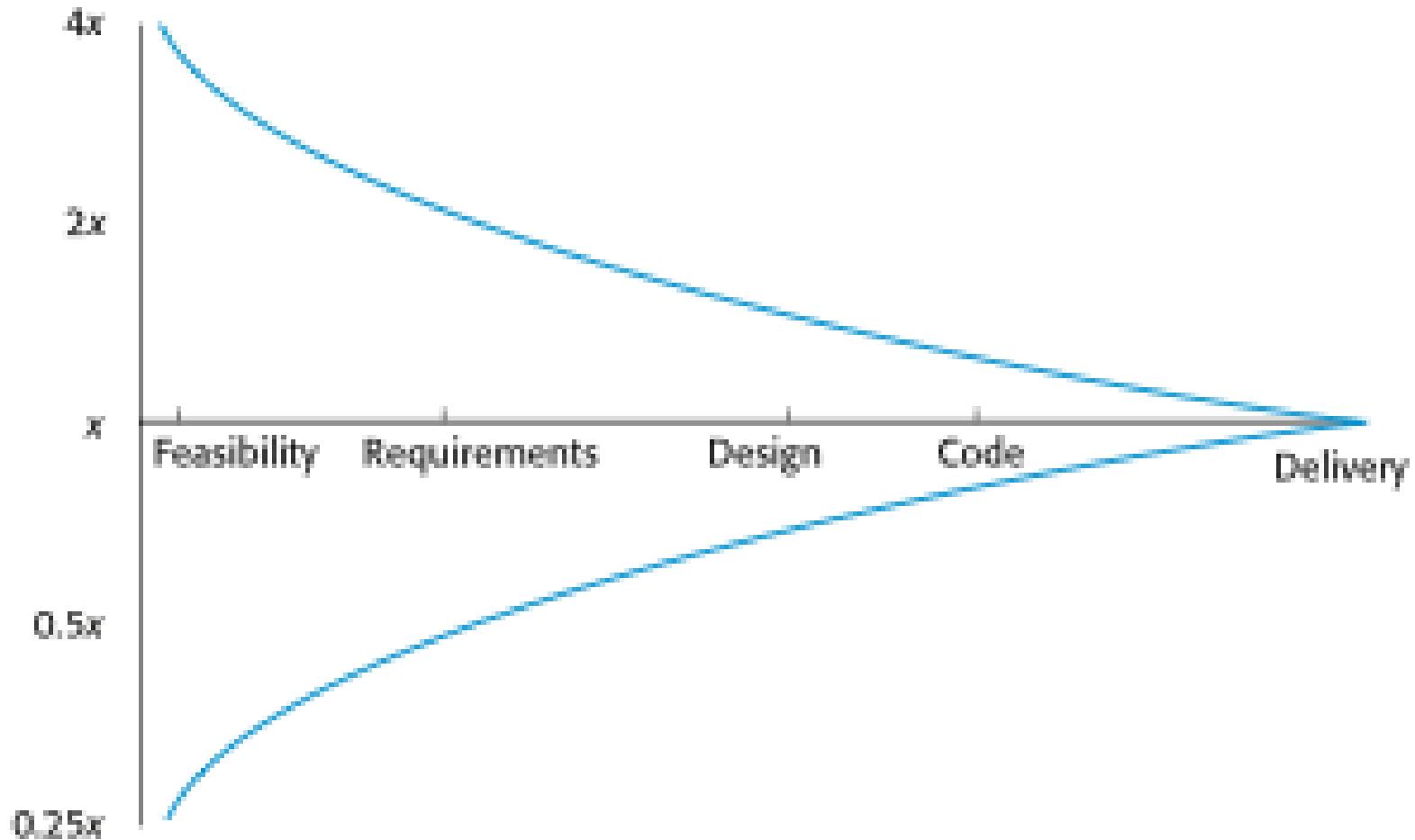


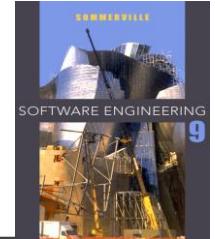
Estimation accuracy

- ✧ The size of a software system can only be known accurately when it is finished.
- ✧ Several factors influence the final size
 - Use of COTS and components;
 - Programming language;
 - Distribution of system.
- ✧ As the development process progresses then the size estimate becomes more accurate.
- ✧ The estimates of the factors contributing to B and M are subjective and vary according to the judgment of the estimator.



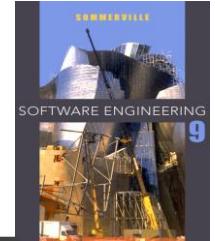
Estimate uncertainty





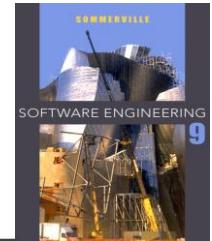
The COCOMO 2 model

- ✧ An empirical model based on project experience.
- ✧ Well-documented, ‘independent’ model which is not tied to a specific software vendor.
- ✧ Long history from initial version published in 1981 (COCOMO-81) through various instantiations to COCOMO 2.
- ✧ COCOMO 2 takes into account different approaches to software development, reuse, etc.

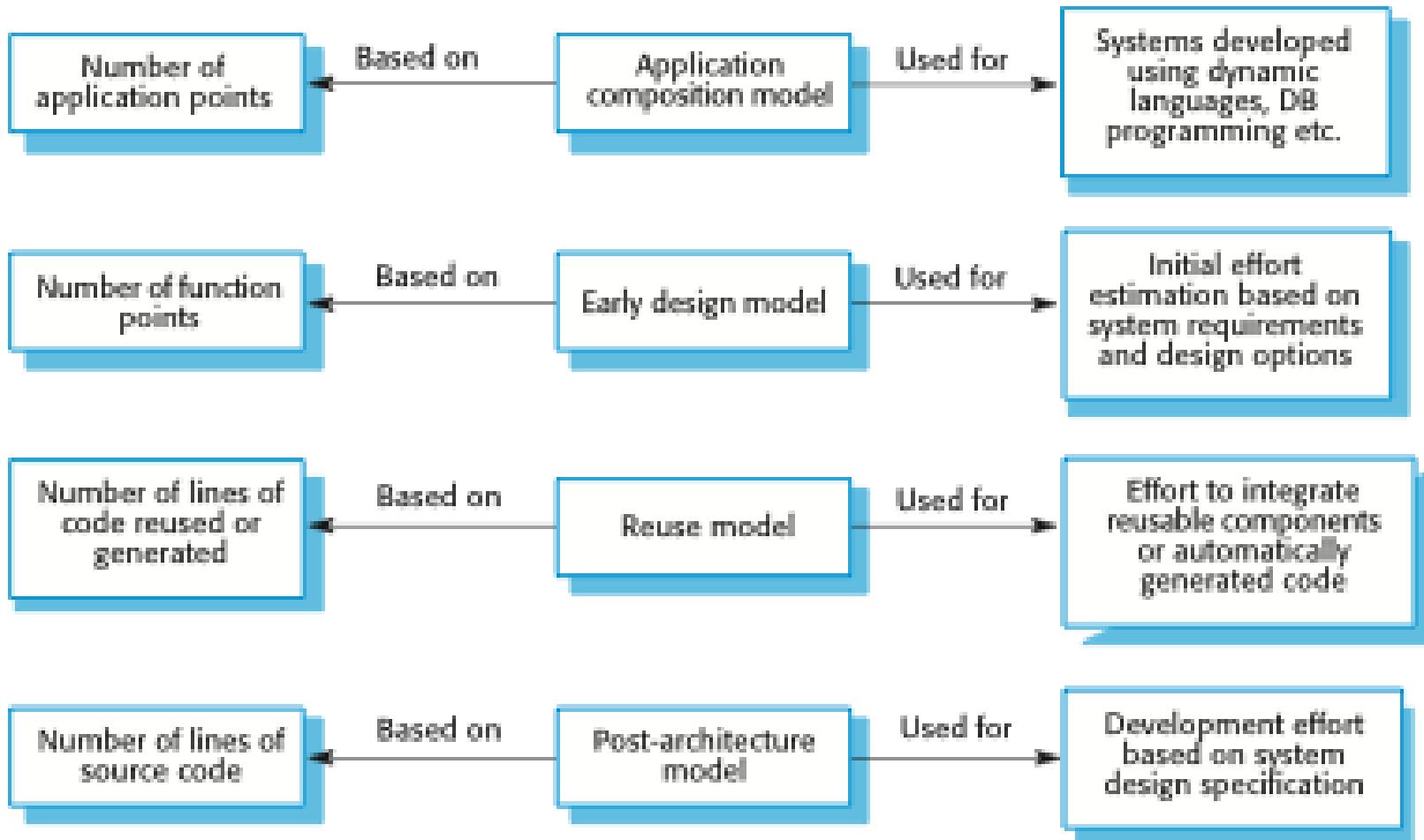


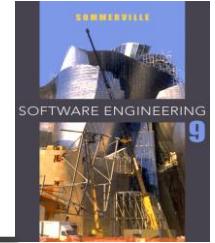
COCOMO 2 models

- ✧ COCOMO 2 incorporates a range of sub-models that produce increasingly detailed software estimates.
- ✧ The sub-models in COCOMO 2 are:
 - **Application composition model.** Used when software is composed from existing parts.
 - **Early design model.** Used when requirements are available but design has not yet started.
 - **Reuse model.** Used to compute the effort of integrating reusable components.
 - **Post-architecture model.** Used once the system architecture has been designed and more information about the system is available.



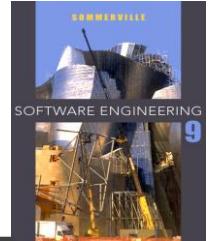
COCOMO estimation models





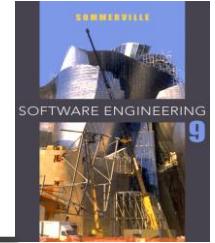
Application composition model

- ✧ Supports prototyping projects and projects where there is extensive reuse.
- ✧ Based on standard estimates of developer productivity in application (object) points/month.
- ✧ Takes CASE tool use into account.
- ✧ Formula is
 - $PM = (NAP \cdot (1 - \%reuse/100)) / PROD$
 - PM is the effort in person-months, NAP is the number of application points and PROD is the productivity.



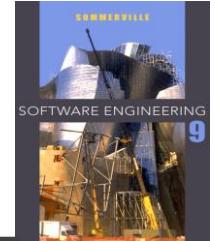
Application-point productivity

Developer's experience and capability	Very low	Low	Nominal	High	Very high
ICASE maturity and capability	Very low	Low	Nominal	High	Very high
PROD (NAP/month)	4	7	13	25	50



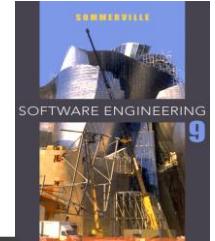
Early design model

- ✧ Estimates can be made after the requirements have been agreed.
- ✧ Based on a standard formula for algorithmic models
 - $PM = A \cdot Size^B \cdot M$ where
 - $M = PERS \cdot RCPX \cdot RUSE \cdot PDIF \cdot PREX \cdot FCIL \cdot SCED;$
 - $A = 2.94$ in initial calibration, Size in KLOC, B varies from 1.1 to 1.24 depending on novelty of the project, development flexibility, risk management approaches and the process maturity.



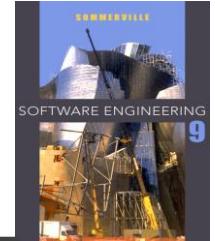
Multipliers

- ✧ Multipliers reflect the capability of the developers, the non-functional requirements, the familiarity with the development platform, etc.
 - RCPX - product reliability and complexity;
 - RUSE - the reuse required;
 - PDIF - platform difficulty;
 - PREX - personnel experience;
 - PERS - personnel capability;
 - SCED - required schedule;
 - FCIL - the team support facilities.



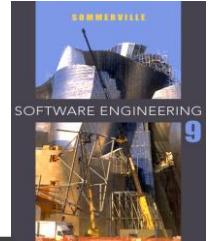
The reuse model

- ✧ Takes into account black-box code that is reused without change and code that has to be adapted to integrate it with new code.
- ✧ There are two versions:
 - Black-box reuse where code is not modified. An effort estimate (PM) is computed.
 - White-box reuse where code is modified. A size estimate equivalent to the number of lines of new source code is computed. This then adjusts the size estimate for new code.



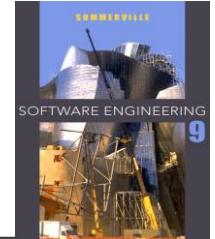
Reuse model estimates 1

- ✧ For generated code:
 - $PM = (ASLOC * AT/100)/ATPROD$
 - ASLOC is the number of lines of generated code
 - AT is the percentage of code automatically generated.
 - ATPROD is the productivity of engineers in integrating this code.



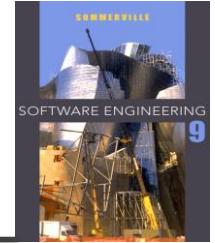
Reuse model estimates 2

- ✧ When code has to be understood and integrated:
 - $\text{ESLOC} = \text{ASLOC} * (1-\text{AT}/100) * \text{AAM}$.
 - ASLOC and AT as before.
 - AAM is the adaptation adjustment multiplier computed from the costs of changing the reused code, the costs of understanding how to integrate the code and the costs of reuse decision making.



Post-architecture level

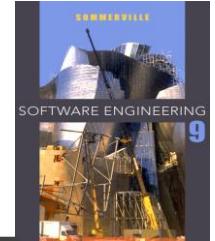
- ✧ Uses the same formula as the early design model but with 17 rather than 7 associated multipliers.
- ✧ The code size is estimated as:
 - Number of lines of new code to be developed;
 - Estimate of equivalent number of lines of new code computed using the reuse model;
 - An estimate of the number of lines of code that have to be modified according to requirements changes.



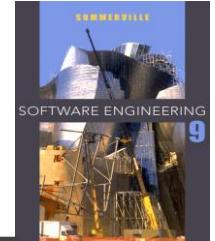
The exponent term

- ✧ This depends on 5 scale factors (see next slide). Their sum/100 is added to 1.01
- ✧ A company takes on a project in a new domain. The client has not defined the process to be used and has not allowed time for risk analysis. The company has a CMM level 2 rating.
 - Precedenteness - new project (4)
 - Development flexibility - no client involvement - Very high (1)
 - Architecture/risk resolution - No risk analysis - V. Low .(5)
 - Team cohesion - new team - nominal (3)
 - Process maturity - some control - nominal (3)
- ✧ Scale factor is therefore 1.17.

Scale factors used in the exponent computation in the post-architecture model



Scale factor	Explanation
Precedentedness	Reflects the previous experience of the organization with this type of project. Very low means no previous experience; extra-high means that the organization is completely familiar with this application domain.
Development flexibility	Reflects the degree of flexibility in the development process. Very low means a prescribed process is used; extra-high means that the client sets only general goals.
Architecture/risk resolution	Reflects the extent of risk analysis carried out. Very low means little analysis; extra-high means a complete and thorough risk analysis.
Team cohesion	Reflects how well the development team knows each other and work together. Very low means very difficult interactions; extra-high means an integrated and effective team with no communication problems.
Process maturity	Reflects the process maturity of the organization. The computation of this value depends on the CMM Maturity Questionnaire, but an estimate can be achieved by subtracting the CMM process maturity level from 5.



Multipliers

✧ Product attributes

- Concerned with required characteristics of the software product being developed.

✧ Computer attributes

- Constraints imposed on the software by the hardware platform.

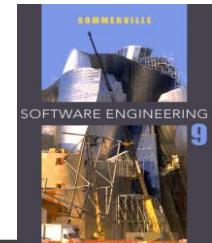
✧ Personnel attributes

- Multipliers that take the experience and capabilities of the people working on the project into account.

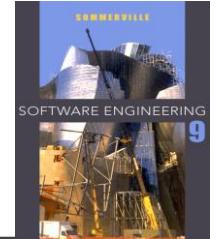
✧ Project attributes

- Concerned with the particular characteristics of the software development project.

