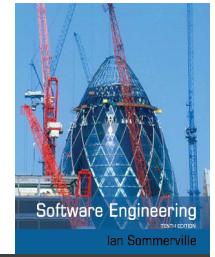




Chapter 1- Introduction

Topics covered



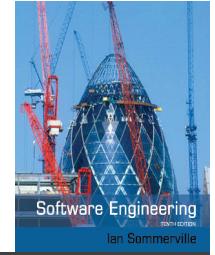
- ✧ Professional software development
 - What is meant by software engineering.
- ✧ Software engineering ethics
 - A brief introduction to ethical issues that affect software engineering.
- ✧ Case studies
 - An introduction to three examples that are used in later chapters in the book.

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 project failure



✧ *Increasing system complexity*

- As new software engineering techniques help us to build larger, more complex systems, the demands change. Systems have to be built and delivered more quickly; larger, even more complex systems are required; systems have to have new capabilities that were previously thought to be impossible.

✧ *Failure to use software engineering methods*

- It is fairly easy to write computer programs without using software engineering methods and techniques. Many companies have drifted into software development as their products and services have evolved. They do not use software engineering methods in their everyday work. Consequently, their software is often more expensive and less reliable than it should be.



Professional software development

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.

Software products



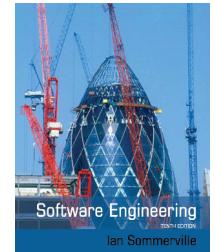
✧ Generic products

- 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

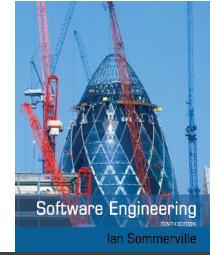
- The specification of what the software should do is owned by the customer for the software and they make decisions on software changes that are required.

Essential attributes of good software



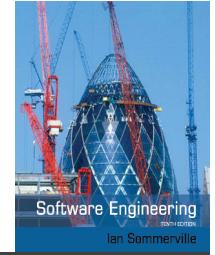
Product characteristic	Description
Maintainability	Software should be written in such a way so that it can evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable requirement of a changing business environment.
Dependability and security	Software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



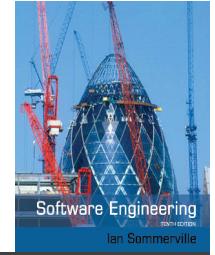
✧ Heterogeneity

- 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.

General issues that affect software



✧ Security and trust

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

✧ Scale

- Software has to be developed across a very wide range of scales, from very small embedded systems in portable or wearable devices through to Internet-scale, cloud-based systems that serve a global community.

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

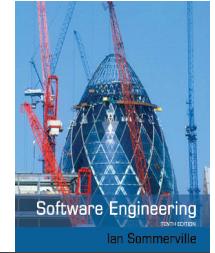
- These are application systems that run on a local computer, such as a PC. They include all necessary functionality and do not need to be connected to a network.

✧ Interactive transaction-based applications

- Applications that execute on a remote computer and are accessed by users from their own PCs or terminals. These include web applications such as e-commerce applications.

✧ Embedded control systems

- These are software control systems that control and manage hardware devices. Numerically, there are probably more embedded systems than any other type of system.



Application types

✧ Batch processing systems

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

✧ Entertainment systems

- These are systems that are primarily for personal use and which are intended to entertain the user.

✧ Systems for modeling and simulation

- These are systems that are developed by scientists and engineers to model physical processes or situations, which include many, separate, interacting objects.



Application types

✧ Data collection systems

- These are systems that collect data from their environment using a set of sensors and send that data to other systems for processing.

✧ Systems of systems

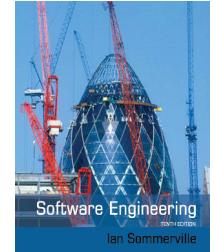
- These are systems that are composed of a number of other software systems.

Software engineering fundamentals



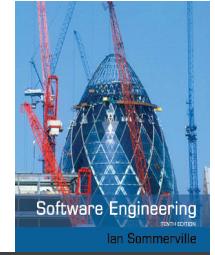
- ✧ Some fundamental principles apply to all types of software system, irrespective of the development techniques used:
 - Systems should be developed using a managed and understood development process. Of course, different processes are used for different types of software.
 - Dependability and performance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

Internet software engineering



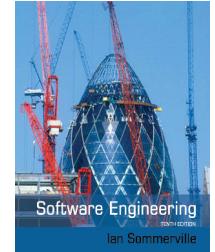
- ✧ The Web is now a platform for running application and organizations are increasingly developing web-based systems rather than local systems.
- ✧ Web services (discussed in Chapter 19) allow application functionality to be accessed over the web.
- ✧ Cloud computing is an approach to the provision of computer services where applications run remotely on the ‘cloud’.
 - Users do not buy software but pay according to use.

Web-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 apply to web-based software in the same way that they apply to other types of software system.

Web software engineering



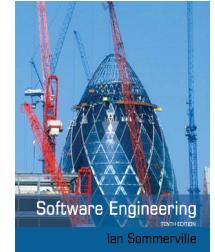
❖ Software reuse

- 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.

❖ Incremental and agile development

- 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.

Web software engineering



✧ Service-oriented systems

- Software may be implemented using service-oriented software engineering, where the software components are stand-alone web services.

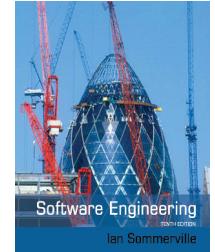
✧ Rich interfaces

- Interface development technologies such as AJAX and HTML5 have emerged that support the creation of rich interfaces within a web browser.



Software engineering ethics

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



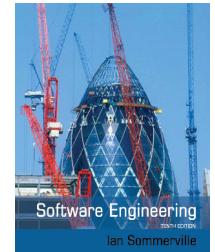
✧ Confidentiality

- 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

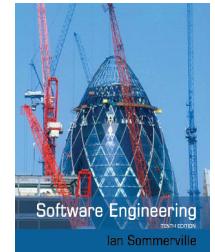
ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices

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

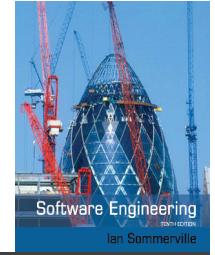


1. PUBLIC - Software engineers shall act consistently with the public interest.
2. CLIENT AND EMPLOYER - Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.
3. PRODUCT - Software engineers shall ensure that their products 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.



Case studies

Ethical dilemmas



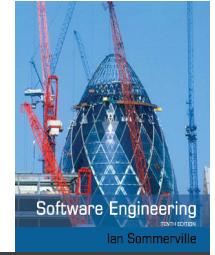
- ✧ Disagreement in principle with the policies of senior management.
- ✧ Your employer acts in an unethical way and releases a safety-critical system without finishing the testing of the system.
- ✧ Participation in the development of military weapons systems or nuclear systems.

Case studies

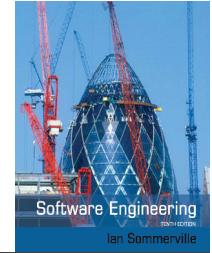


- ✧ A personal insulin pump
 - An embedded system in an insulin pump used by diabetics to maintain blood glucose control.
- ✧ A mental health case patient management system
 - Mentcare. A system used to maintain records of people receiving care for mental health problems.
- ✧ A wilderness weather station
 - A data collection system that collects data about weather conditions in remote areas.
- ✧ iLearn: a digital learning environment
 - A system to support learning in schools

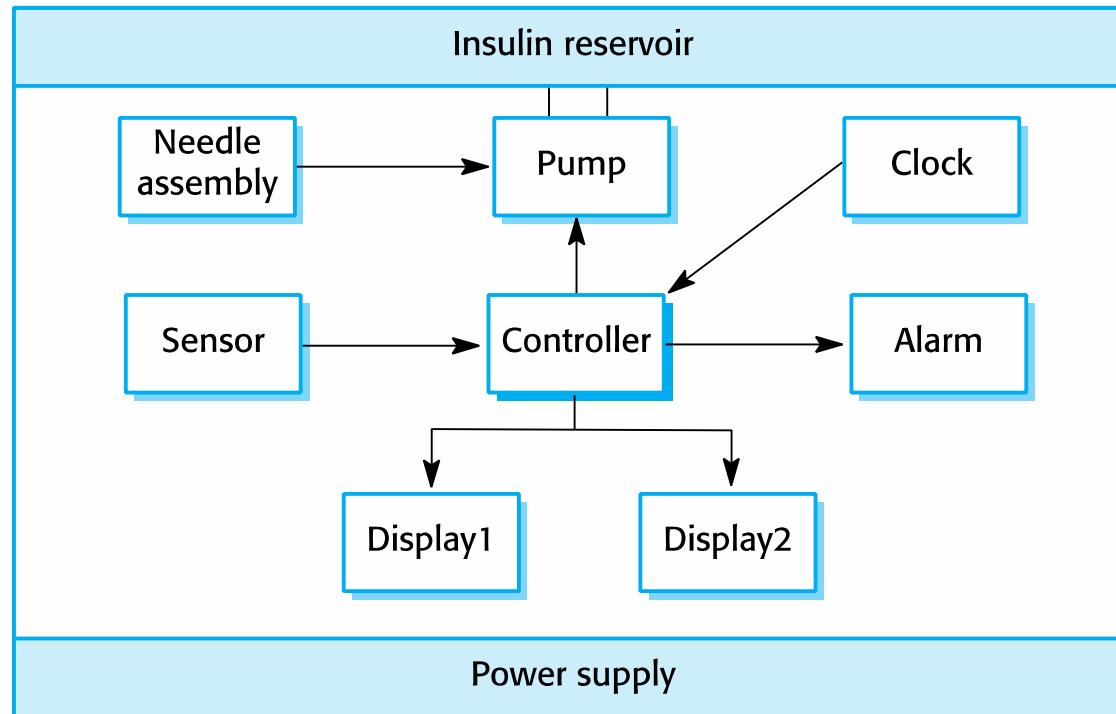
Insulin pump control system



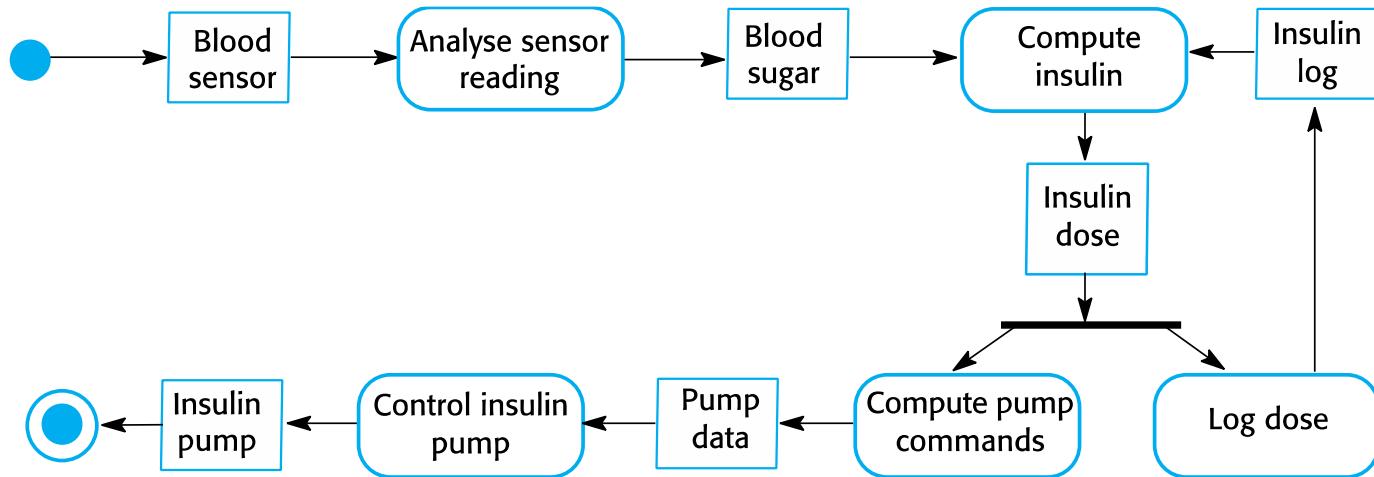
- ✧ Collects data from a blood sugar sensor and calculates the amount of insulin required to be injected.
- ✧ Calculation based on the rate of change of blood sugar levels.
- ✧ Sends signals to a micro-pump to deliver the correct dose of insulin.
- ✧ Safety-critical system as low blood sugars can lead 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