The effect of cost drivers on effort estimates

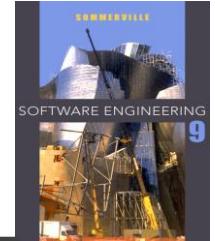


Exponent value	1.17
System size (including factors for reuse and requirements volatility)	128,000 DSI
Initial COCOMO estimate without cost drivers	730 person-months
Reliability	Very high, multiplier = 1.39
Complexity	Very high, multiplier = 1.3
Memory constraint	High, multiplier = 1.21
Tool use	Low, multiplier = 1.12
Schedule	Accelerated, multiplier = 1.29
Adjusted COCOMO estimate	2,306 person-months



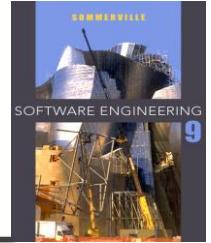
The effect of cost drivers on effort estimates

Exponent value	1.17
Reliability	Very low, multiplier = 0.75
Complexity	Very low, multiplier = 0.75
Memory constraint	None, multiplier = 1
Tool use	Very high, multiplier = 0.72
Schedule	Normal, multiplier = 1
Adjusted COCOMO estimate	295 person-months



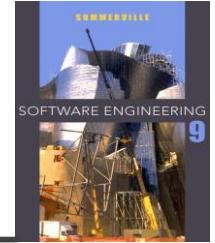
Project duration and staffing

- ✧ As well as effort estimation, managers must estimate the calendar time required to complete a project and when staff will be required.
- ✧ Calendar time can be estimated using a COCOMO 2 formula
 - $TDEV = 3^{\wedge} (PM)^{(0.33+0.2*(B-1.01))}$
 - PM is the effort computation and B is the exponent computed as discussed above (B is 1 for the early prototyping model). This computation predicts the nominal schedule for the project.
- ✧ The time required is independent of the number of people working on the project.



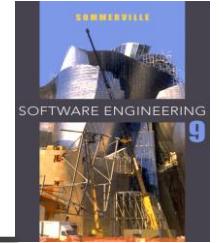
Staffing requirements

- ✧ Staff required can't be computed by diving the development time by the required schedule.
- ✧ The number of people working on a project varies depending on the phase of the project.
- ✧ The more people who work on the project, the more total effort is usually required.
- ✧ A very rapid build-up of people often correlates with schedule slippage.



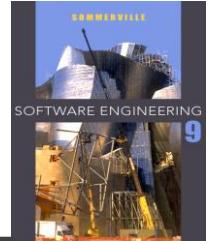
Key points

- ✧ Estimation techniques for software may be experience-based, where managers judge the effort required, or algorithmic, where the effort required is computed from other estimated project parameters.
- ✧ The COCOMO II costing model is an algorithmic cost model that uses project, product, hardware and personnel attributes as well as product size and complexity attributes to derive a cost estimate.



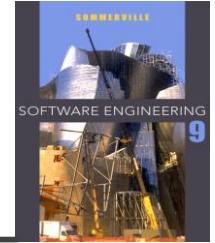
Chapter 20- Embedded Systems

Lecture 1



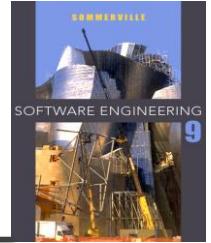
Topics covered

- ✧ Embedded systems design
- ✧ Architectural patterns
- ✧ Timing analysis
- ✧ Real-time operating systems



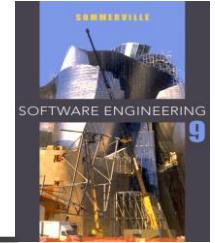
Embedded software

- ✧ 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.



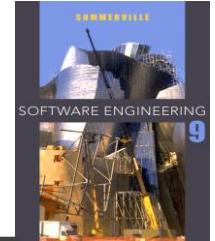
Responsiveness

- ✧ 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.



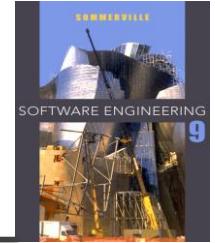
Definition

- ✧ 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.



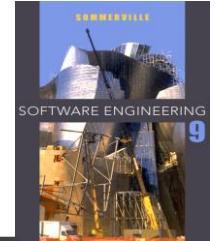
Embedded system characteristics

- ✧ 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.



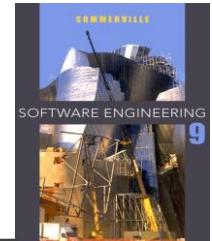
Embedded system design

- ✧ 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.



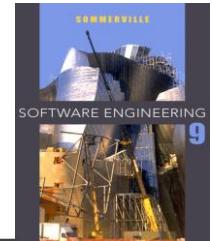
Reactive systems

- ✧ 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.

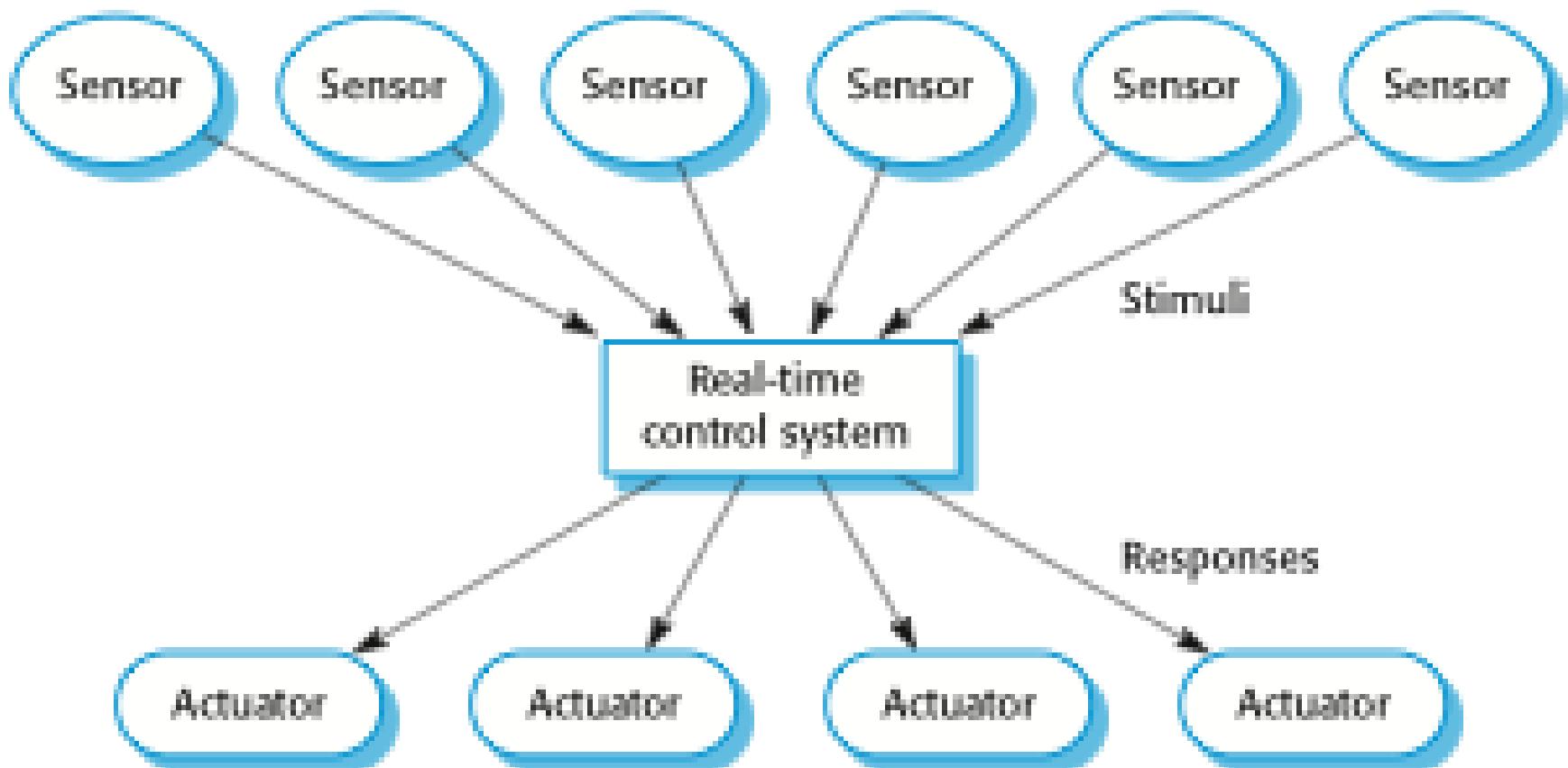


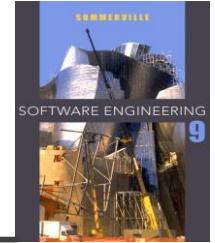
Stimuli and responses for a burglar alarm system

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.



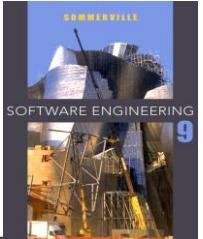
A general model of an embedded real-time system



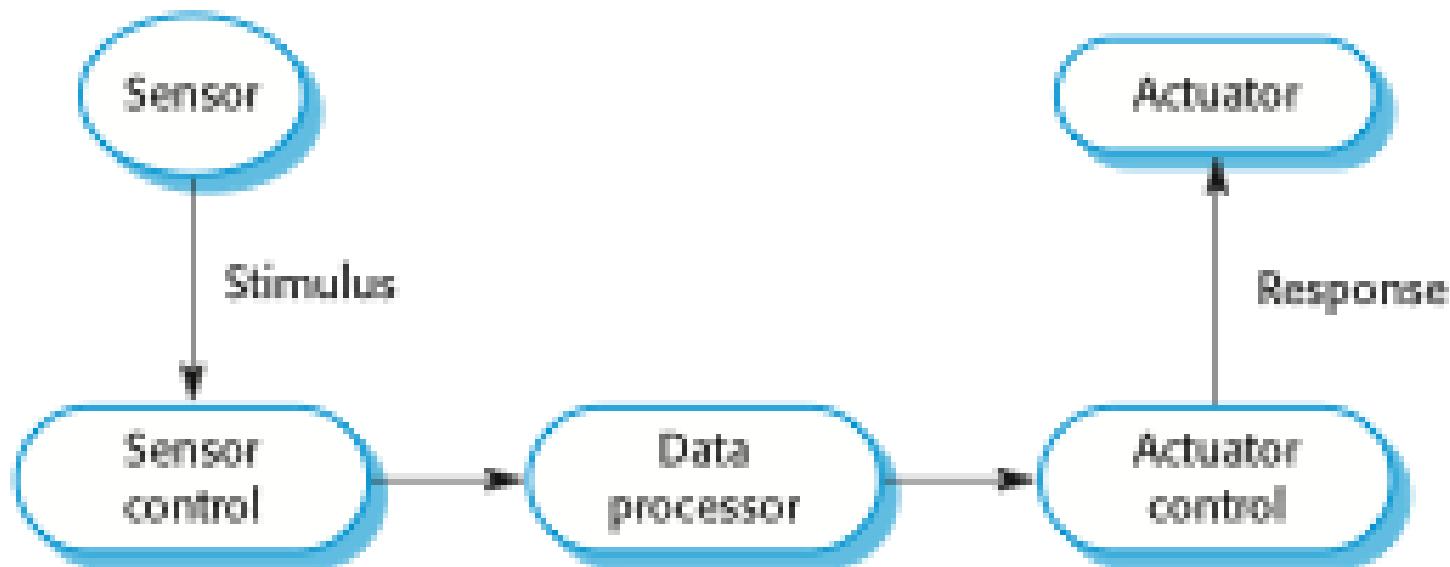


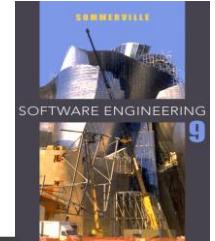
Architectural considerations

- ✧ 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

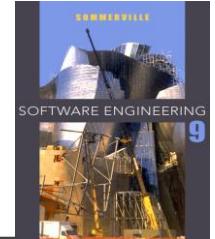
- Collect information from sensors. May buffer information collected in response to a sensor stimulus.

✧ Data processor

- Carries out processing of collected information and computes the system response.

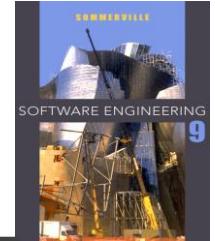
✧ Actuator control processes

- Generates control signals for the actuators.



Design process activities

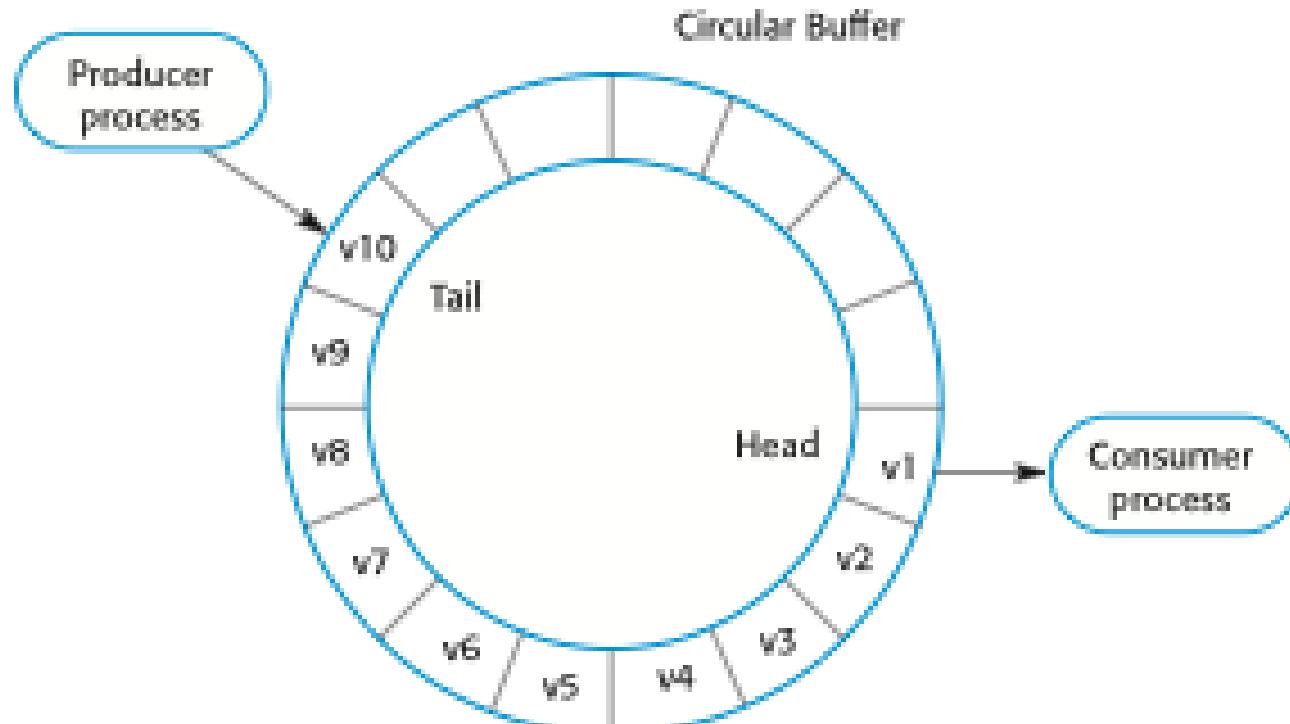
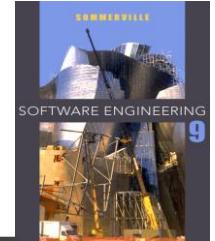
- ✧ Platform selection
- ✧ Stimuli/response identification
- ✧ Timing analysis
- ✧ Process design
- ✧ Algorithm design
- ✧ Data design
- ✧ Process scheduling

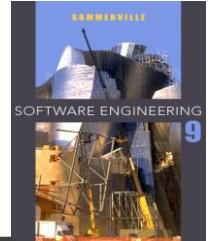


Process coordination

- ✧ 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.