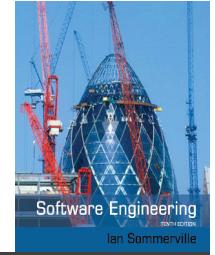


Essential high-level requirements



- ✧ 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.

Mentcare: A patient information system for mental health care



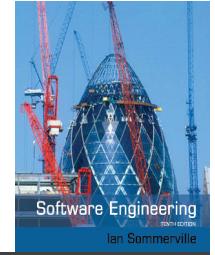
- ✧ A patient information system to support mental health care is a medical information system that maintains information about patients suffering from mental health problems and the treatments that they have received.
- ✧ Most mental health patients do not require dedicated hospital treatment but need 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.

Mentcare

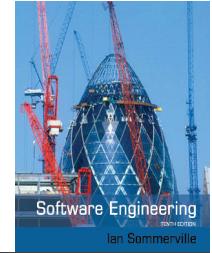


- ✧ Mentcare is an information system that is intended for use in clinics.
- ✧ It makes use of a centralized database of patient information but has also been designed to run on a PC, so that it may be accessed and used from sites that do not have secure network connectivity.
- ✧ When the local systems 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.

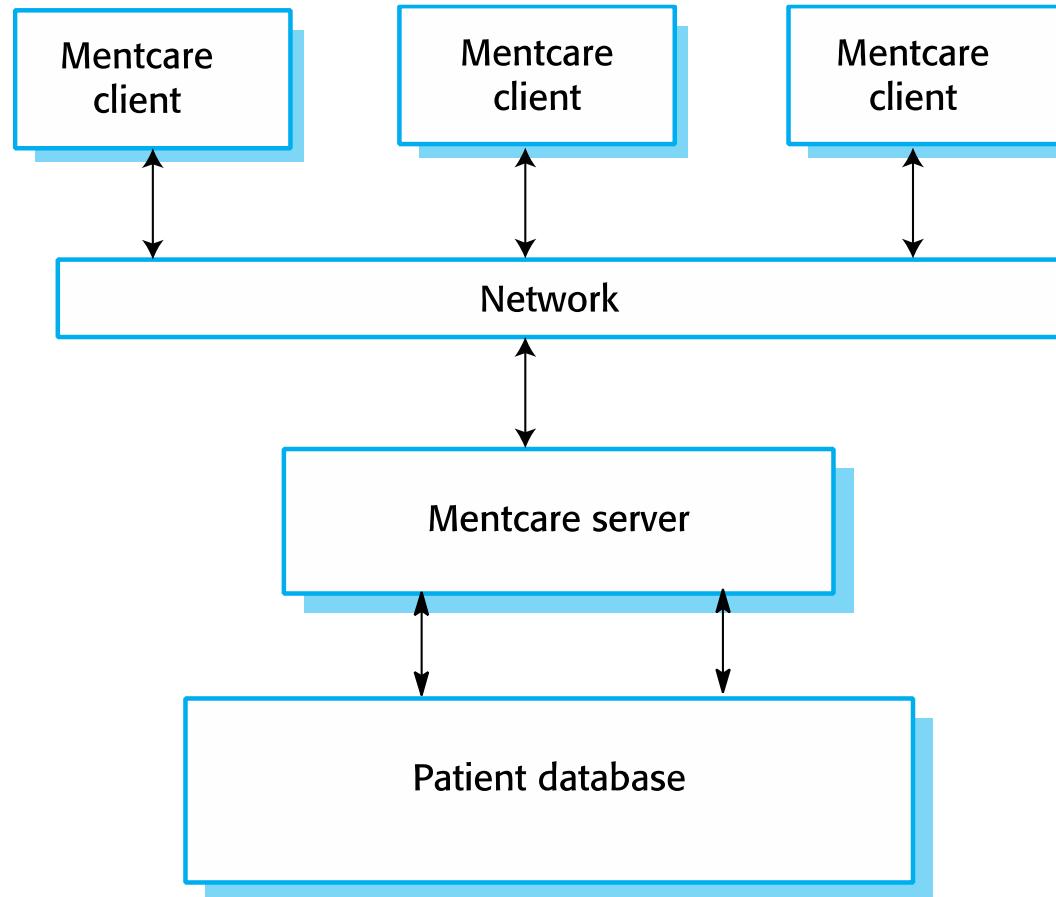
Mentcare goals

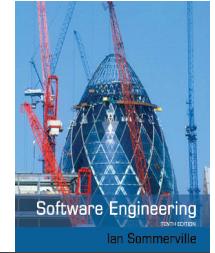


- ✧ To generate management information that allows health service managers to assess performance against local and government targets.
- ✧ To provide medical staff with timely information to support the treatment of patients.



The organization of the Mentcare system





Key features of the Mentcare system

✧ 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.

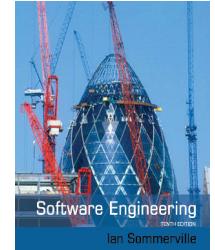
✧ Patient monitoring

- 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.

Mentcare system 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

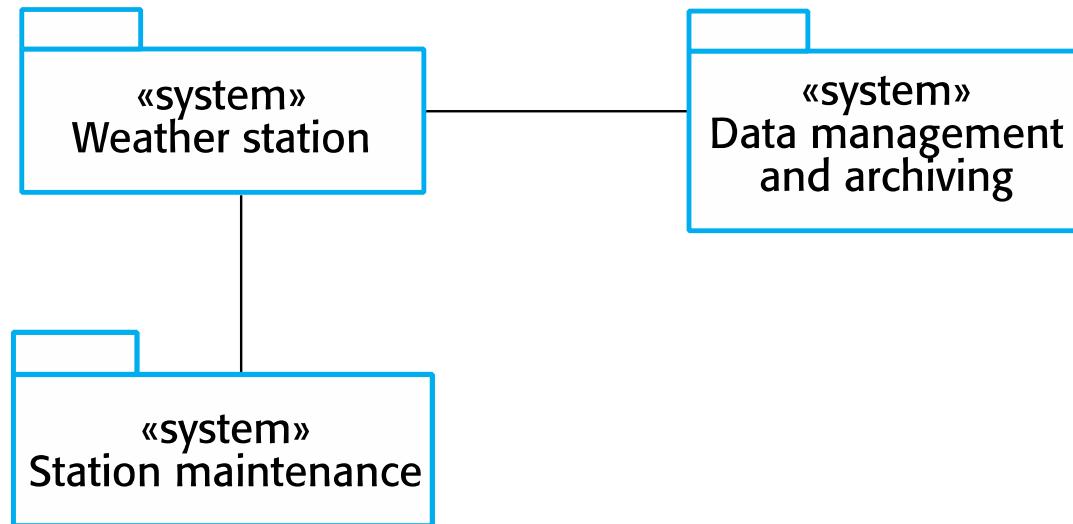
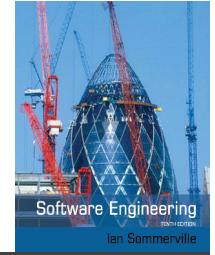
- Some mental illnesses cause patients to become suicidal or a danger to other people. Wherever possible, the system should warn medical staff about potentially suicidal or dangerous patients.
- The system must be available when needed otherwise safety may be compromised and it may be impossible to prescribe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

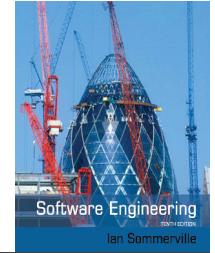
Wilderness weather station



- ✧ 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





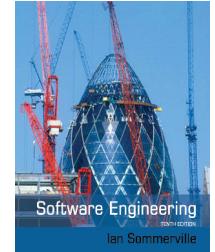
Weather information system

Software Engineering
Ian Sommerville

10th Edition

- ✧ 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



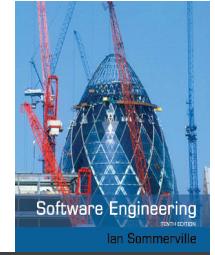
- ✧ Monitor the instruments, power and communication hardware and report faults to the management system.
- ✧ Manage the system power, ensuring that batteries are charged whenever the environmental conditions permit but also that generators are shut down in potentially damaging weather conditions, such 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.

iLearn: A digital learning environment



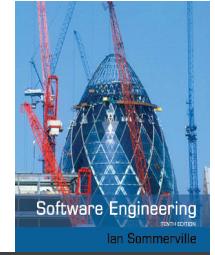
- ✧ A digital learning environment is a framework in which a set of general-purpose and specially designed tools for learning may be embedded plus a set of applications that are geared to the needs of the learners using the system.
- ✧ The tools included in each version of the environment are chosen by teachers and learners to suit their specific needs.
 - These can be general applications such as spreadsheets, learning management applications such as a Virtual Learning Environment (VLE) to manage homework submission and assessment, games and simulations.