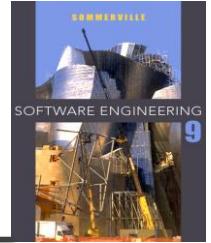
Producer/consumer processes sharing a circular buffer





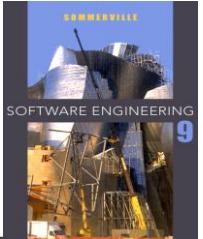
Mutual exclusion

- ✧ 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.

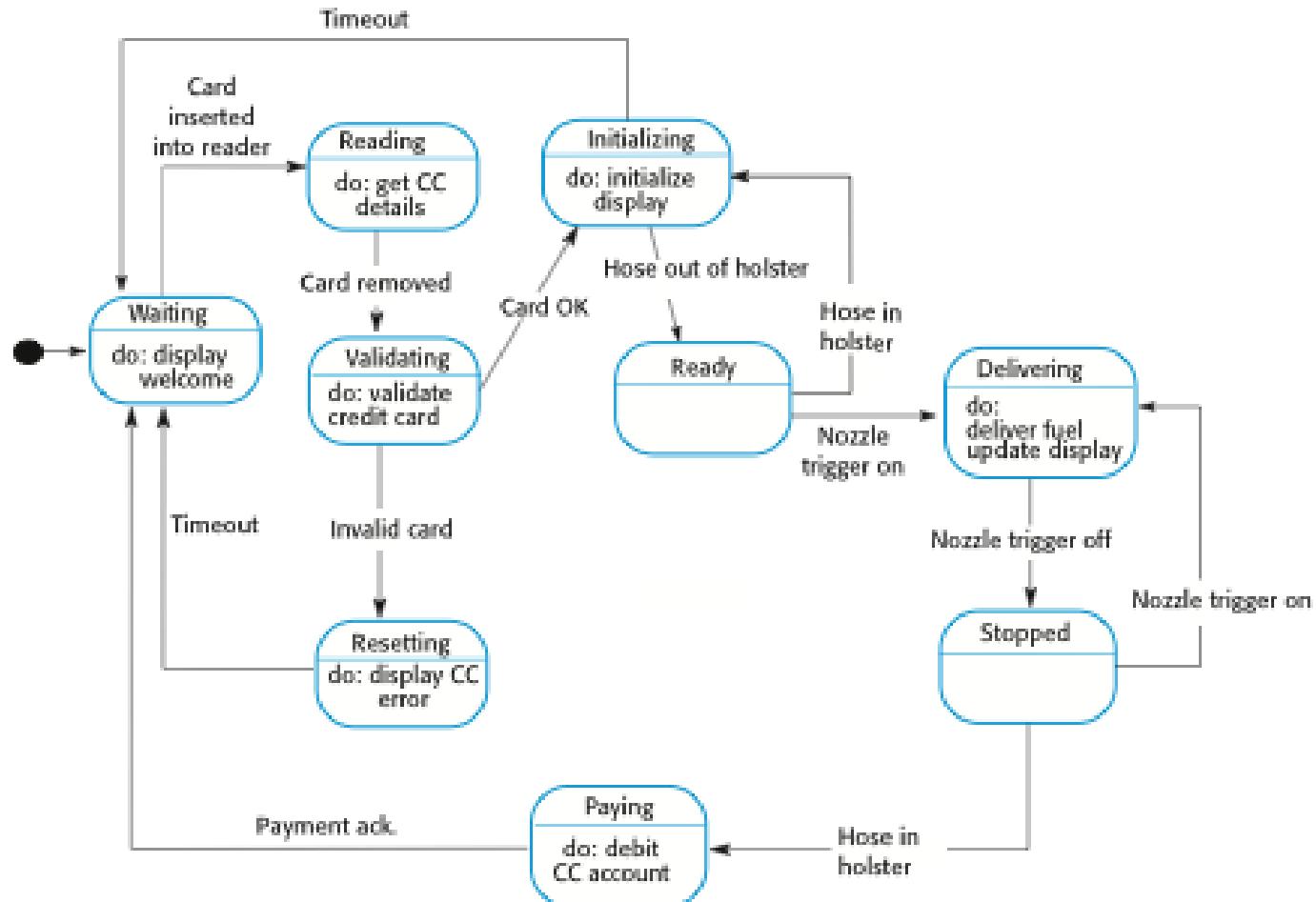


Real-time system modelling

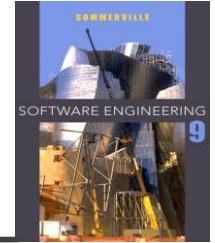
- ✧ The effect of a stimulus in a real-time system may trigger a transition from one state to another.
- ✧ State models are therefore often used to describe embedded real-time systems.
- ✧ UML state diagrams may be used to show the states and state transitions in a real-time system.



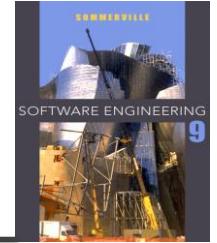
State machine model of a petrol (gas) pump

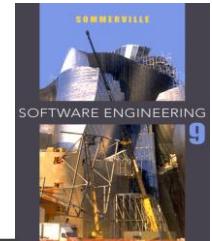


Real-time programming



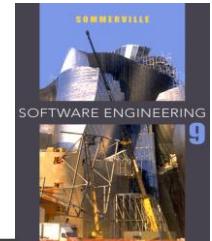
Architectural patterns for embedded systems



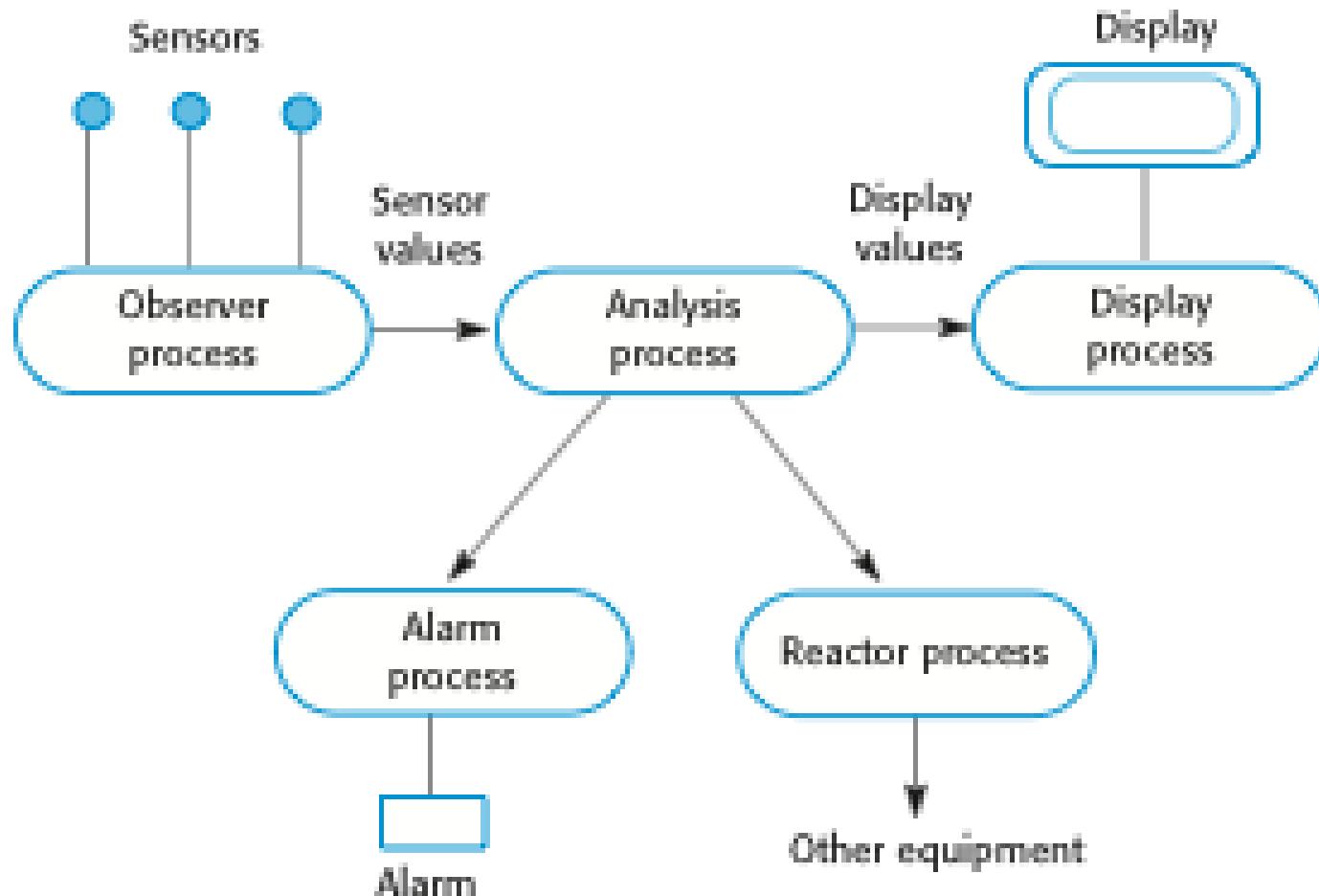


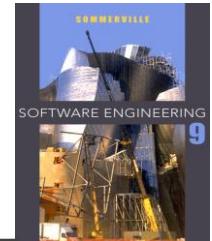
The Observe and React pattern

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.

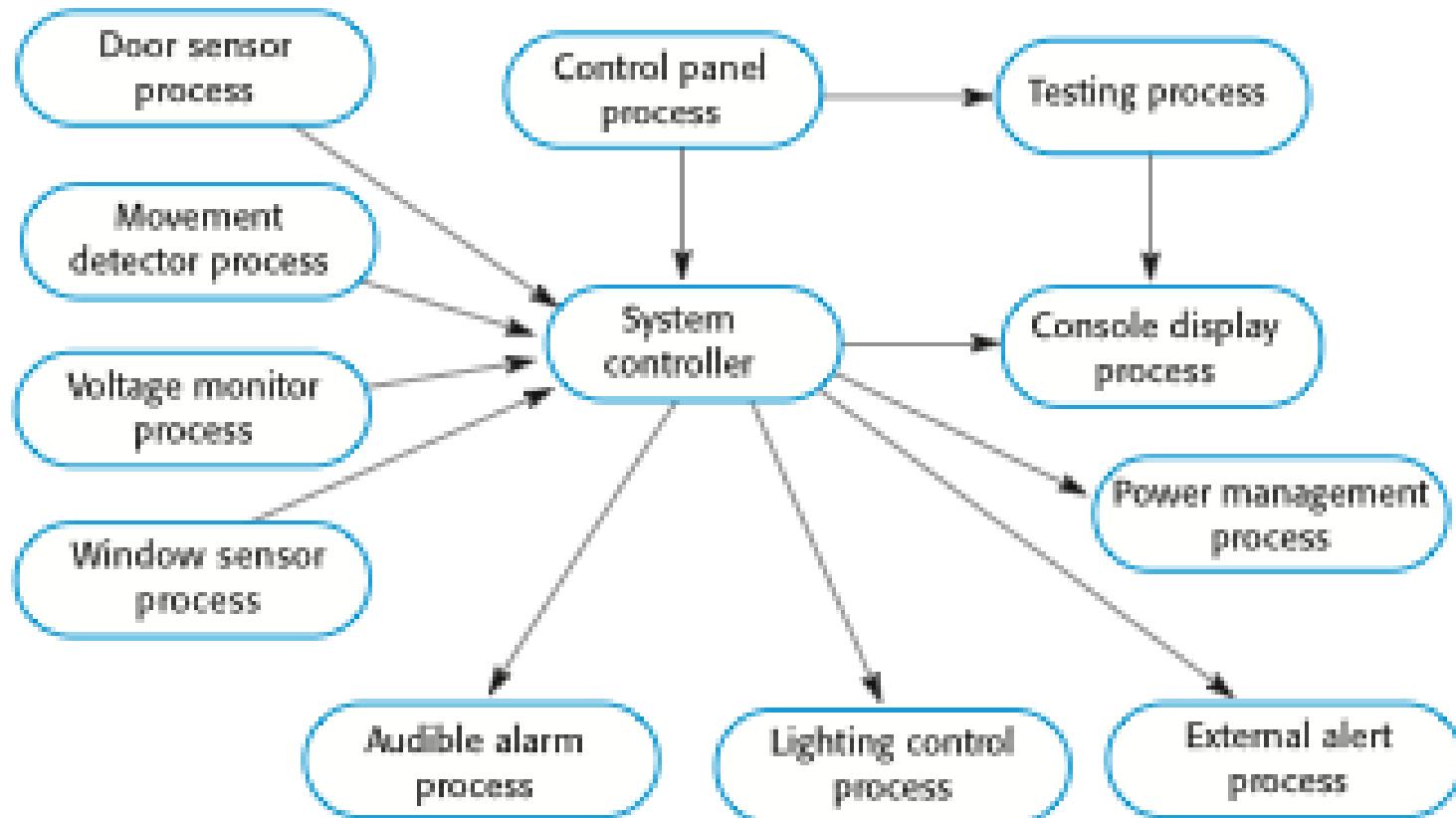


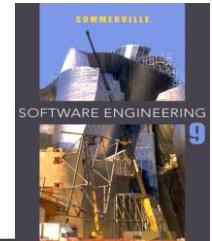
Observe and React process structure





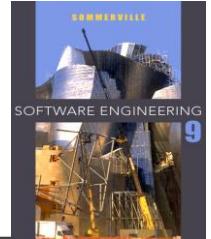
Process structure for a burglar alarm system



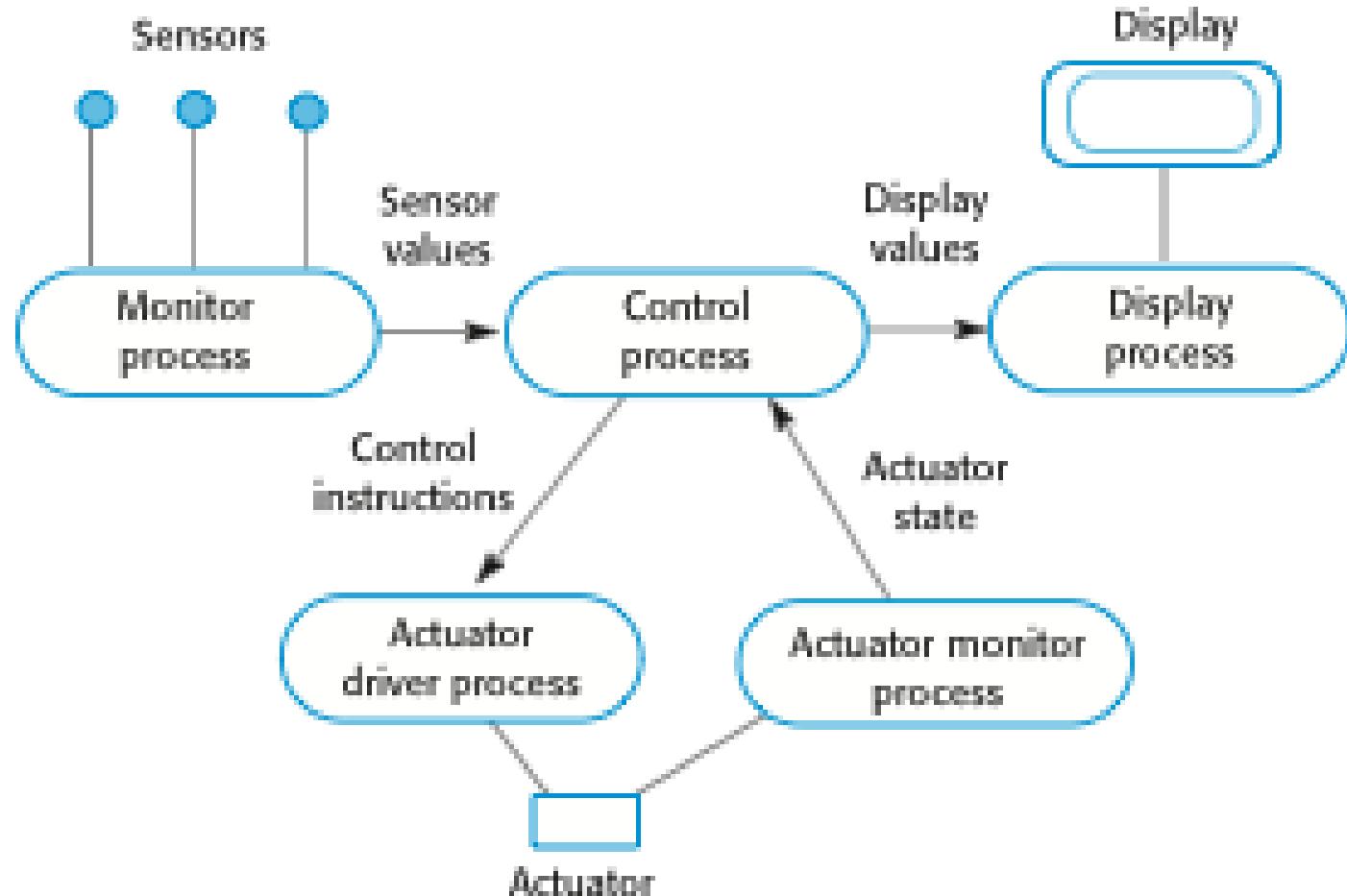


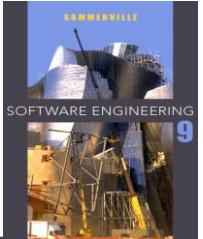
The Environmental Control pattern

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.