Service-oriented systems



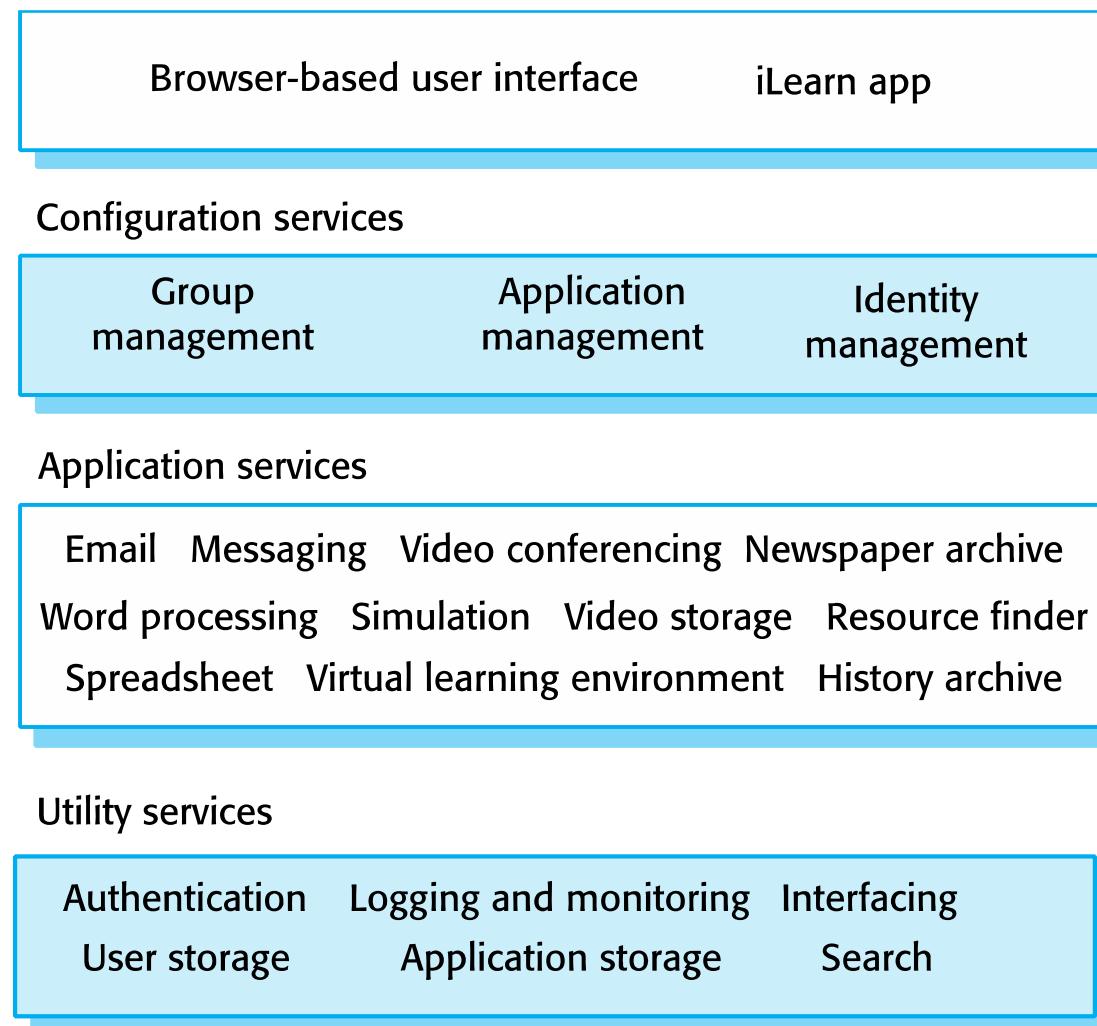
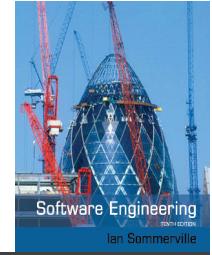
- ✧ The system is a service-oriented system with all system components considered to be a replaceable service.
- ✧ This allows the system to be updated incrementally as new services become available.
- ✧ It also makes it possible to rapidly configure the system to create versions of the environment for different groups such as very young children who cannot read, senior students, etc.

iLearn services

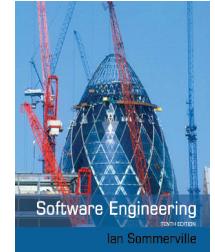


- ✧ *Utility services* that provide basic application-independent functionality and which may be used by other services in the system.
- ✧ *Application services* that provide specific applications such as email, conferencing, photo sharing etc. and access to specific educational content such as scientific films or historical resources.
- ✧ *Configuration services* that are used to adapt the environment with a specific set of application services and do define how services are shared between students, teachers and their parents.

iLearn architecture

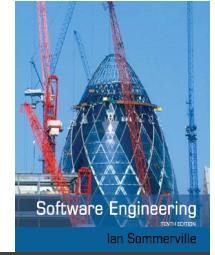


iLearn service integration

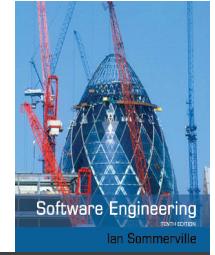


- ✧ *Integrated services* are services which offer an API (application programming interface) and which can be accessed by other services through that API. Direct service-to-service communication is therefore possible.
- ✧ *Independent services* are services which are simply accessed through a browser interface and which operate independently of other services. Information can only be shared with other services through explicit user actions such as copy and paste; re-authentication may be required for each independent service.

Key points



- ✧ 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.



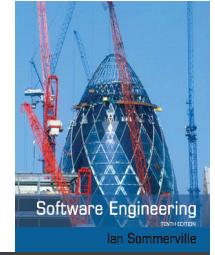
Key points

- ✧ 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.
- ✧ 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.



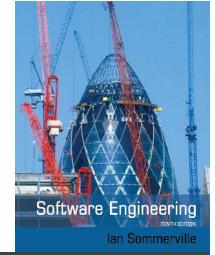
Chapter 2 – Software Processes

Topics covered



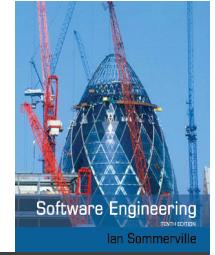
- ✧ Software process models
- ✧ Process activities
- ✧ Coping with change
- ✧ Process improvement

The software process



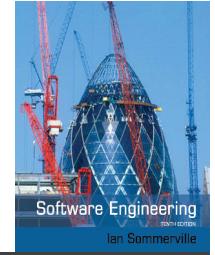
- ✧ A structured set of activities required to develop a software system.
- ✧ Many different software processes but all involve:
 - Specification – defining what the system should do;
 - Design and implementation – defining the organization of the system and implementing the system;
 - Validation – checking 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

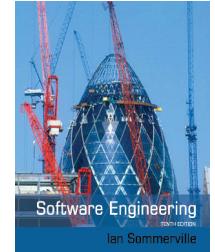


- ✧ Plan-driven processes are processes where all of the process activities are planned in advance and progress is measured against this plan.
- ✧ In agile processes, planning is incremental and it is easier to change the process to reflect changing customer requirements.
- ✧ In practice, most practical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



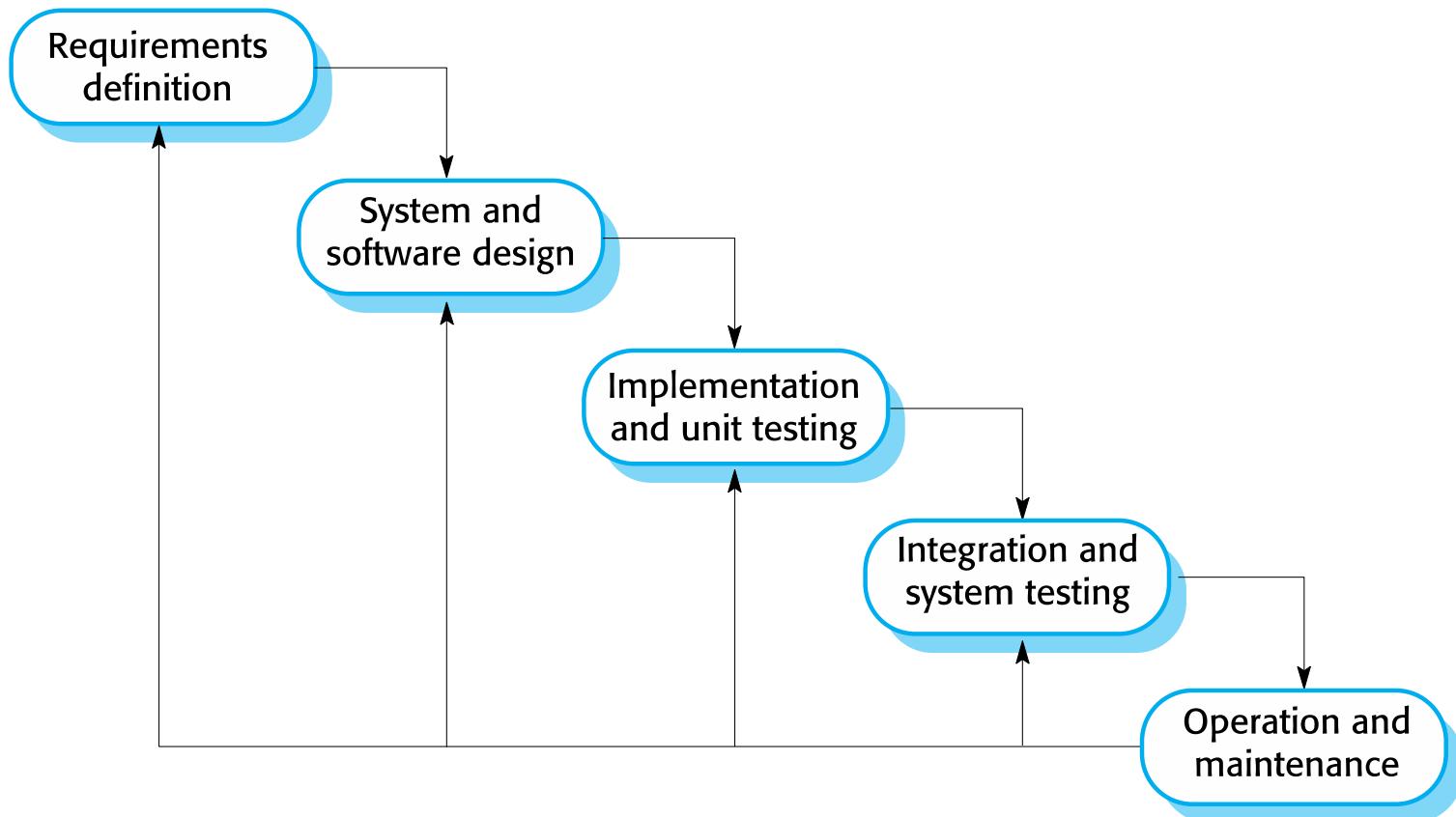
Software process models

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.
- ✧ Integration and configuration
 - The system is assembled from existing configurable 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.

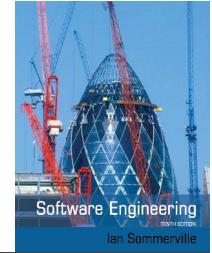
The waterfall model



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.