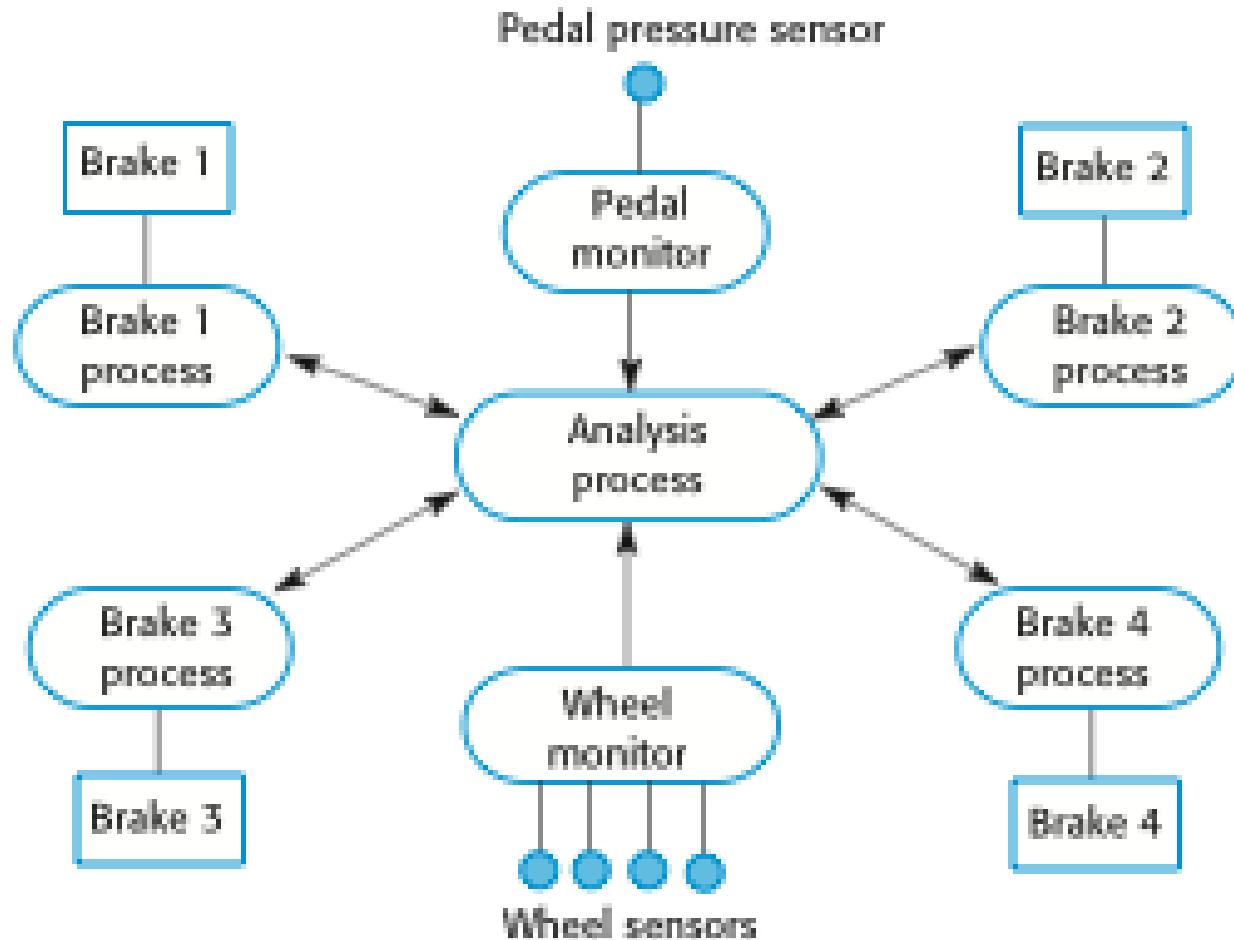


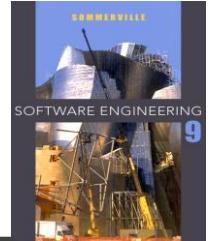
Environmental Control process structure





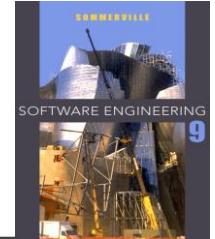
Control system architecture for an anti-skid braking system





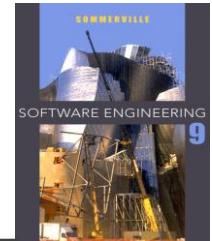
The Process Pipeline pattern

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

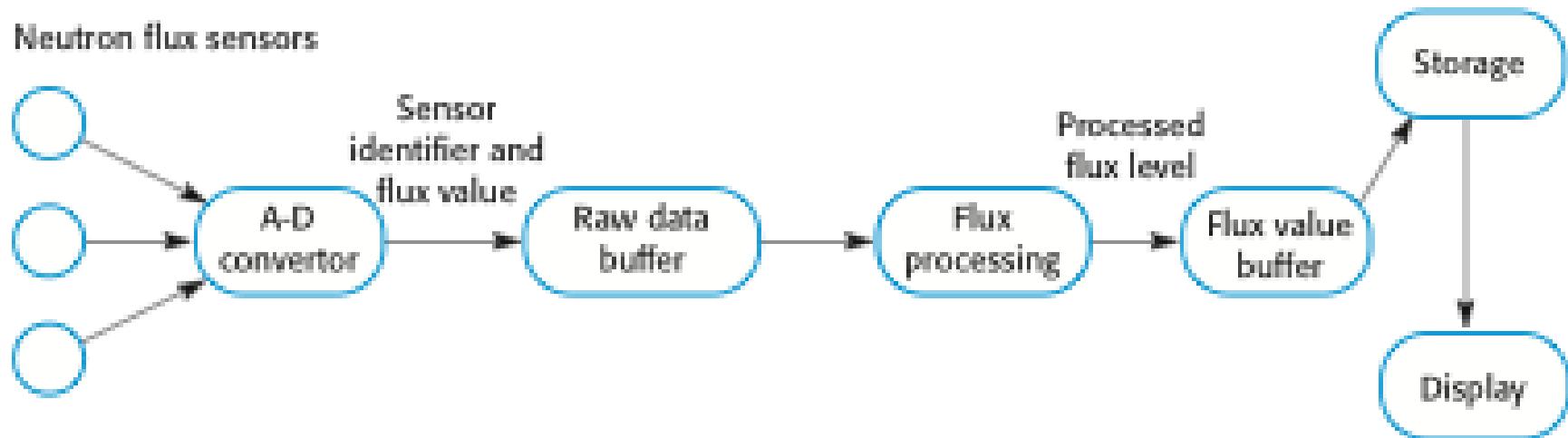


Process Pipeline process structure

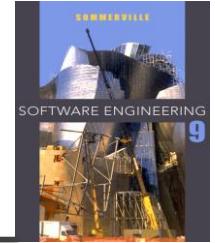


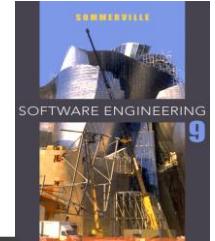


Neutron flux data acquisition



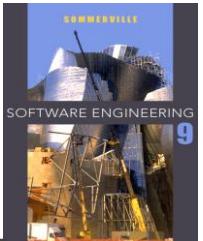
Timing analysis



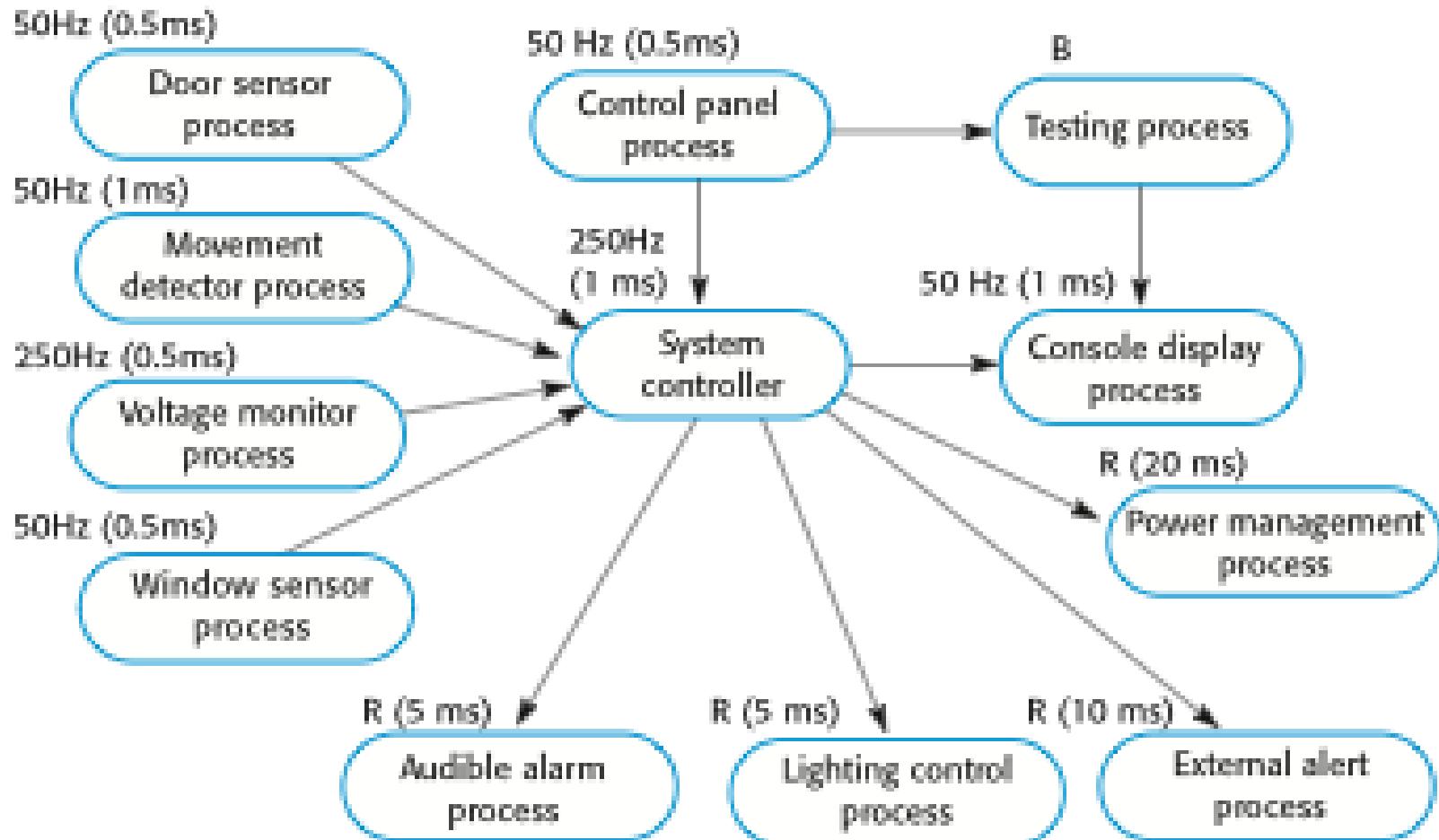


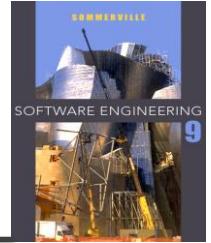
Timing requirements for the burglar alarm system

Stimulus/Response	Timing requirements
Power failure	The switch to backup power must be completed within a deadline of 50 ms.
Door alarm	Each door alarm should be polled twice per second.
Window alarm	Each window alarm should be polled twice per second.
Movement detector	Each movement detector should be polled twice per second.
Audible alarm	The audible alarm should be switched on within half a second of an alarm being raised by a sensor.
Lights switch	The lights should be switched on within half a second of an alarm being raised by a sensor.
Communications	The call to the police should be started within 2 seconds of an alarm being raised by a sensor.
Voice synthesizer	A synthesized message should be available within 2 seconds of an alarm being raised by a sensor.

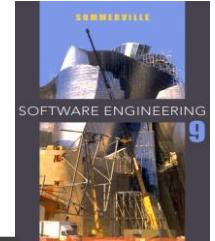


Alarm process timing



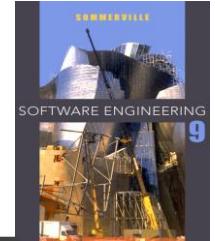


Real-time operating systems



Operating system components

- ✧ Real-time clock
 - Provides information for process scheduling.
- ✧ Interrupt handler
 - Manages aperiodic requests for service.
- ✧ Scheduler
 - Chooses the next process to be run.
- ✧ Resource manager
 - Allocates memory and processor resources.
- ✧ Dispatcher
 - Starts process execution.



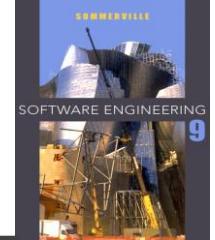
Non-stop system components

✧ Configuration manager

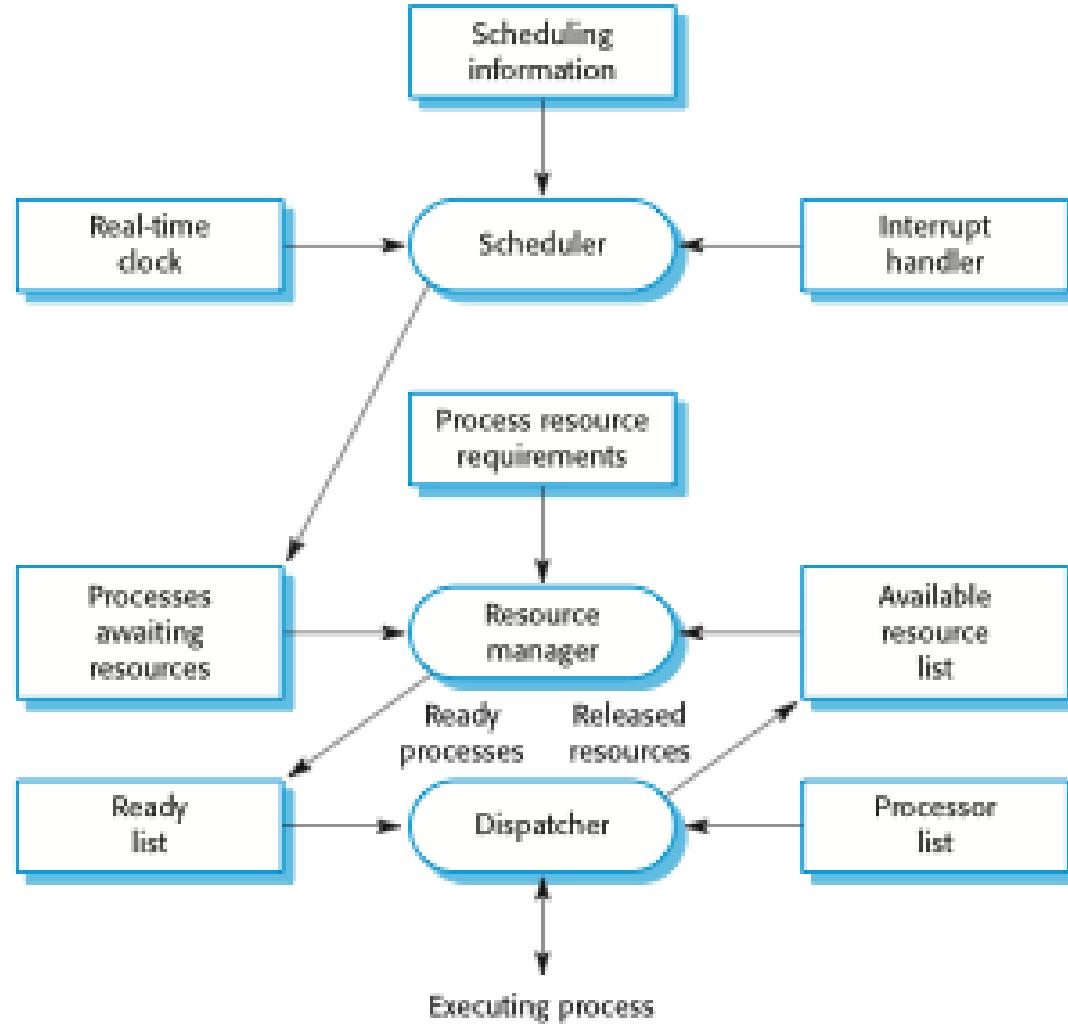
- Responsible for the dynamic reconfiguration of the system software and hardware. Hardware modules may be replaced and software upgraded without stopping the systems.

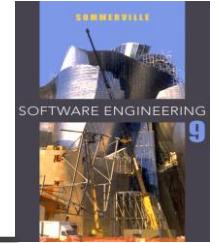
✧ Fault manager

- Responsible for detecting software and hardware faults and taking appropriate actions (e.g. switching to backup disks) to ensure that the system continues in operation.



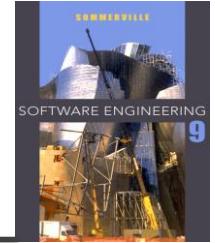
Components of a real-time operating system





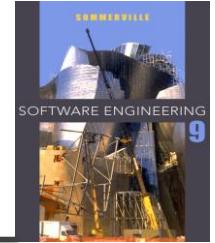
Process priority

- ✧ The processing of some types of stimuli must sometimes take priority.
- ✧ Interrupt level priority. Highest priority which is allocated to processes requiring a very fast response.
- ✧ Clock level priority. Allocated to periodic processes.
- ✧ Within these, further levels of priority may be assigned.



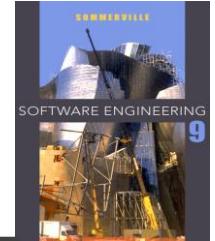
Interrupt servicing

- ✧ Control is transferred automatically to a pre-determined memory location.
- ✧ This location contains an instruction to jump to an interrupt service routine.
- ✧ Further interrupts are disabled, the interrupt serviced and control returned to the interrupted process.
- ✧ Interrupt service routines MUST be short, simple and fast.



Periodic process servicing

- ✧ In most real-time systems, there will be several classes of periodic process, each with different periods (the time between executions), execution times and deadlines (the time by which processing must be completed).
- ✧ The real-time clock ticks periodically and each tick causes an interrupt which schedules the process manager for periodic processes.
- ✧ The process manager selects a process which is ready for execution.



Process management

- ✧ Concerned with managing the set of concurrent processes.
- ✧ Periodic processes are executed at pre-specified time intervals.
- ✧ The RTOS uses the real-time clock to determine when to execute a process taking into account:
 - Process period - time between executions.
 - Process deadline - the time by which processing must be complete.

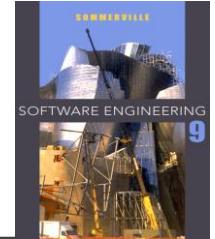
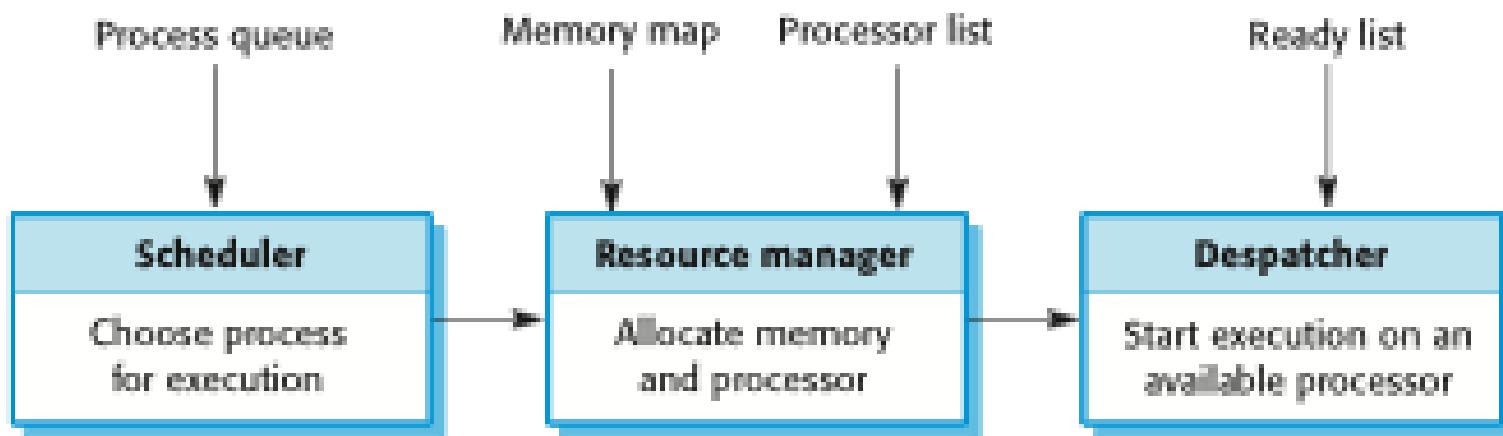
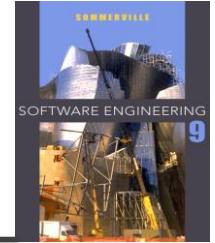


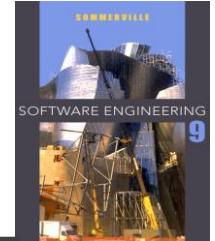
Figure 20.18 RTOS actions required to start a process





Process switching

- ✧ The scheduler chooses the next process to be executed by the processor. This depends on a scheduling strategy which may take the process priority into account.
- ✧ The resource manager allocates memory and a processor for the process to be executed.
- ✧ The dispatcher takes the process from ready list, loads it onto a processor and starts execution.



Scheduling strategies

✧ Non pre-emptive scheduling

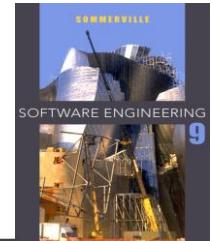
- Once a process has been scheduled for execution, it runs to completion or until it is blocked for some reason (e.g. waiting for I/O).

✧ Pre-emptive scheduling

- The execution of an executing processes may be stopped if a higher priority process requires service.

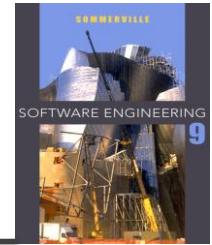
✧ Scheduling algorithms

- Round-robin;
- Rate monotonic;
- Shortest deadline first.



Key points

- ✧ An embedded software system is part of a hardware/software system that reacts to events in its environment. The software is 'embedded' in the hardware. Embedded systems are normally real-time systems.
- ✧ A real-time system is a software system that must respond to events in real time. System correctness does not just depend on the results it produces, but also on the time when these results are produced.
- ✧ Real-time systems are usually implemented as a set of communicating processes that react to stimuli to produce responses.
- ✧ State models are an important design representation for embedded real-time systems. They are used to show how the system reacts to its environment as events trigger changes of state in the system.



Key points

- ✧ There are several standard patterns that can be observed in different types of embedded system. These include a pattern for monitoring the system's environment for adverse events, a pattern for actuator control and a data-processing pattern.
- ✧ Designers of real-time systems have to do a timing analysis, which is driven by the deadlines for processing and responding to stimuli. They have to decide how often each process in the system should run and the expected and worst-case execution time for processes.
- ✧ A real-time operating system is responsible for process and resource management. It always includes a scheduler, which is the component responsible for deciding which process should be scheduled for execution.

Real-time Software Engineering

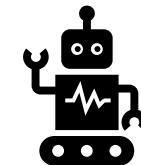
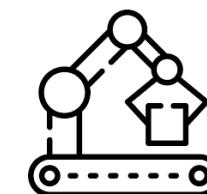
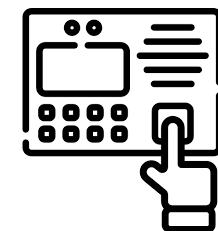
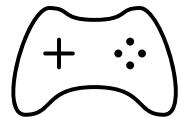
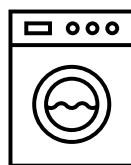
CS3023 – Software Engineering

Topics covered

- ✧ Embedded system design
- ✧ Architectural patterns for real-time software
- ✧ Timing analysis
- ✧ Real-time operating systems

Embedded software

- ✧ Computers control a wide range of systems
 - E.g. simple domestic machines, games controllers, manufacturing plants...
- ✧ Their software
 - Must react to events generated by the hardware
 - Issue control signals in response to these events
 - Is embedded in system hardware, often in read-only memory (ROM)
 - 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.
- ✧ Correctness
 - For non-real-time systems, defined by specifying how system inputs map to corresponding outputs
 - In a real-time system, 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.

Examples of hard and soft real-time systems

Vehicle braking system	Hard
Insulin monitoring and delivery system	Soft
Wilderness weather station	Soft
Microwave oven	Hard
Missile guidance system	Hard

Characteristics of embedded systems

1. Embedded systems generally run continuously and do not terminate.
2. Unpredictable interactions with the system's environment.
3. There may be physical limitations that affect the design of a system.
4. Direct hardware interaction may be necessary.
5. Issues of safety and reliability may dominate the system design.

Embedded system design

Embedded system design

Design process for embedded systems:

- ✧ Is a systems engineering process that considers, in detail, the design and performance of the system hardware.
- ✧ 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.
- ✧ May have to include additional software functionality, such as battery and power management.

Reactive systems

Real-time systems are often considered to be **reactive systems**.

- ✧ Given a stimulus, the system must produce a reaction or response within a specified time.

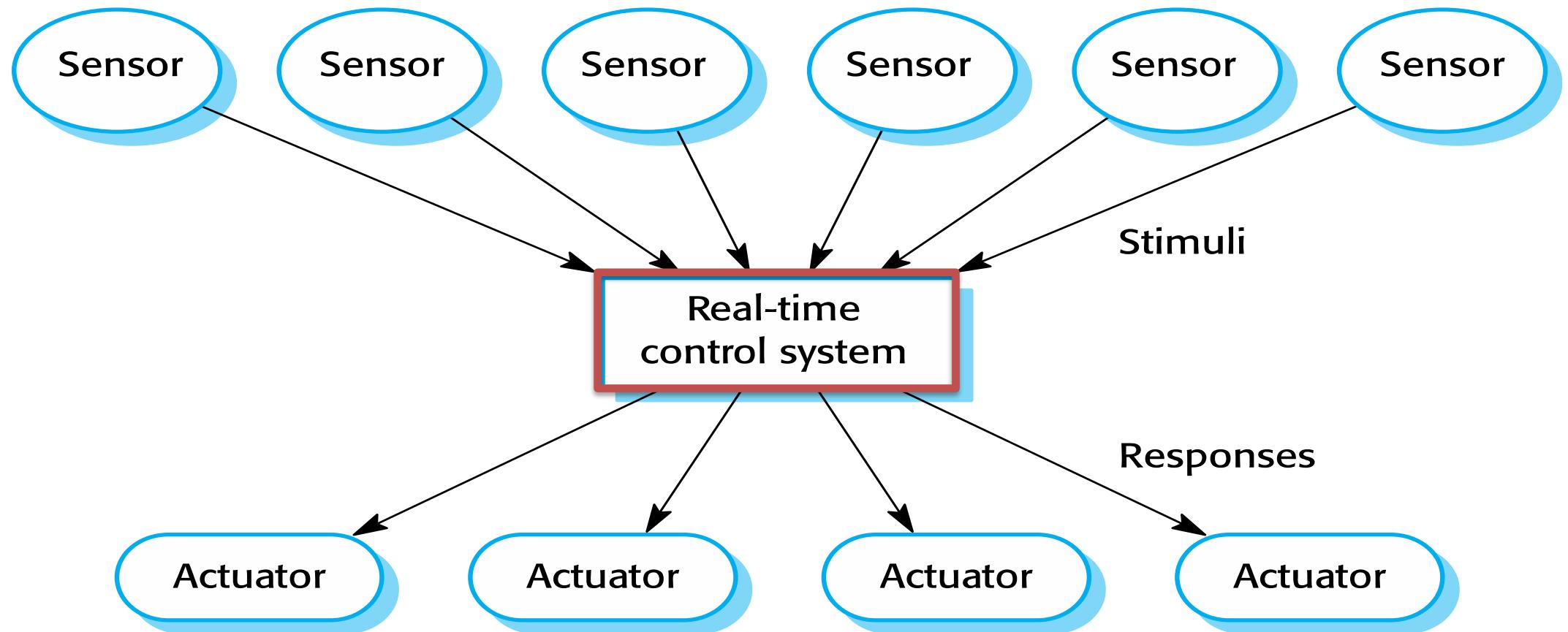
- ✧ **Periodic stimuli.** Stimuli which occur at predictable time intervals
 - E.g., read a temperature sensor every 100 millisecond (and respond depending on sensor value)

- ✧ **Aperiodic stimuli.** Stimuli which occur at unpredictable times
 - E.g., a system power failure may trigger an interrupt which must be processed by the system.

Example: stimuli and responses for a burglar alarm system

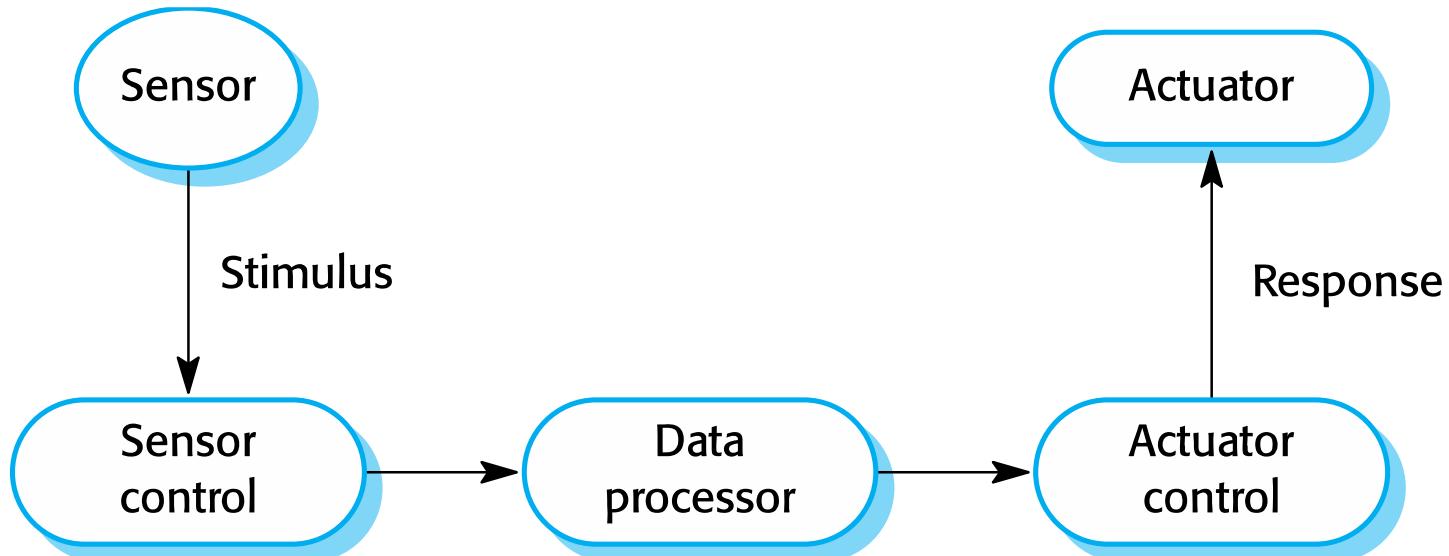
Stimulus	Response
Clear alarms	Switch off all active alarms; switch off all lights that have been switched on.
Console panic button positive	Initiate alarm; turn on lights around console; call police.
Power supply failure	Call service technician.
Sensor failure	Call service technician.
Single sensor 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.