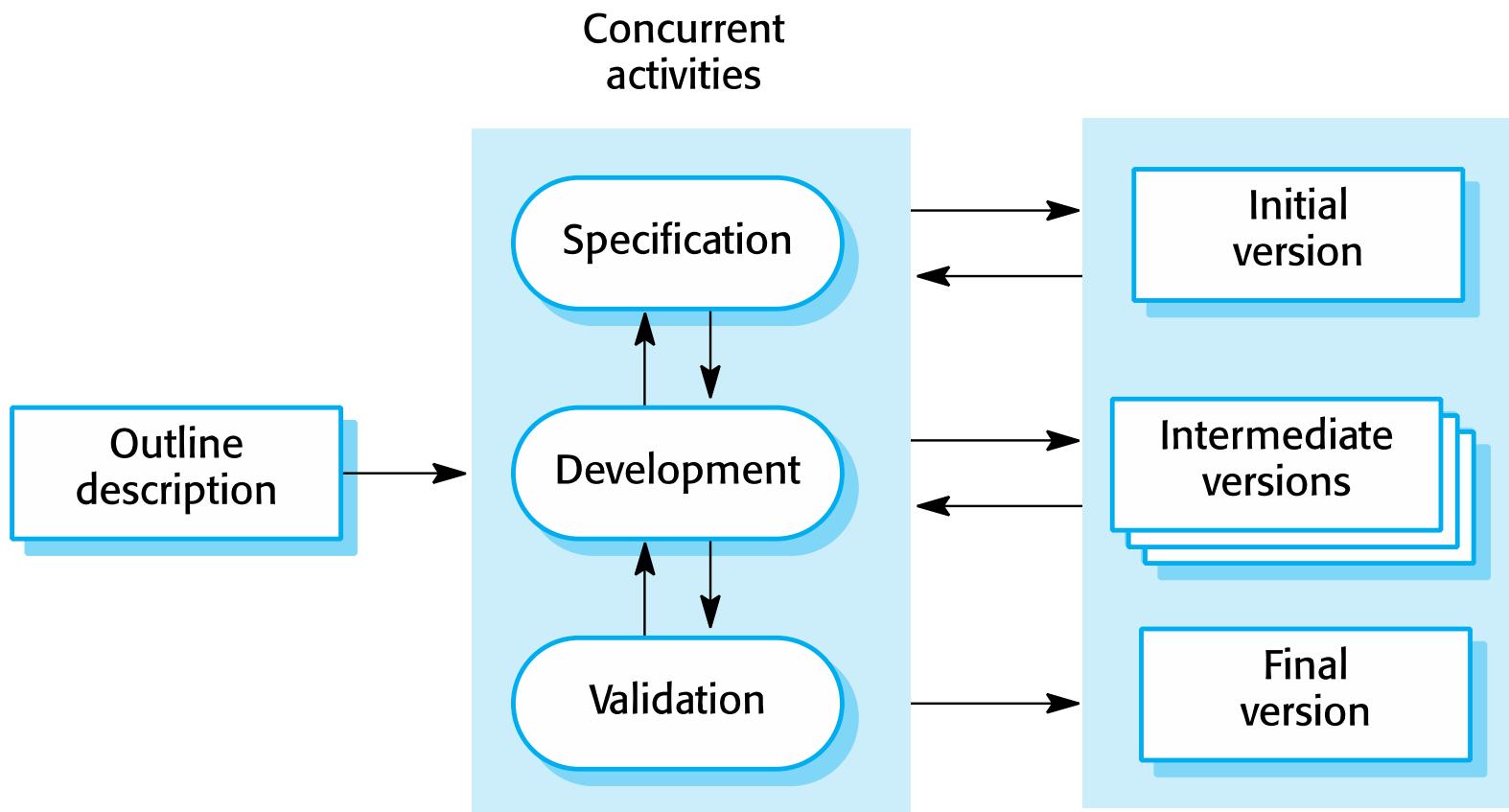
Waterfall model problems

- ✧ 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



Software Engineering
Ian Sommerville



Incremental development benefits



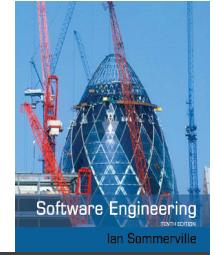
- ✧ The cost of accommodating changing customer requirements is reduced.
 - The amount of analysis and documentation that has to be redone is much less than is required with the waterfall model.
- ✧ It is easier to get customer feedback on the development work that has been done.
 - Customers can comment 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



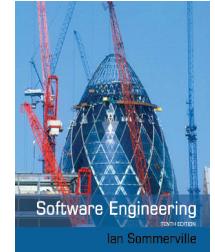
- ✧ The process is not visible.
 - Managers need regular deliverables to measure progress. If systems are developed quickly, it is not cost-effective to produce documents that reflect every version of the system.
- ✧ System structure tends to degrade as new increments are added.
 - Unless time and money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

Integration and configuration



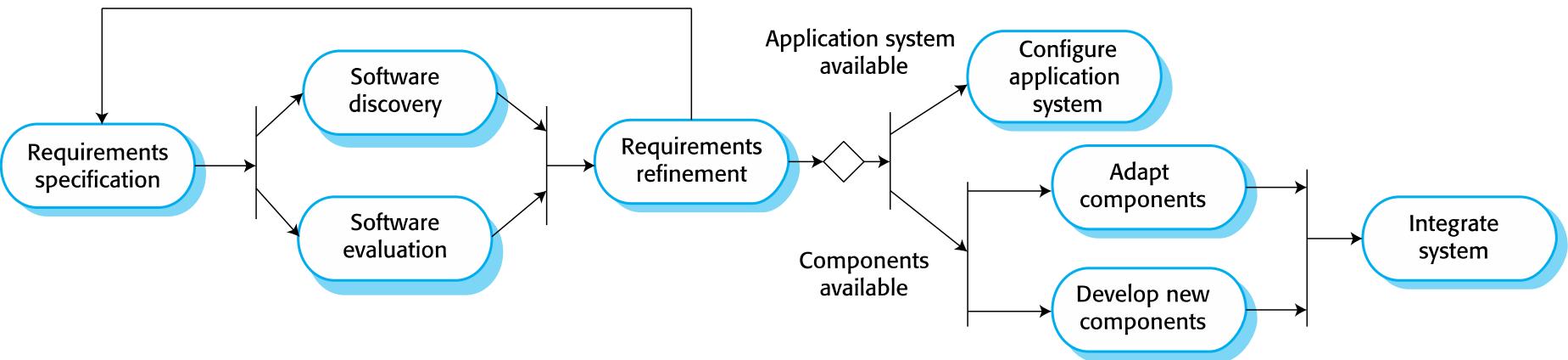
- ✧ Based on software reuse where systems are integrated from existing components or application systems (sometimes called COTS -Commercial-off-the-shelf) systems.
- ✧ Reused elements may be configured to adapt their behaviour and functionality to a user's requirements
- ✧ Reuse is now the standard approach for building many types of business system
 - Reuse covered in more depth in Chapter 15.

Types of reusable software



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

Reuse-oriented software engineering

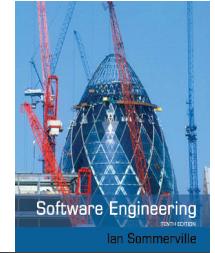


Key process stages



- ✧ Requirements specification
- ✧ Software discovery and evaluation
- ✧ Requirements refinement
- ✧ Application system configuration
- ✧ Component adaptation and integration

Advantages and disadvantages

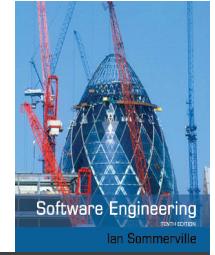


- ✧ Reduced costs and risks as less software is developed from scratch
- ✧ Faster delivery and deployment of system
- ✧ But requirements compromises are inevitable so system may not meet real needs of users
- ✧ Loss of control over evolution of reused system elements



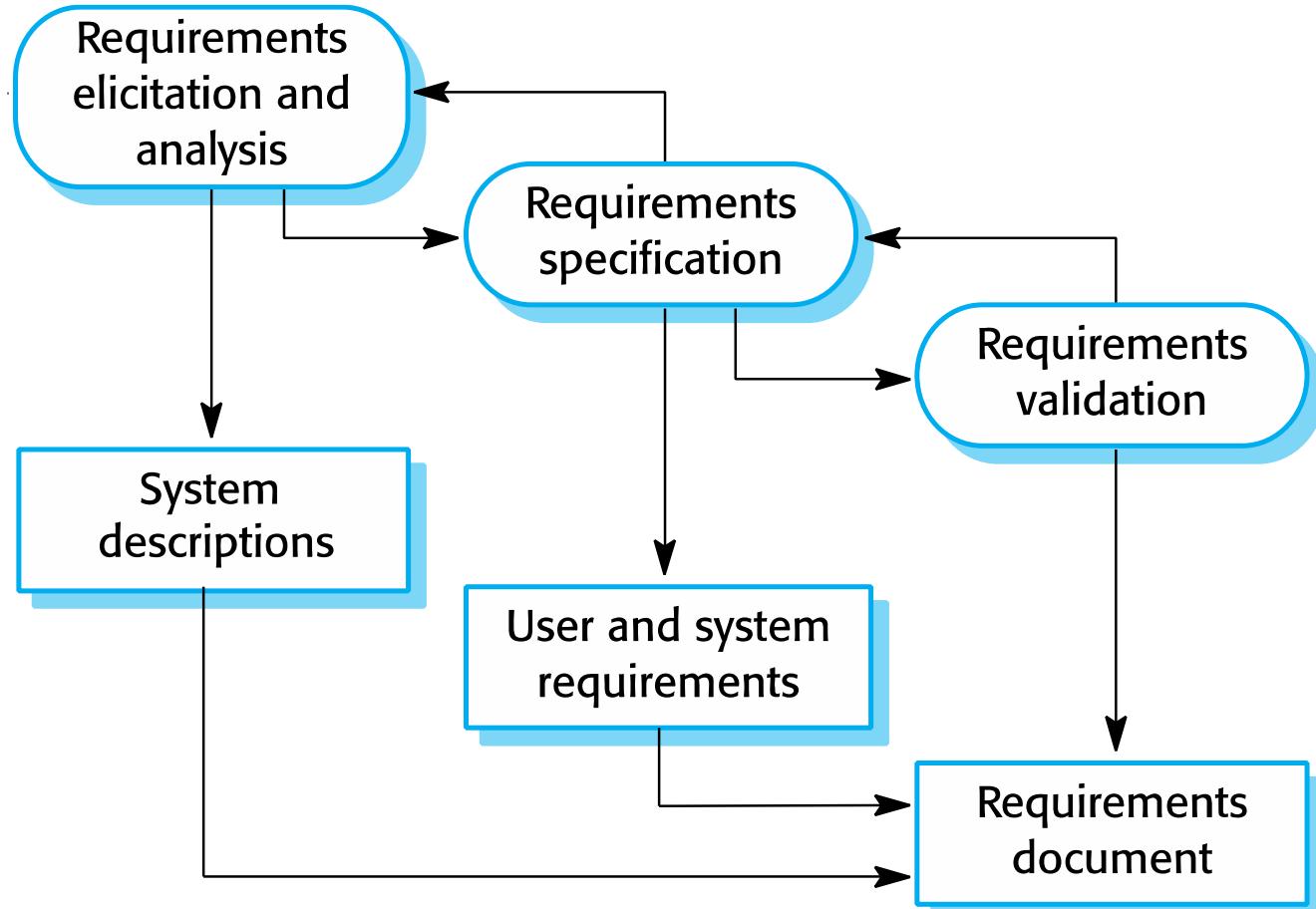
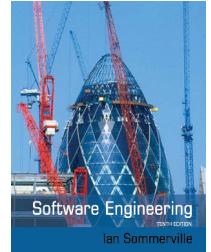
Process activities