A general model of an embedded real-time system



Sensor and actuator processes

A separate process for each type of sensor and actuator



A general design guideline

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.

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.

Design process activities

1. Platform selection
2. Stimuli/response identification
3. Timing analysis
4. Process design
5. Algorithm design
6. Data design
7. Process scheduling

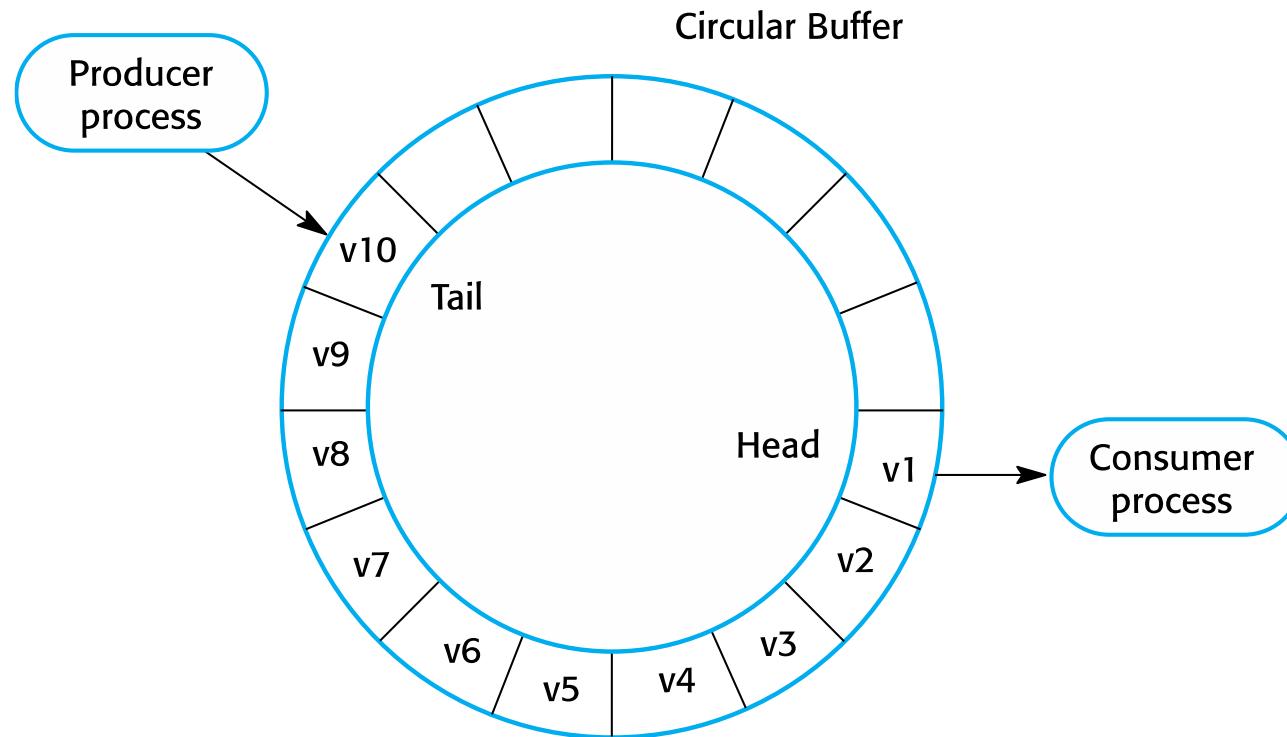
Process coordination

- ✧ Real-time system processes must 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, need to consider that these processes may be running at different speeds.

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.

Producer/consumer processes sharing a circular buffer



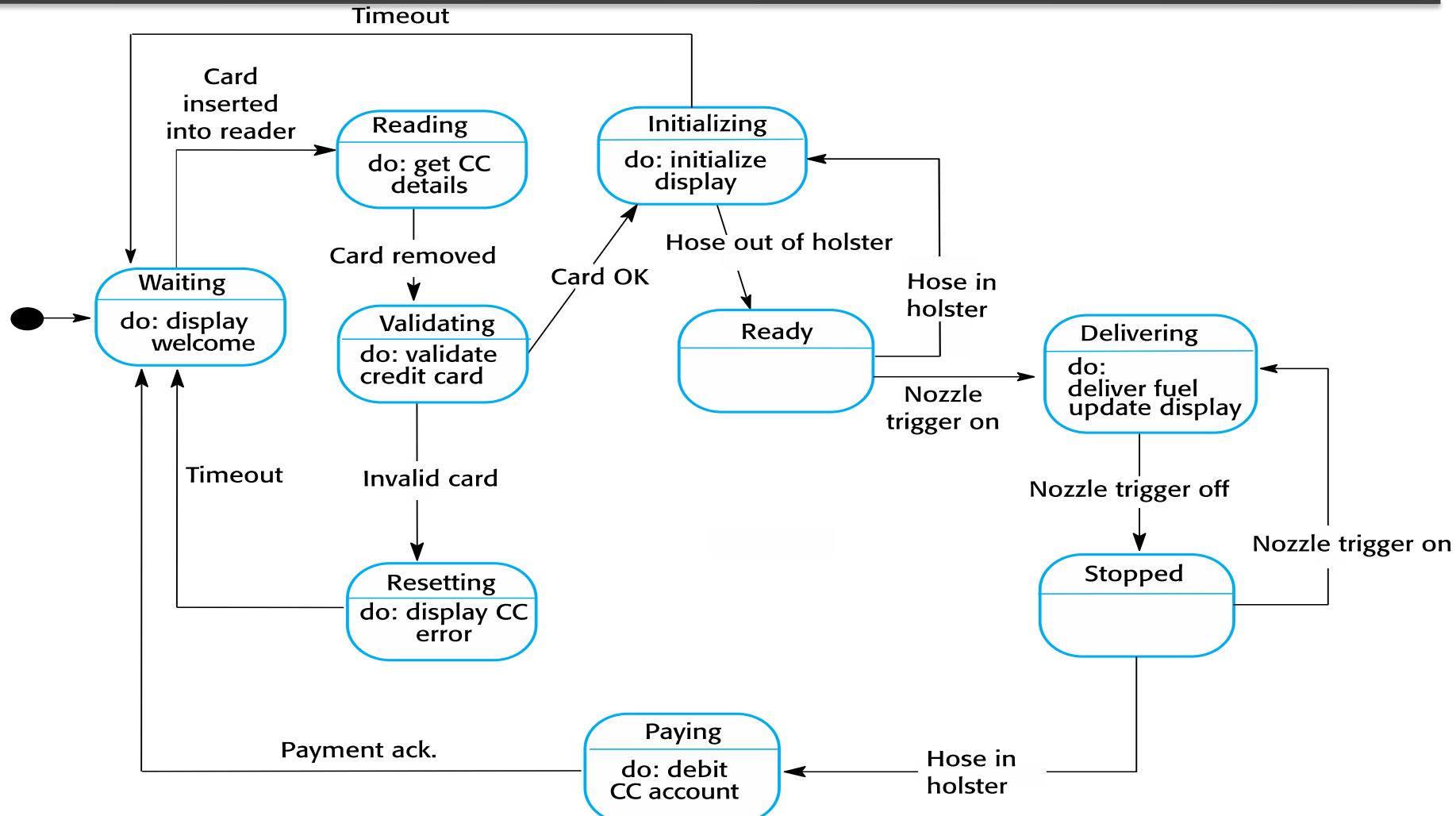
Real-time system modelling

- ✧ The effect of a stimulus in a real-time system may trigger a transition from one state to another.
- ✧ State models are 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 pump



State machine model of a petrol pump



Sequence of actions in real-time pump control system

- ✧ The buyer inserts a credit card into a card reader built into the pump.
- ✧ Removal of the card triggers a transition to a Validating state where the card is validated.
- ✧ If the card is valid, the system initializes the pump and, when the fuel hose is removed from its holster, transitions to the Delivering state.
- ✧ After the fuel delivery is complete and the hose replaced in its holster, the system moves to a Paying state
- ✧ After payment, the pump software returns to the Waiting state

Real-time programming

✧ Programming languages for real-time systems development

- facilities to access system hardware
- possible to predict the timing of operations
- C, Assembly, ~~Java~~
- ~~Object-oriented languages~~

Performance overhead due to extra code to mediate access to attributes and handle calls to operations -> maybe impossible to meet real-time deadlines.

Architectural patterns for real-time software

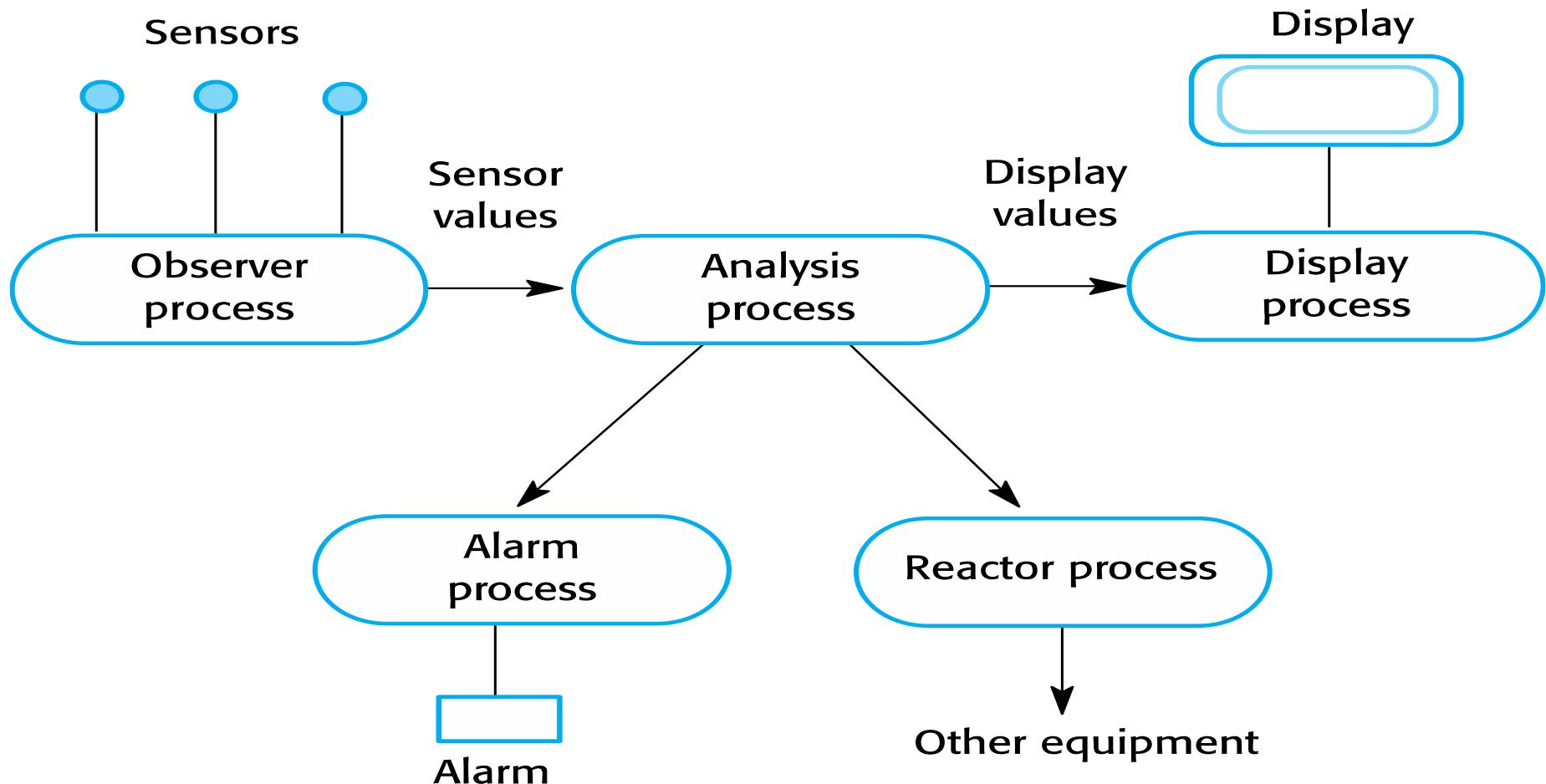
Architectural patterns for embedded systems

1. *Observe and React* - Used when a set of sensors are routinely monitored and displayed.
2. *Environmental Control* - Used when a system includes sensors, which provide information about the environment and actuators that can change the environment
3. *Process Pipeline* - Used when data has to be transformed from one representation to another before it can be processed.

1) 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

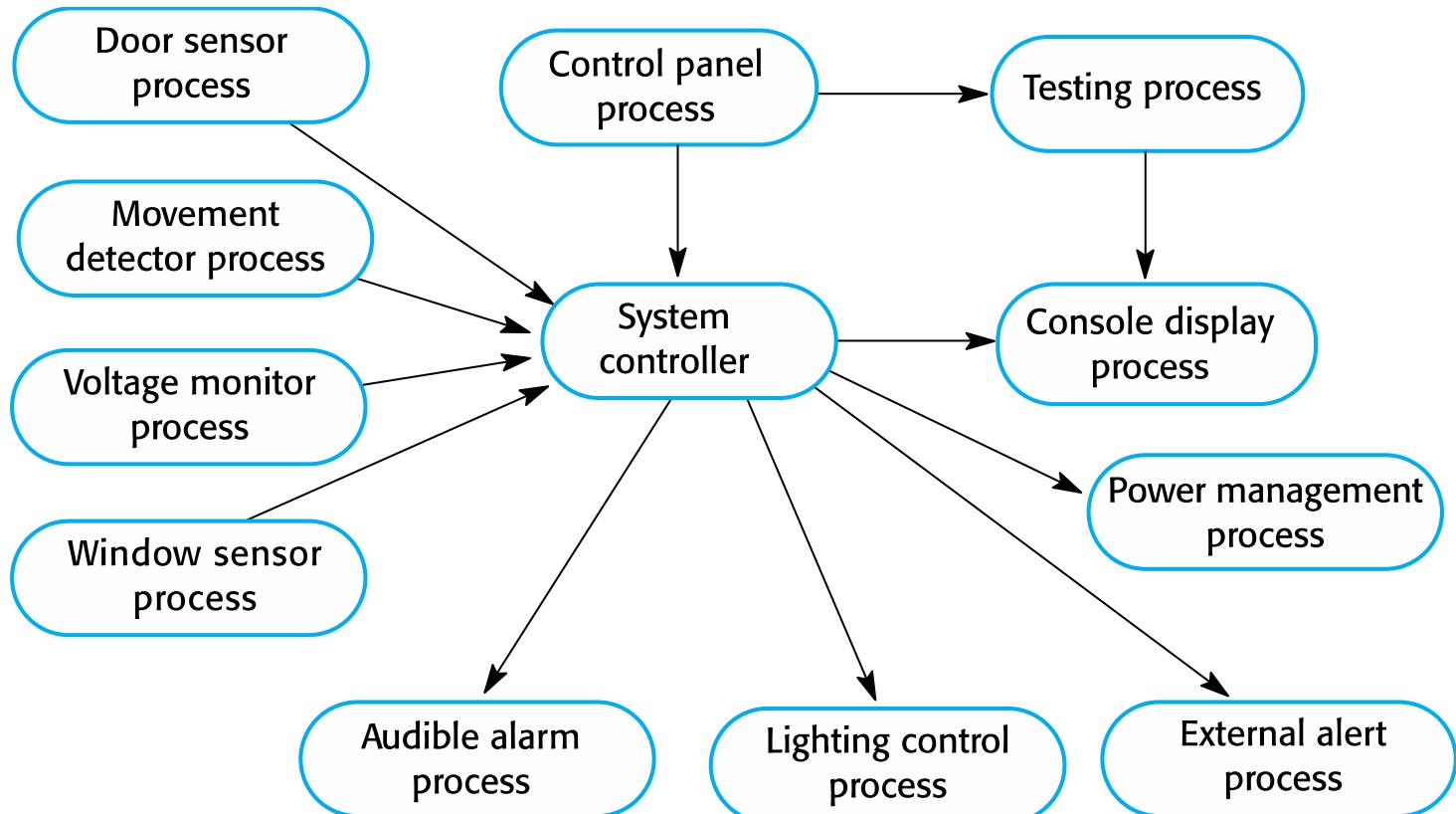


Example: Alarm system description

A software system is to be implemented as part of a burglar alarm system for commercial buildings. This uses several different types of sensor. These include movement detectors in individual rooms, door sensors that detect corridor doors opening, and window sensors on ground-floor windows that detect when a window has been opened.