Process activities

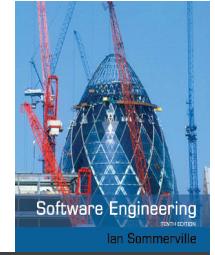


- ✧ Real software processes are inter-leaved sequences of technical, collaborative and managerial activities with the overall goal of specifying, designing, implementing and testing a software system.
- ✧ The four basic process activities of specification, development, validation and evolution are organized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

The requirements engineering process



Software specification



- ✧ The process of establishing what services are required and the constraints on the system's operation and development.
- ✧ Requirements engineering process
 - 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

Software design and implementation



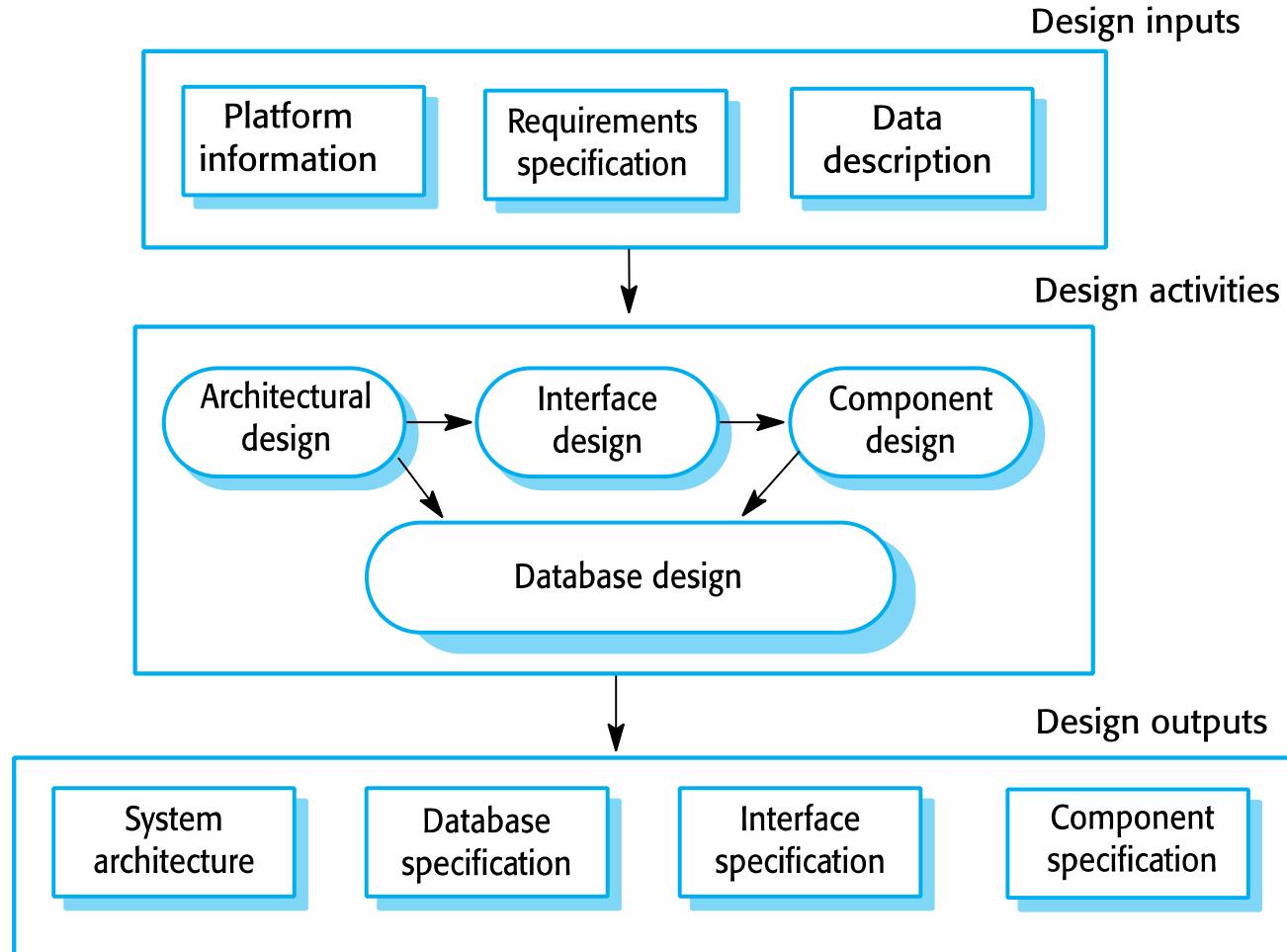
- ✧ The process of converting the system specification into an executable system.
- ✧ Software design
 - Design a software structure that realises the specification;
- ✧ Implementation
 - Translate this structure into an executable program;
- ✧ The activities of design and implementation are closely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

A general model of the design process



Software Engineering

Ian Sommerville



Design activities



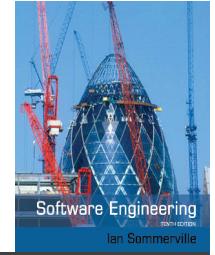
- ✧ *Architectural design*, where you identify the overall structure of the system, the principal components (subsystems or modules), their relationships and how they are distributed.
- ✧ *Database design*, where you design the system data structures and how these are to be represented in a database.
- ✧ *Interface design*, where you define the interfaces between system components.
- ✧ *Component selection and design*, where you search for reusable components. If unavailable, you design how it will operate.

System implementation



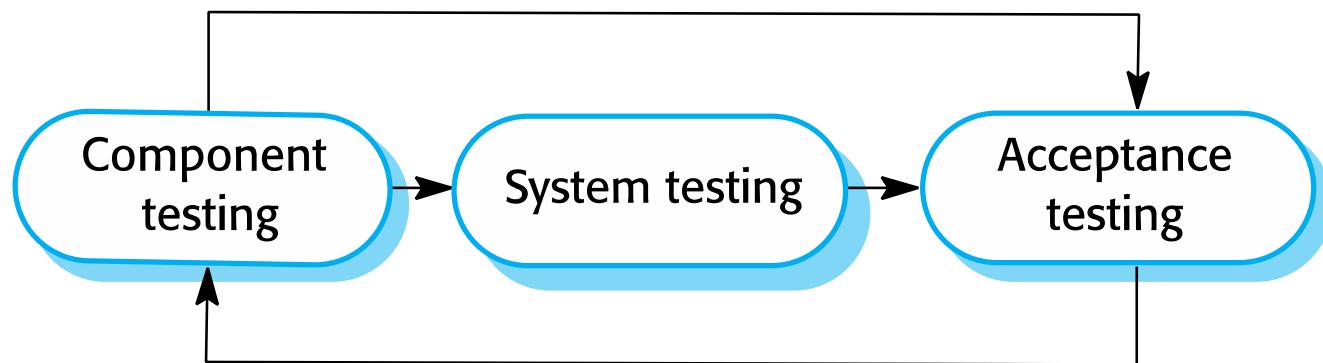
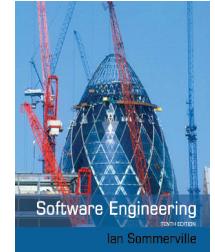
- ✧ The software is implemented either by developing a program or programs or by configuring an application system.
- ✧ Design and implementation are interleaved activities for most types of software system.
- ✧ Programming is an individual activity with no standard process.
- ✧ Debugging is the activity of finding program faults and correcting these faults.

Software validation

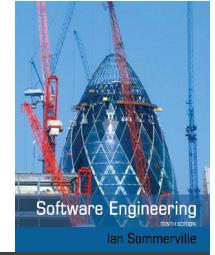


- ✧ Verification and validation (V & V) is intended to show that a system conforms to its specification and meets the requirements of the system customer.
- ✧ Involves checking and review processes and system testing.
- ✧ System testing involves executing the system with test cases that are derived from 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



✧ Component testing

- Individual components are tested independently;
- Components may be functions or objects or coherent groupings of these entities.

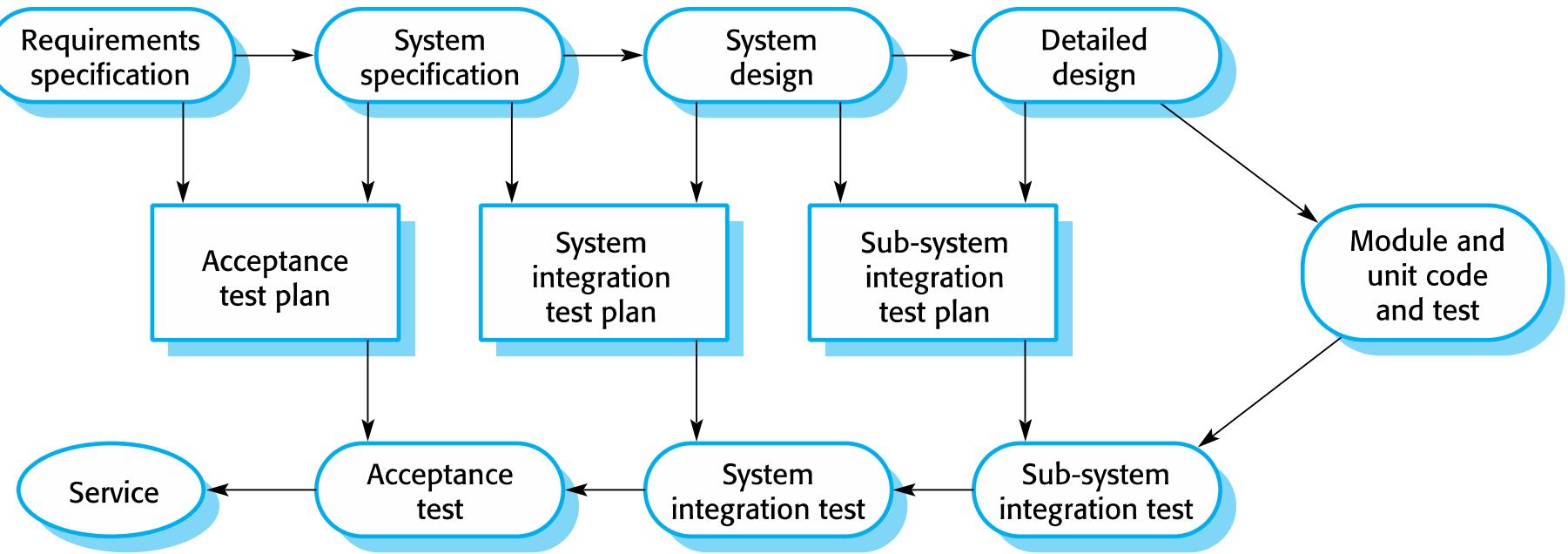
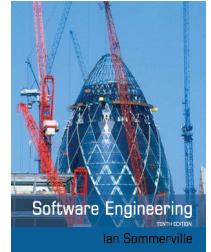
✧ System testing

- Testing of the system as a whole. Testing of emergent properties is particularly important.

✧ Customer testing

- Testing with customer data to check that the system meets the customer's needs.

Testing phases in a plan-driven software process (V-model)

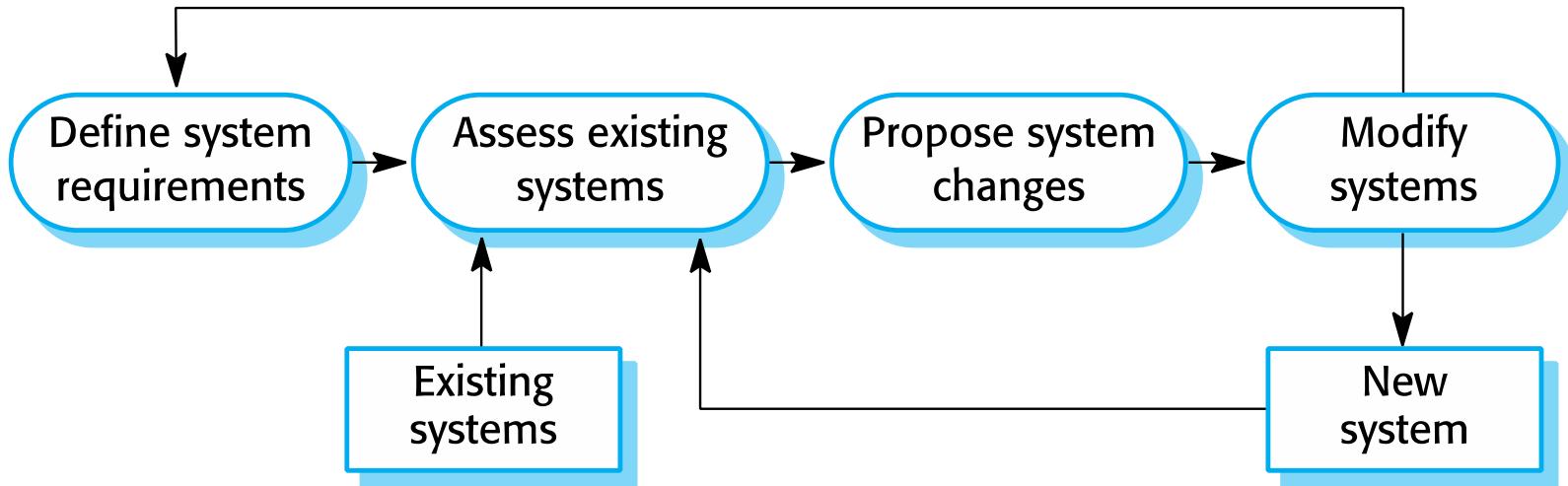


Software evolution



- ✧ 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





Software Engineering

Ian Sommerville

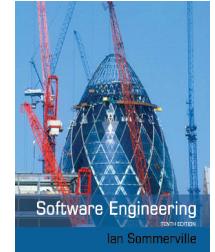
Coping with change

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



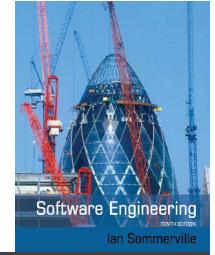
- ✧ Change anticipation, where the software process includes activities that can anticipate possible changes before significant rework is required.
 - For example, a prototype system may be developed to show some key features of the system to customers.
- ✧ Change tolerance, where the process is designed 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.

Coping with changing requirements



- ✧ System prototyping, where a version of the system or part of the system is developed quickly to check the customer's requirements and the feasibility of design decisions. This approach supports change anticipation.
- ✧ Incremental delivery, where system increments are delivered to the customer for comment and experimentation. This supports both change avoidance and change tolerance.

Software prototyping



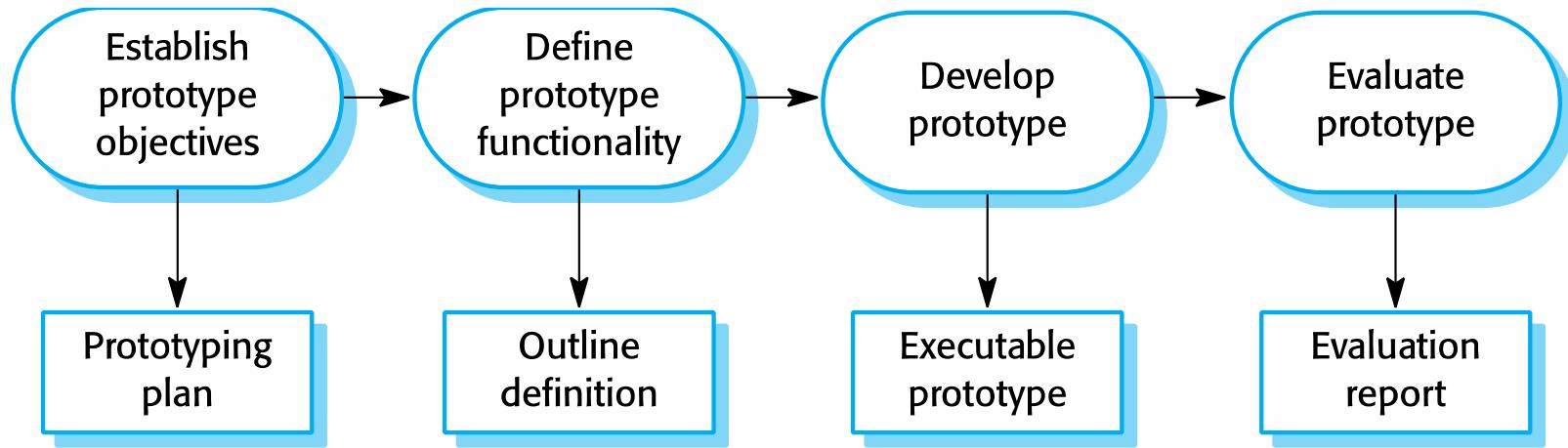
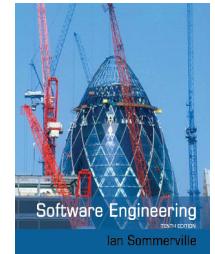
- ✧ A prototype is an initial version of a system used to demonstrate concepts and try out design options.
- ✧ A prototype can be used in:
 - The requirements engineering process to help with requirements elicitation and validation;
 - In design processes to explore options and develop a UI design;
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

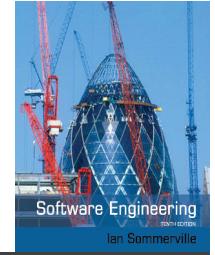


Prototype development



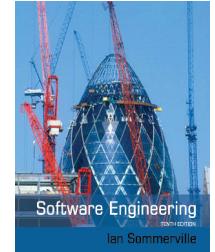
- ✧ May be based on rapid prototyping languages or tools
- ✧ May involve leaving out functionality
 - Prototype should focus on areas of the product that are not well-understood;
 - Error checking and recovery may not be included in the prototype;
 - Focus on functional rather than non-functional requirements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



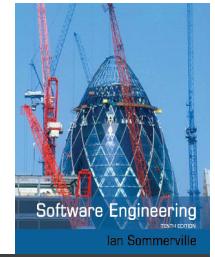
✧ Incremental development

- 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.

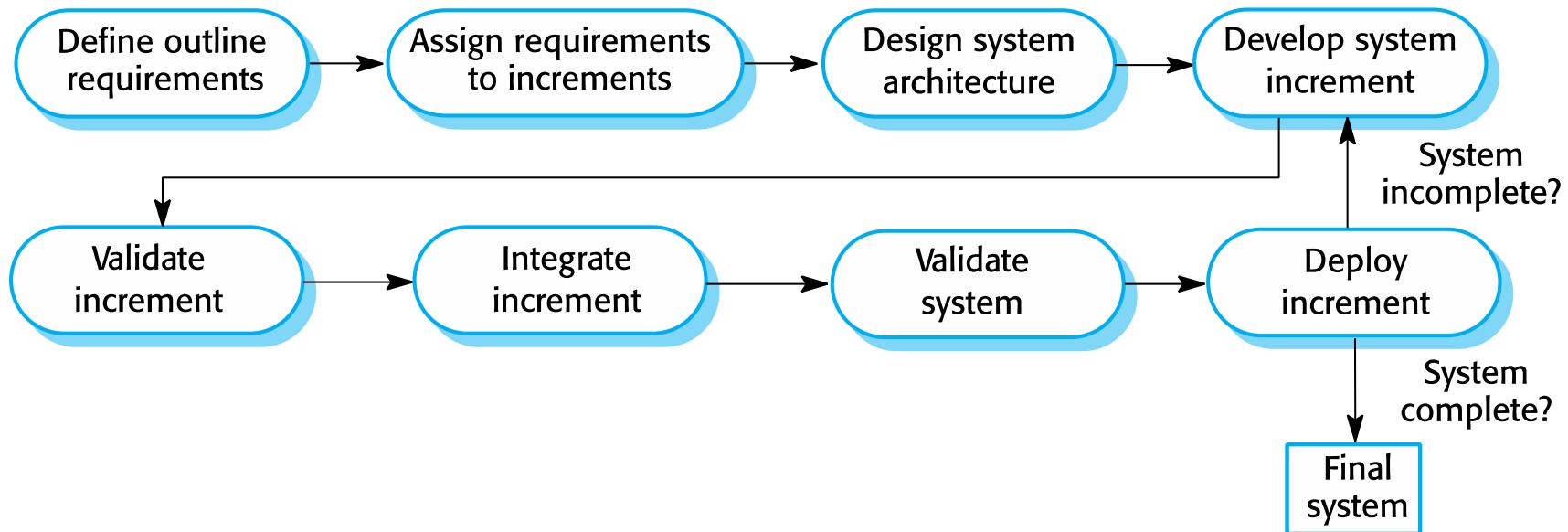
✧ Incremental delivery

- 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.

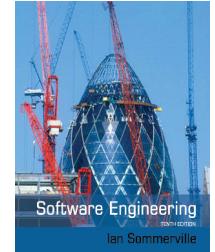
Incremental delivery



Software Engineering
Ian Sommerville



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.