When a sensor detects the presence of an intruder, the system automatically calls the local police and, using a voice synthesizer, reports the location of the alarm. It switches on lights in the rooms around the active sensor and sets off an audible alarm. The sensor system is normally powered by mains power but is equipped with a battery backup. Power loss is detected using a separate power circuit monitor that monitors the mains voltage. If a voltage drop is detected, the system assumes that intruders have interrupted the power supply, so an alarm is raised.

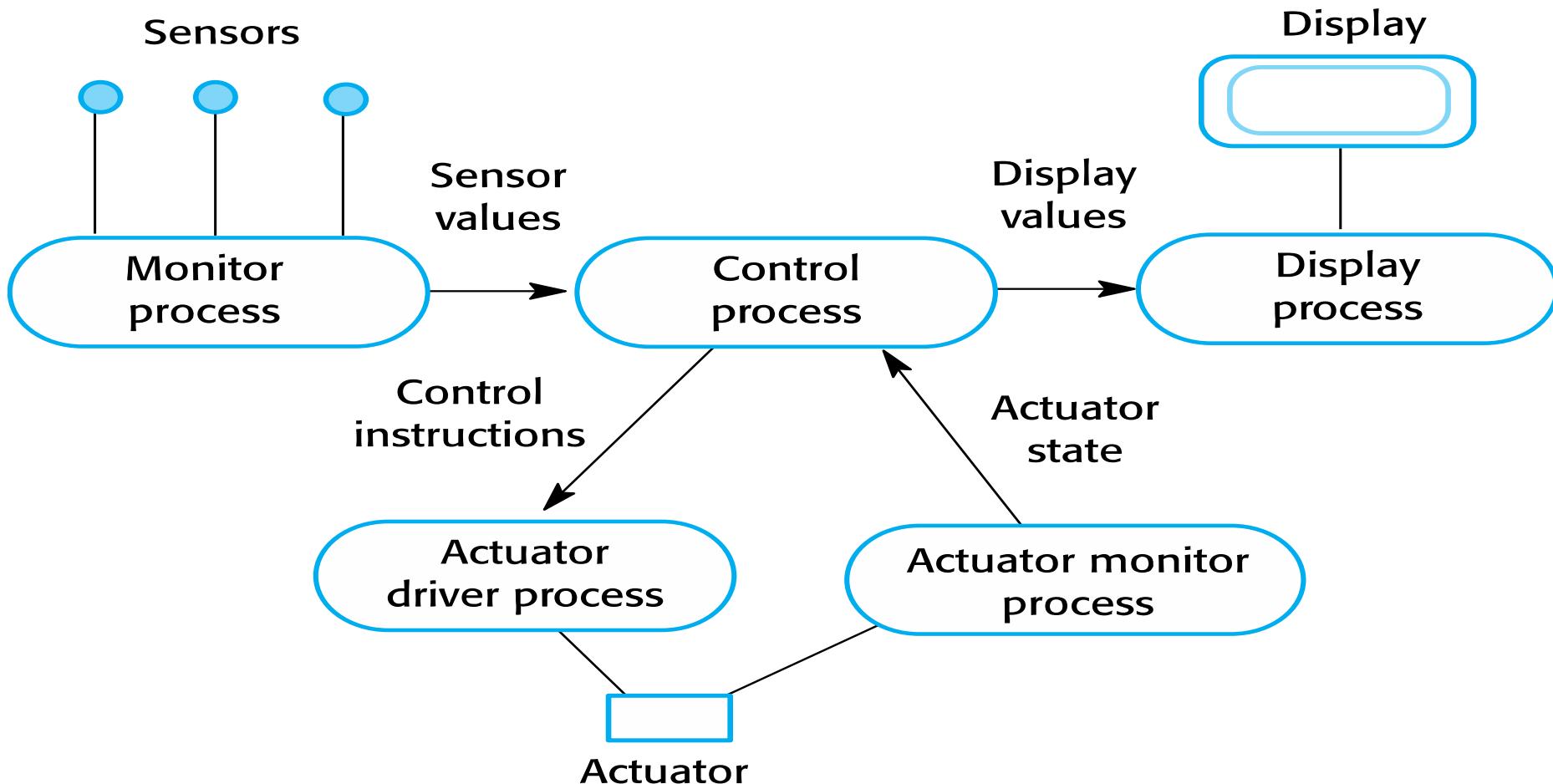
Example: Process structure for a burglar alarm system



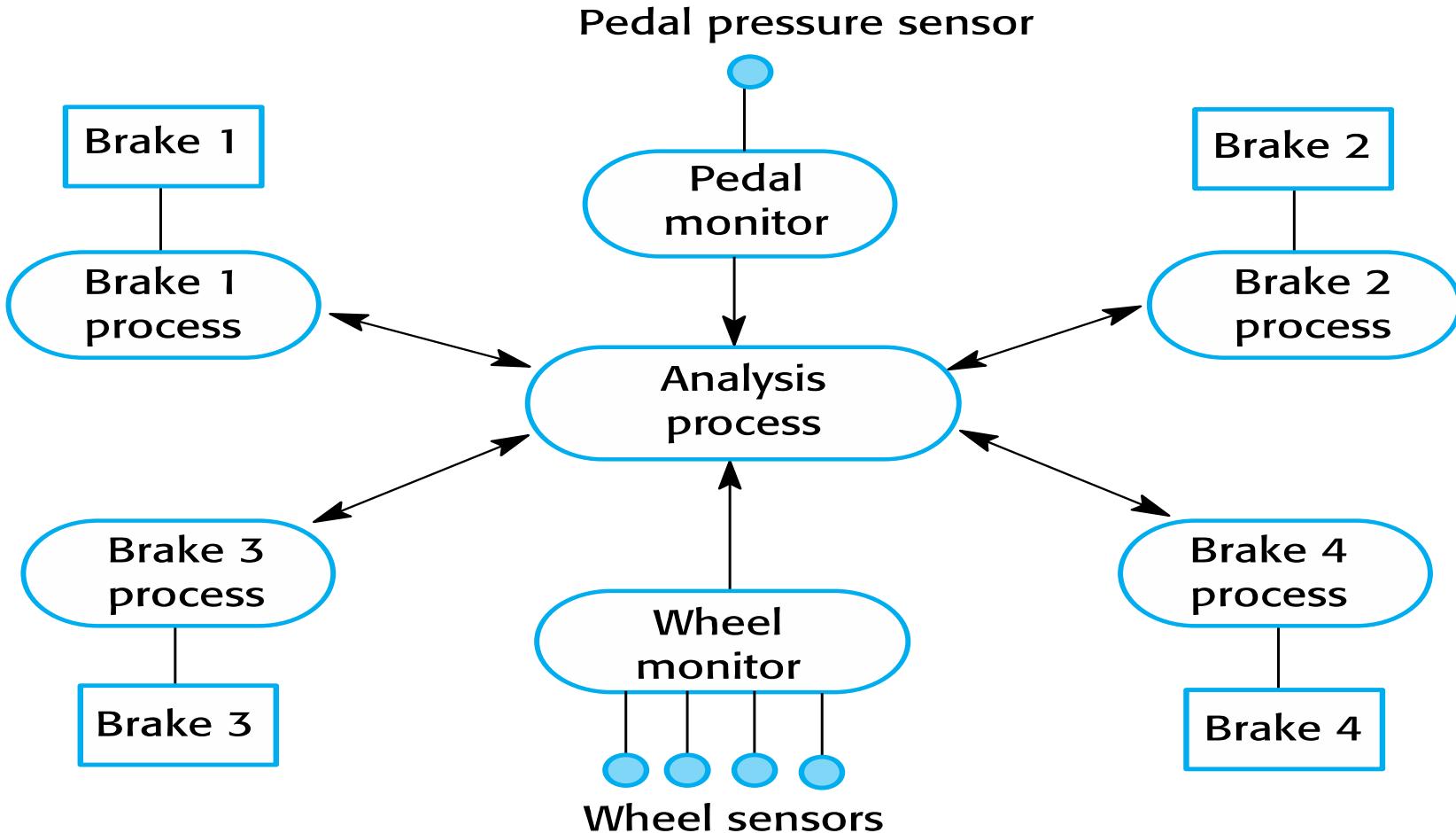
2) The Environmental Control pattern

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



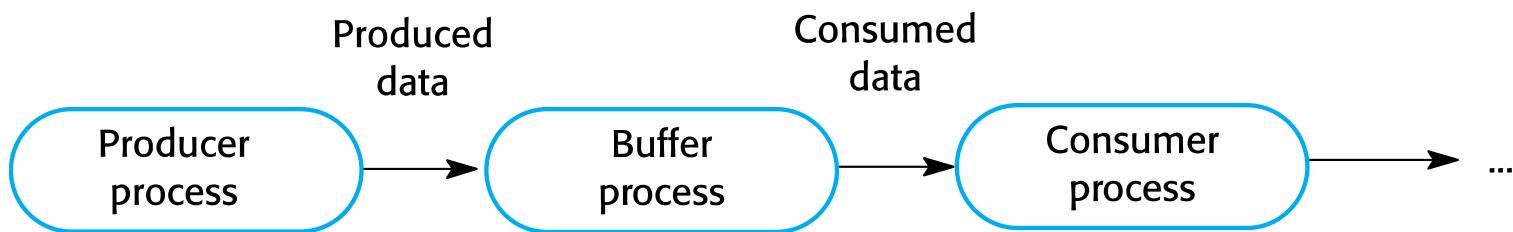
Example: Control system architecture for an anti-skid braking system



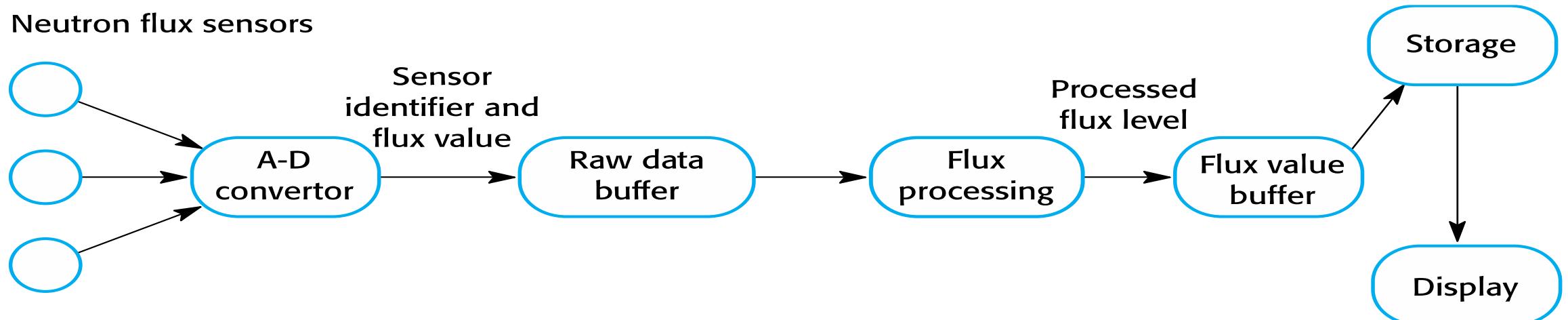
3) The Process Pipeline pattern

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



Example: Neutron flux data acquisition



Part of control software for a nuclear reactor

Timing analysis

Timing analysis

Correctness of a real-time system depends on the correctness of its outputs AND on the time at which these outputs were produced.

Calculate how often each process in the system must be executed to ensure that all inputs are processed, and all system responses produced in a timely way.

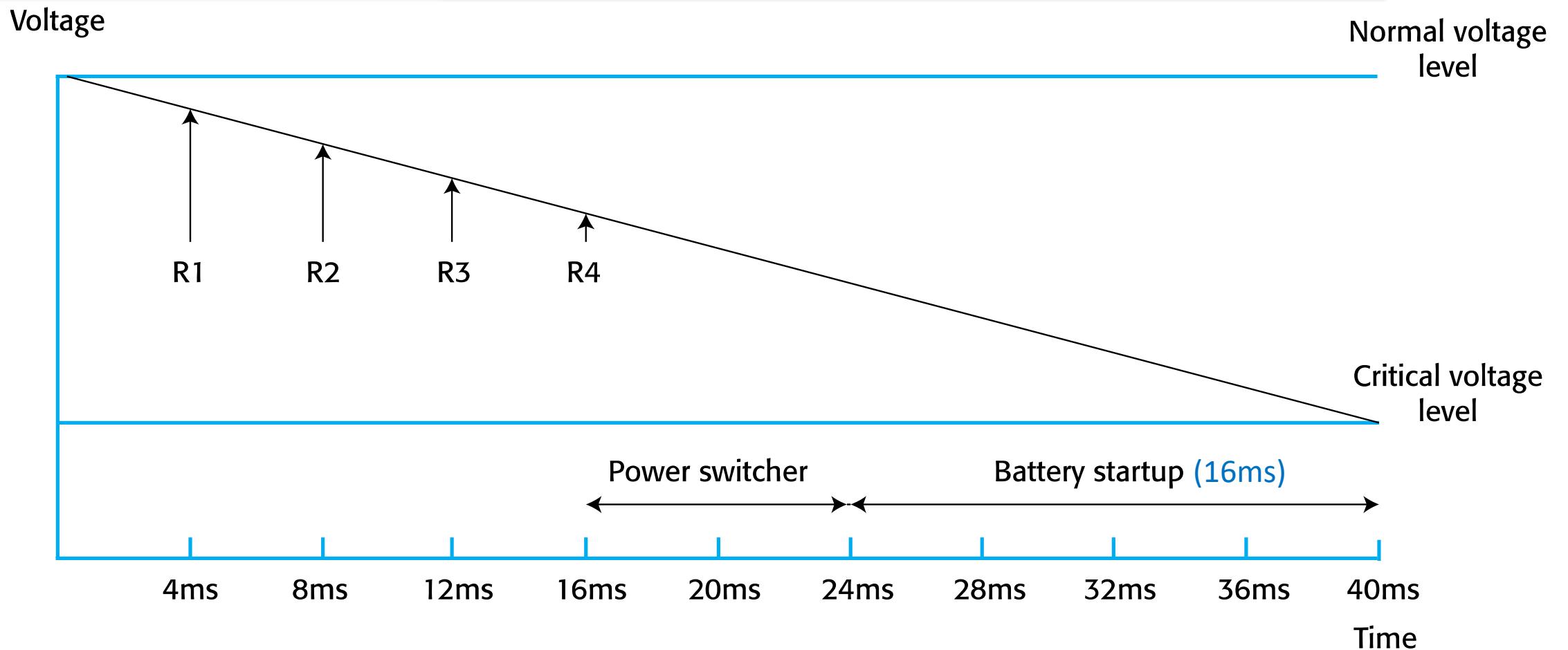
✧ Results of timing analysis are used to decide:

- How frequently each process should execute and
- How these processes should be **scheduled** by the RTOS.

Factors in timing analysis

1. *Deadlines* - The times by which stimuli must be processed and some response produced by the system.
2. *Frequency* - Number of times per second that a process must execute (to be confident that it can always meet its deadlines).
3. *Execution time* - Time required to process a stimulus and produce a response.

Example: Power failure timing analysis



Example: Power failure timings

- ✧ It takes 50 milliseconds (ms) for the supplied voltage to drop to a level where the equipment may be damaged. The battery backup must therefore be activated and in operation within 50ms.
- ✧ It takes 16ms from starting the backup power supply to the supply being fully operational.
- ✧ There is a checking process that is scheduled to run 250 times per second (i.e. every 4ms).
 - This process assumes that there is a power supply problem if there is a significant drop in voltage between readings and this is sustained for 3 readings.

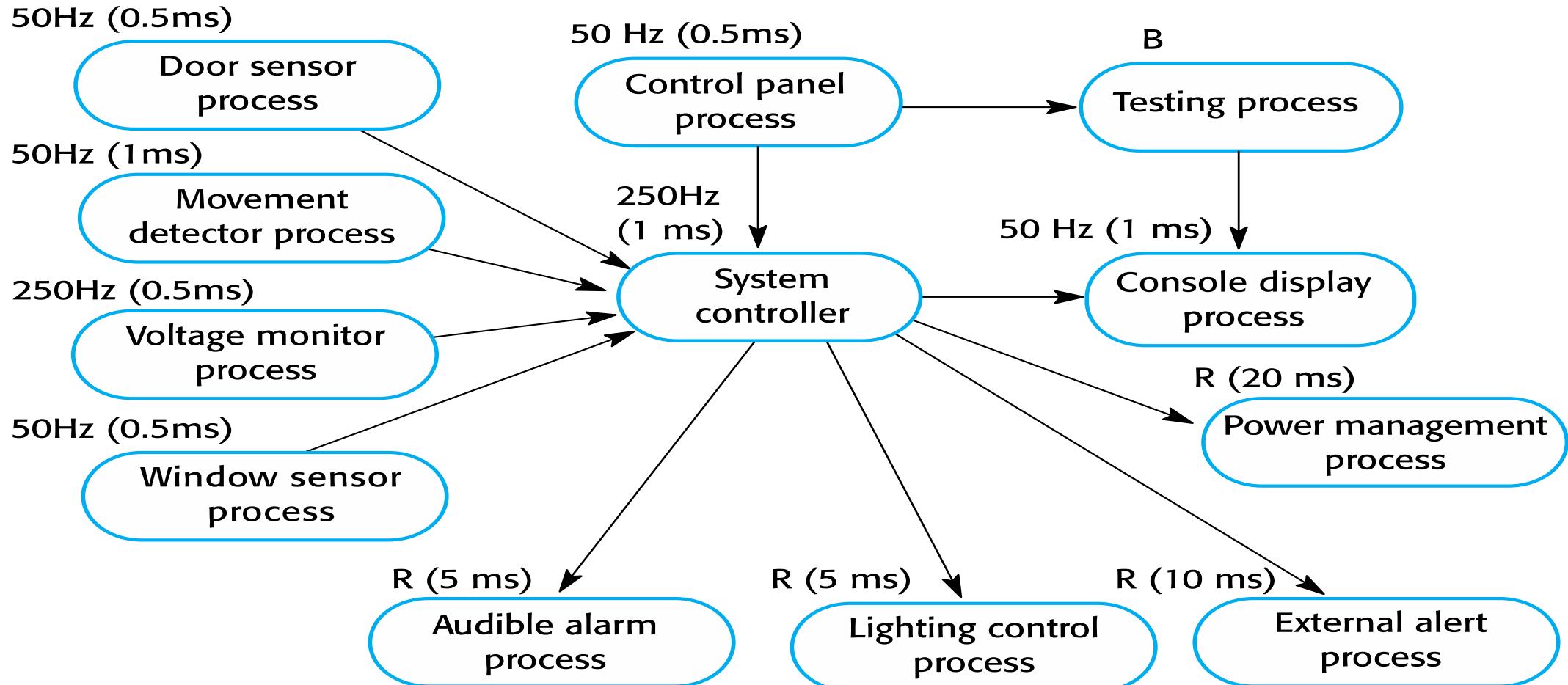
Example: Power failure timings

- ✧ Assume the power fails immediately after a reading has been taken. Therefore reading R1 is the start reading for the power fail check.
- ✧ Voltage continues to drop for readings R2–R4, so a power failure is assumed. (This is the worst possible case)
- ✧ At this stage, the process to switch to the battery backup is started. Because the battery backup takes 16ms to become operational, this means that the worst-case execution time for this process is **8ms**.

Example: Timing requirements for the burglar alarm system

Stimulus/Response	Timing requirements
Audible alarm	The audible alarm should be switched on within <u>half a second</u> of an alarm being raised by a sensor.
Communications	The call to the police should be started within <u>2 seconds</u> of an alarm being raised by a sensor.
Door alarm	Each door alarm should be <u>polled twice per second</u> .
Lights switch	The lights should be switched on within <u>half a second</u> of an alarm being raised by a sensor.
Movement detector	Each movement detector should be <u>polled twice per second</u> .
Power failure	The switch to backup power must be completed within a <u>deadline of 50 ms</u> .
Voice synthesizer	A synthesized message should be available within <u>2 seconds</u> of an alarm being raised by a sensor.
Window alarm	Each window alarm should be <u>polled twice per second</u> .

Example: Alarm process timing



Stimuli to be processed

- ✧ Power failure is detected by observing a voltage drop of more than 20%.
 - The required response is to switch the circuit to backup power by signalling an electronic power-switching device that switches the mains power to battery backup.
- ✧ Intruder alarm is a stimulus generated by one of the system sensors.
 - The response to this stimulus is to compute the room number of the active sensor, set up a call to the police, initiate the voice synthesizer to manage the call, and switch on the audible intruder alarm and building lights in the area.

Frequency and execution time

- ✧ The deadline for detecting a change of state is 0.25 seconds, which means that each sensor has to be checked 4 times per second. If you examine 1 sensor during each process execution, then if there are N sensors of a particular type, you must schedule the process $4N$ times per second to ensure that all sensors are checked within the deadline.
- ✧ If you examine 4 sensors, say, during each process execution, then the execution time is increased to about 4 ms, but you need only run the process N times/second to meet the timing requirement.

Real-time operating systems

Activity

Which of the following are real-time operating systems (RTOS)?

1. Ubuntu Linux
2. Android
3. freeRTOS
4. Windows 11
5. Windows CE

(Do not refer to the Internet to answer this question)



 Apache NuttX

Community Documentation

Apache NuttX

Apache NuttX is a mature, real-time embedded operating system (RTOS).

[Get NuttX](#)



Real-time operating systems

- ✧ Are specialised operating systems which manage the processes in the RTS.
- ✧ Responsible for process management and resource (processor and memory) allocation.
- ✧ May be based on a standard kernel which is used unchanged or modified for a particular application.
- ✧ Do not normally include facilities such as file management.

Operating system components

1. *Real-time clock* - Provides information for process scheduling.
2. *Interrupt handler* - Manages aperiodic requests for service.
3. *Scheduler* - Chooses the next process to be run.
4. *Resource manager* - Allocates memory and processor resources.
5. *Dispatcher* - Starts process execution.

Non-stop system components

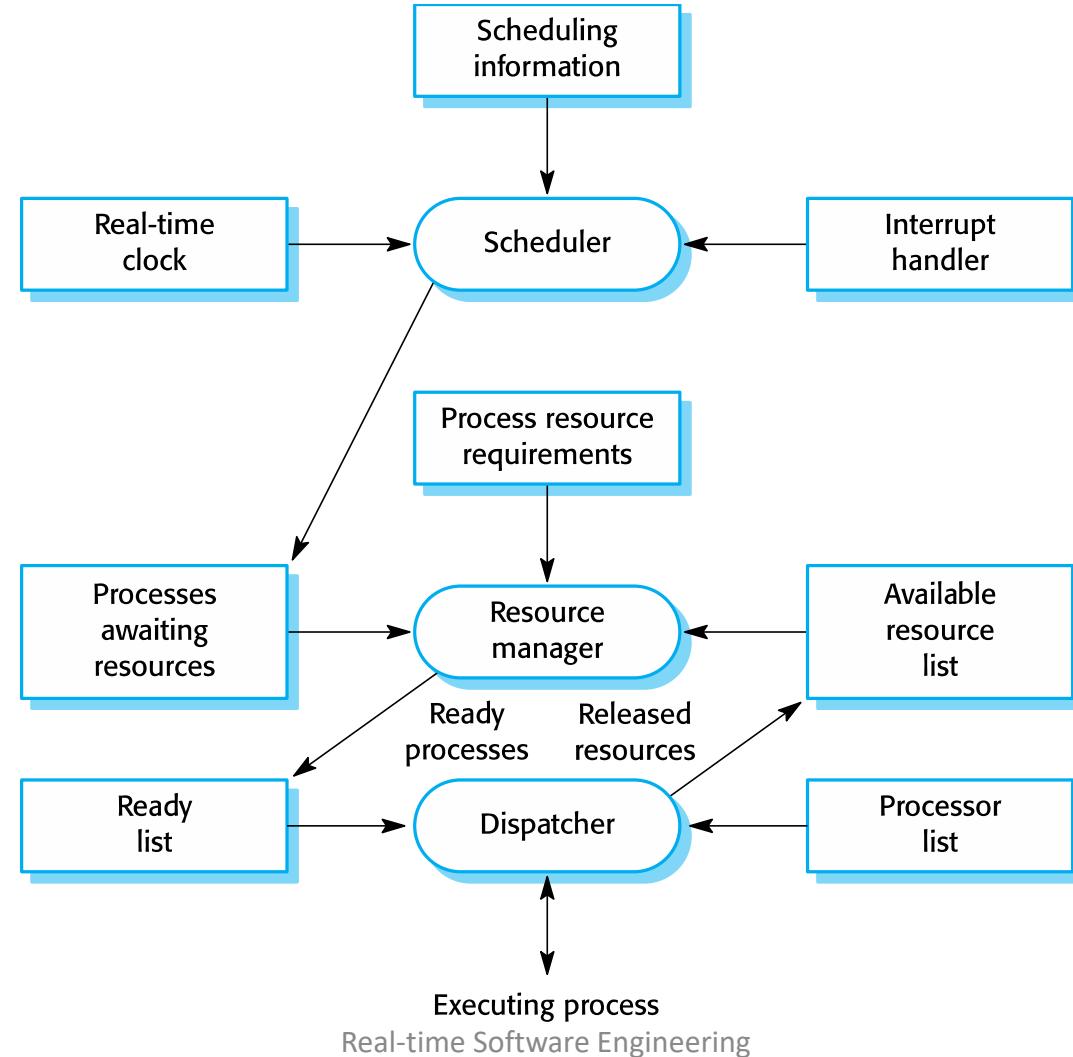
✧ Configuration manager

- Responsible for the dynamic reconfiguration of the system software and hardware. Hardware modules may be replaced and software upgraded without stopping the systems.

✧ Fault manager

- Responsible for detecting software and hardware faults and taking appropriate actions (e.g. switching to backup disks) to ensure that the system continues in operation.

Components of a real-time operating system



Process management

- ✧ Concerned with managing the set of concurrent processes.
- ✧ Periodic processes are executed at pre-specified time intervals.
- ✧ The RTOS uses the real-time clock to determine when to execute a process taking into account:
 - Process period - time between executions.
 - Process deadline - the time by which processing must be complete.

Process management

The processing of some types of stimuli must sometimes take priority.

1. *Interrupt level priority*. Highest priority which is allocated to processes requiring a very fast response.
2. *Clock level priority*. Allocated to periodic processes.

Within these, further levels of priority may be assigned. (E.g. background processes)

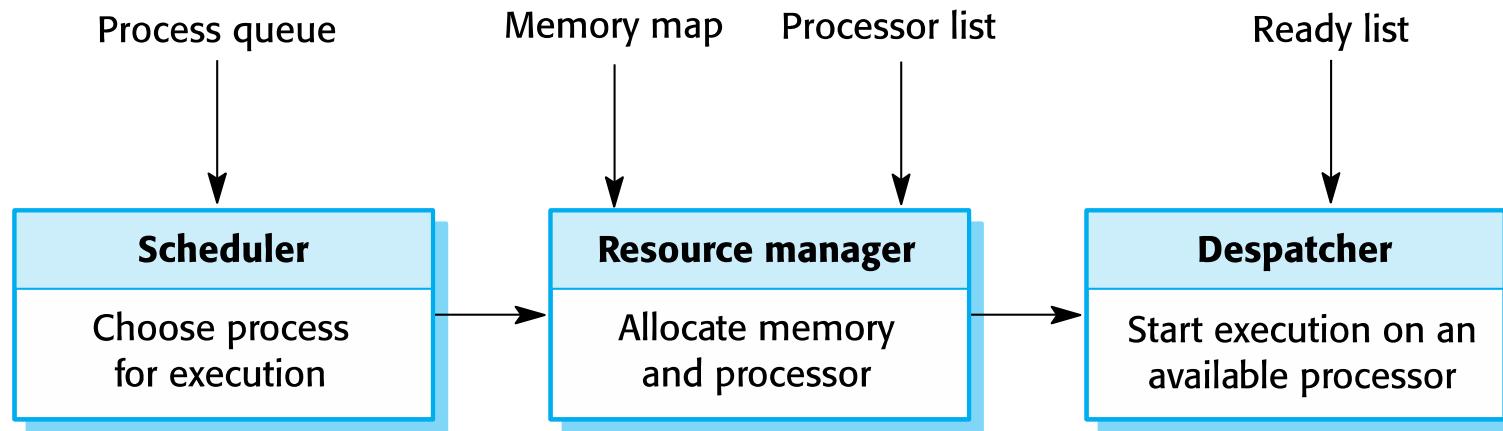
Interrupt servicing

- ✧ Control is transferred automatically to a pre-determined memory location.
- ✧ This location contains an instruction to jump to an interrupt service routine.
- ✧ Further interrupts are disabled, the interrupt serviced and control returned to the interrupted process.
- ✧ Interrupt service routines MUST be short, simple and fast.

Periodic process servicing

- ✧ In most real-time systems, there will be several classes of periodic process, each with different periods (the time between executions), execution times and deadlines (the time by which processing must be completed).
- ✧ The real-time clock ticks periodically and each tick causes an interrupt which schedules the process manager for periodic processes.
- ✧ The process manager selects a process which is ready for execution.

RTOS actions required to start a process



Process switching

- ✧ The scheduler chooses the next process to be executed by the processor. This depends on a scheduling strategy which may take the process priority into account.
- ✧ The resource manager allocates memory and a processor for the process to be executed.
- ✧ The dispatcher takes the process from ready list, loads it onto a processor and starts execution.

Scheduling strategies

✧ Non pre-emptive scheduling

- Once a process has been scheduled for execution, it runs to completion or until it is blocked for some reason (e.g. waiting for I/O).

✧ Pre-emptive scheduling

- The execution of an executing processes may be stopped if a higher priority process requires service.

✧ Scheduling algorithms

- Round-robin;
- Rate monotonic;
- Shortest deadline first.

Key points

- ✧ An embedded software system is part of a hardware/software system that reacts to events in its environment. The software is ‘embedded’ in the hardware. Embedded systems are normally real-time systems.
- ✧ A real-time system is a software system that must respond to events in real time. System correctness does not just depend on the results it produces, but also on the time when these results are produced.
- ✧ Real-time systems are usually implemented as a set of communicating processes that react to stimuli to produce responses.
- ✧ State models are an important design representation for embedded real-time systems. They are used to show how the system reacts to its environment as events trigger changes of state in the system.

Key points

- ✧ There are several standard patterns that can be observed in different types of embedded system. These include a pattern for monitoring the system's environment for adverse events, a pattern for actuator control and a data-processing pattern.
- ✧ Designers of real-time systems have to do a timing analysis, which is driven by the deadlines for processing and responding to stimuli. They have to decide how often each process in the system should run and the expected and worst-case execution time for processes.
- ✧ A real-time operating system is responsible for process and resource management. It always includes a scheduler, which is the component responsible for deciding which process should be scheduled for execution.