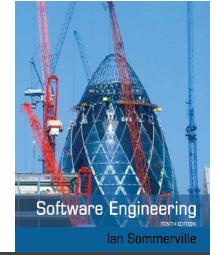
Process improvement

Process improvement



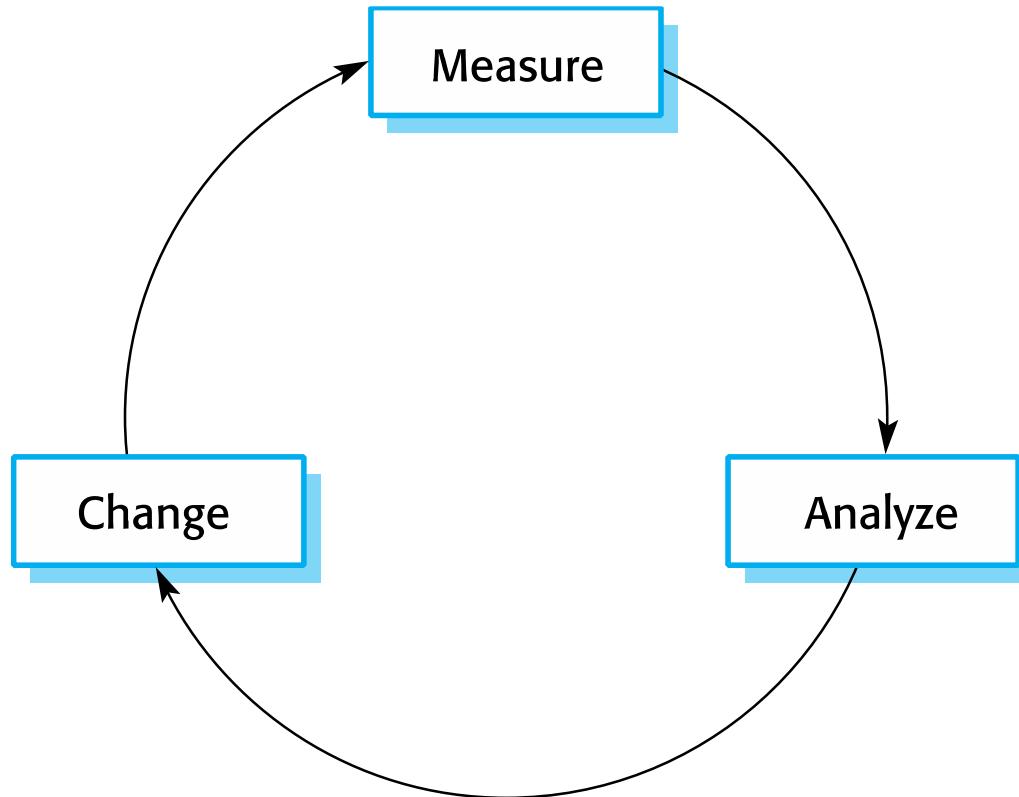
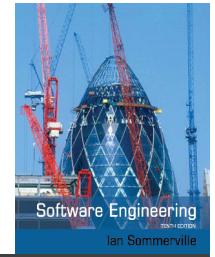
- ✧ Many software companies have turned to software process improvement as a way of enhancing the quality of their software, reducing costs or accelerating their development processes.
- ✧ Process improvement means understanding existing processes and changing these processes to increase product quality and/or reduce costs and development time.

Approaches to improvement



- ✧ The process maturity approach, which focuses on improving process and project management and introducing good software engineering practice.
 - The level of process maturity reflects the extent to which good technical and management practice has been adopted in organizational software development processes.
- ✧ The agile approach, which focuses on iterative development and the reduction of overheads in the software process.
 - The primary characteristics of agile methods are rapid delivery of functionality and responsiveness to changing customer requirements.

The process improvement cycle



Process improvement activities



✧ *Process measurement*

- You measure one or more attributes of the software process or product. These measurements forms a baseline that helps you decide if process improvements have been effective.

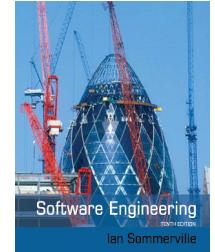
✧ *Process analysis*

- The current process is assessed, and process weaknesses and bottlenecks are identified. Process models (sometimes called process maps) that describe the process may be developed.

✧ *Process change*

- Process changes are proposed to address some of the identified process weaknesses. These are introduced and the cycle resumes to collect data about the effectiveness of the changes.

Process measurement



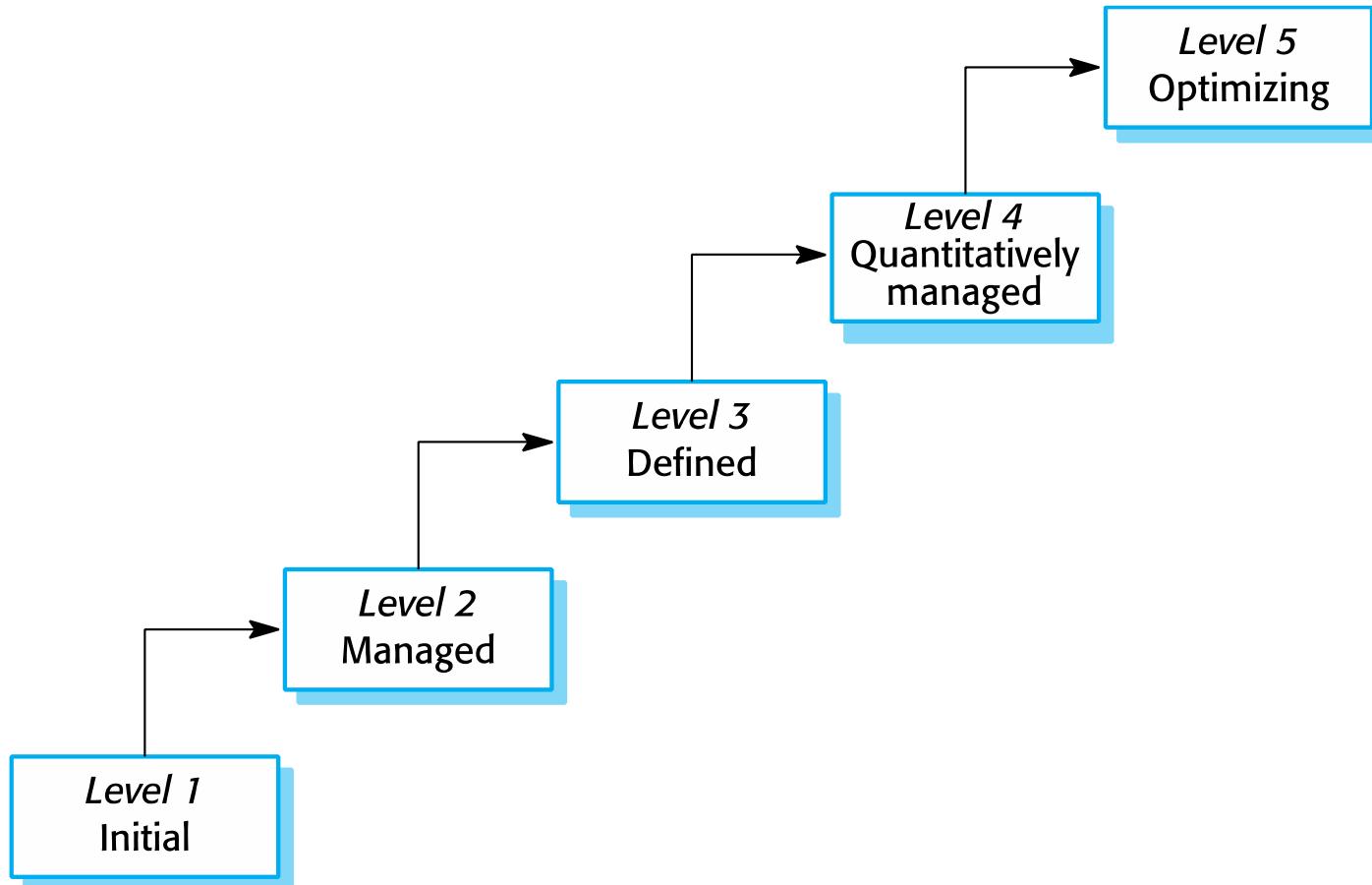
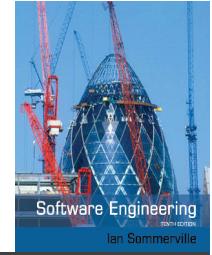
- ✧ Wherever possible, quantitative process data should be collected
 - However, where organisations do not have clearly defined process standards this is very difficult as you don't know what to measure. A process may have to be defined before any measurement is possible.
- ✧ Process measurements should be used to assess process improvements
 - But this does not mean that measurements should drive the improvements. The improvement driver should be the organizational objectives.

Process metrics

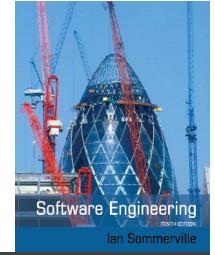


- ✧ Time taken for process activities to be completed
 - E.g. Calendar time or effort to complete an activity or process.
- ✧ Resources required for processes or activities
 - E.g. Total effort in person-days.
- ✧ Number of occurrences of a particular event
 - E.g. Number of defects discovered.

Capability maturity levels

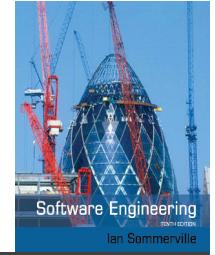


The SEI capability maturity model



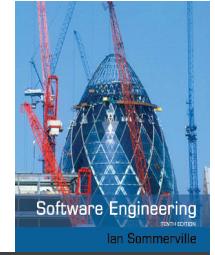
- ✧ Initial
 - Essentially uncontrolled
- ✧ Repeatable
 - Product management procedures defined and used
- ✧ Defined
 - Process management procedures and strategies defined and used
- ✧ Managed
 - Quality management strategies defined and used
- ✧ Optimising
 - Process improvement strategies defined and used

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.
- ✧ Requirements engineering is the process of developing a software specification.

Key points



- ✧ 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.
- ✧ Processes should include activities such as prototyping and incremental delivery to cope with change.



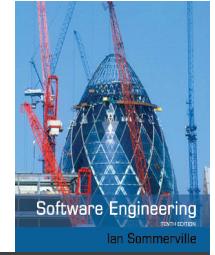
Key points

- ✧ Processes may be structured for iterative development and delivery so that changes may be made without disrupting the system as a whole.
- ✧ The principal approaches to process improvement are agile approaches, geared to reducing process overheads, and maturity-based approaches based on better process management and the use of good software engineering practice.
- ✧ The SEI process maturity framework identifies maturity levels that essentially correspond to the use of good software engineering practice.



Chapter 3 – Agile Software Development

Topics covered



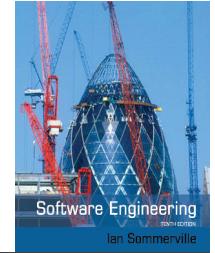
- ✧ Agile methods
- ✧ Agile development techniques
- ✧ Agile project management
- ✧ Scaling agile methods

Rapid software development



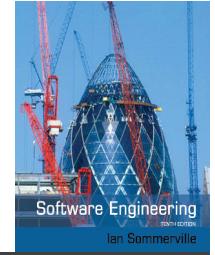
- ✧ Rapid development and delivery is now often the most important requirement for software systems
 - Businesses operate in a fast –changing requirement and it is practically impossible to produce a set of stable software requirements
 - Software has to evolve quickly to reflect changing business needs.
- ✧ 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

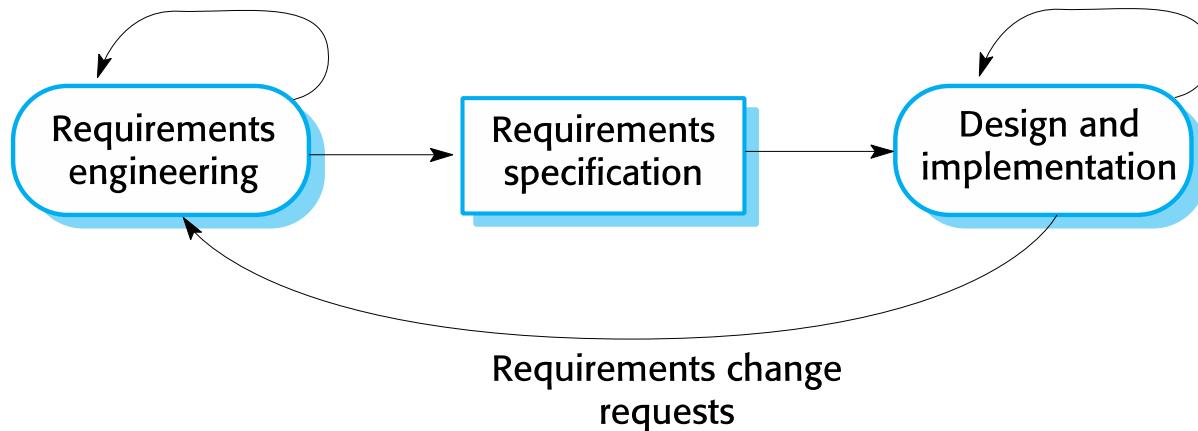


- ✧ Program specification, design and implementation are inter-leaved
- ✧ The system is developed as a series of versions or increments with stakeholders involved in version specification and evaluation
- ✧ Frequent delivery of new versions for evaluation
- ✧ Extensive tool support (e.g. automated testing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

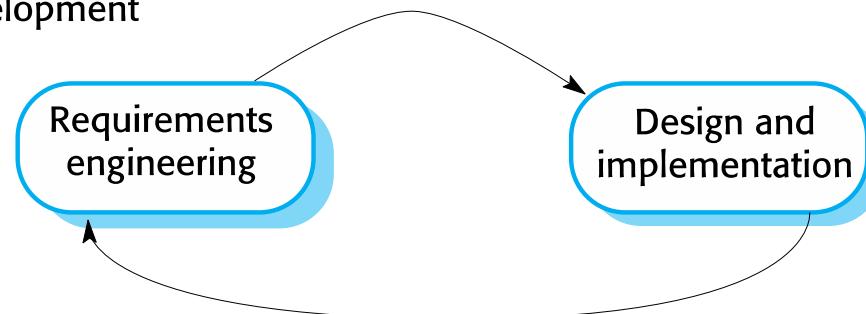
Plan-driven and agile development



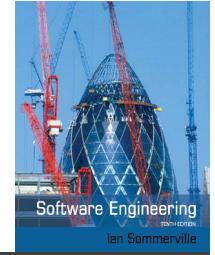
Plan-based development



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.

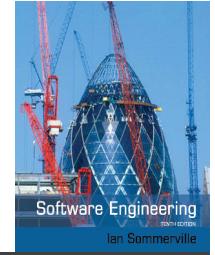
✧ Agile development

- 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



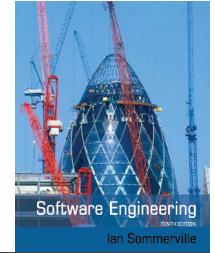
- ✧ *We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:*
 - *Individuals and interactions over processes and tools*
 - Working software over comprehensive documentation*
 - Customer collaboration over contract negotiation*
 - 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

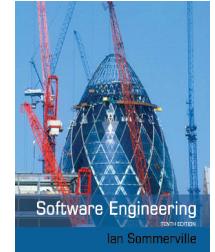


- ✧ Product development where a software company is developing a small or medium-sized product for sale.
 - Virtually all software products and apps are now developed using an agile approach
- ✧ Custom system development within an organization, where there is a clear commitment from the customer to become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



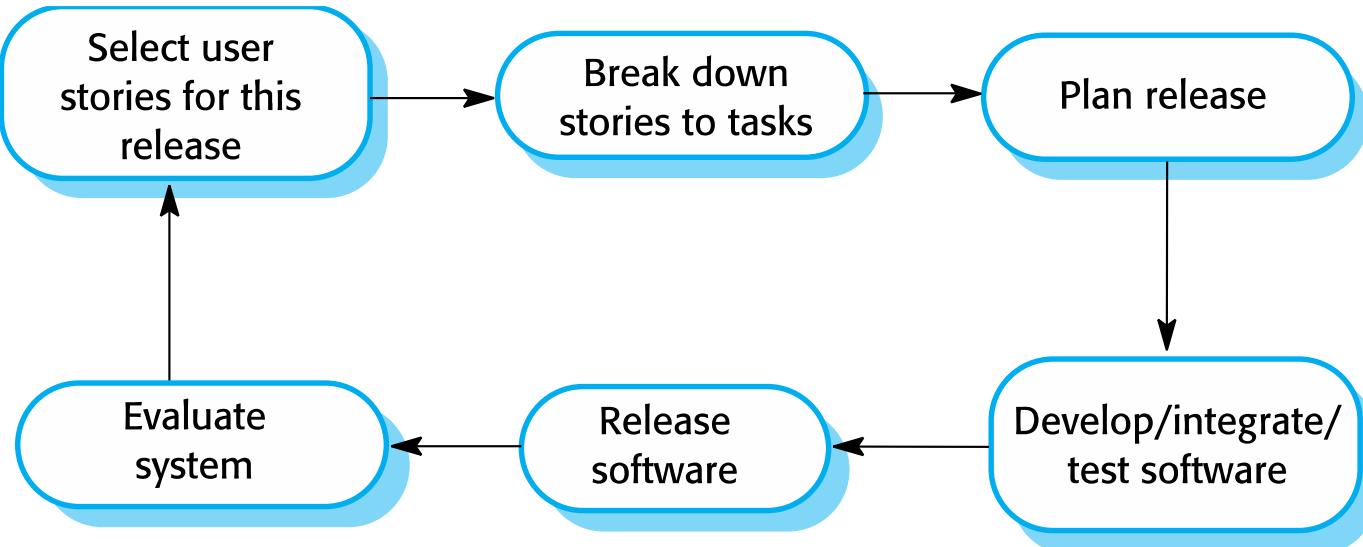
Agile development techniques

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)



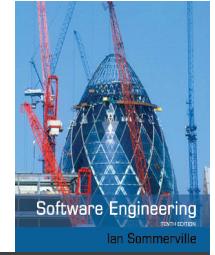
Pair programming	Developers work in pairs, checking each other's work and providing the support to always do a good job.
Collective ownership	The pairs of developers work on all areas of the system, so that no islands of expertise develop and all the developers take responsibility for all of the code. 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



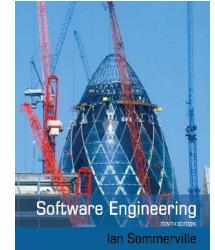
- ✧ Extreme programming has a technical focus and is not easy to integrate with management practice in most organizations.
- ✧ Consequently, while agile development uses practices from XP, the method as originally defined is not widely used.
- ✧ Key practices
 - User stories for specification
 - Refactoring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

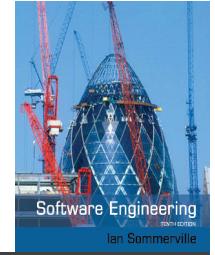
Task 3: Dose checking

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.

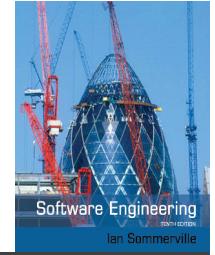
Check the prescribed dose against the minimum and maximum. If outside the range, issue an error message saying that the dose is too high or too low. If within the range, enable the 'Confirm' button.

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



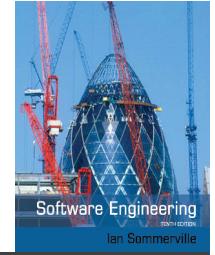
- ✧ Programming team look for possible software improvements and make these improvements even where there is no immediate need for them.
- ✧ This improves the understandability of the software and so reduces the need for documentation.
- ✧ Changes are easier to make because the code is well-structured 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:

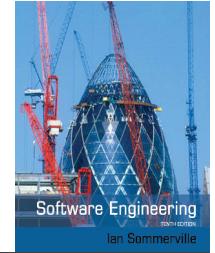
OK or error message indicating that the dose is outside the safe range.

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



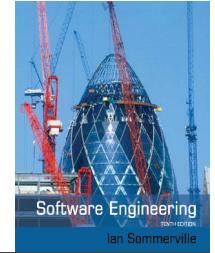
- ✧ Programmers prefer programming to testing and sometimes they take short cuts when writing tests. For example, they may write incomplete tests that do not check for all possible exceptions that may occur.
- ✧ Some tests can be very difficult to write incrementally. For example, in a complex user interface, 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

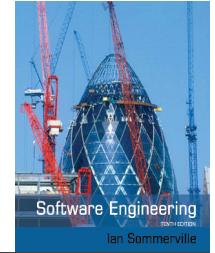


- ✧ In pair programming, programmers sit together at the same computer to develop the software.
- ✧ Pairs are created dynamically so that all team members work with each other during the development process.
- ✧ The sharing of knowledge that happens during pair programming is very important as it reduces 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



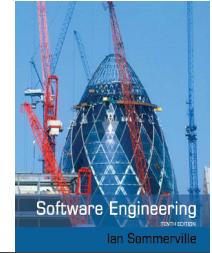
- ✧ The principal responsibility of software project managers is to manage the project so that the software is delivered on time and within the planned budget for the project.
- ✧ The standard approach to project management is plan-driven. Managers draw up a plan for the project showing what should be 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)

Software Engineering

Ian Sommerville

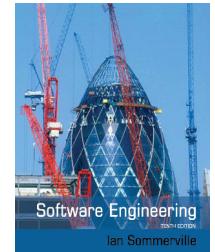
Scrum term	Definition
Development team	A self-organizing group of software developers, which should be no more than 7 people. They are responsible for developing the software and other essential project documents.
Potentially shippable product increment	The software increment that is delivered from a 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.



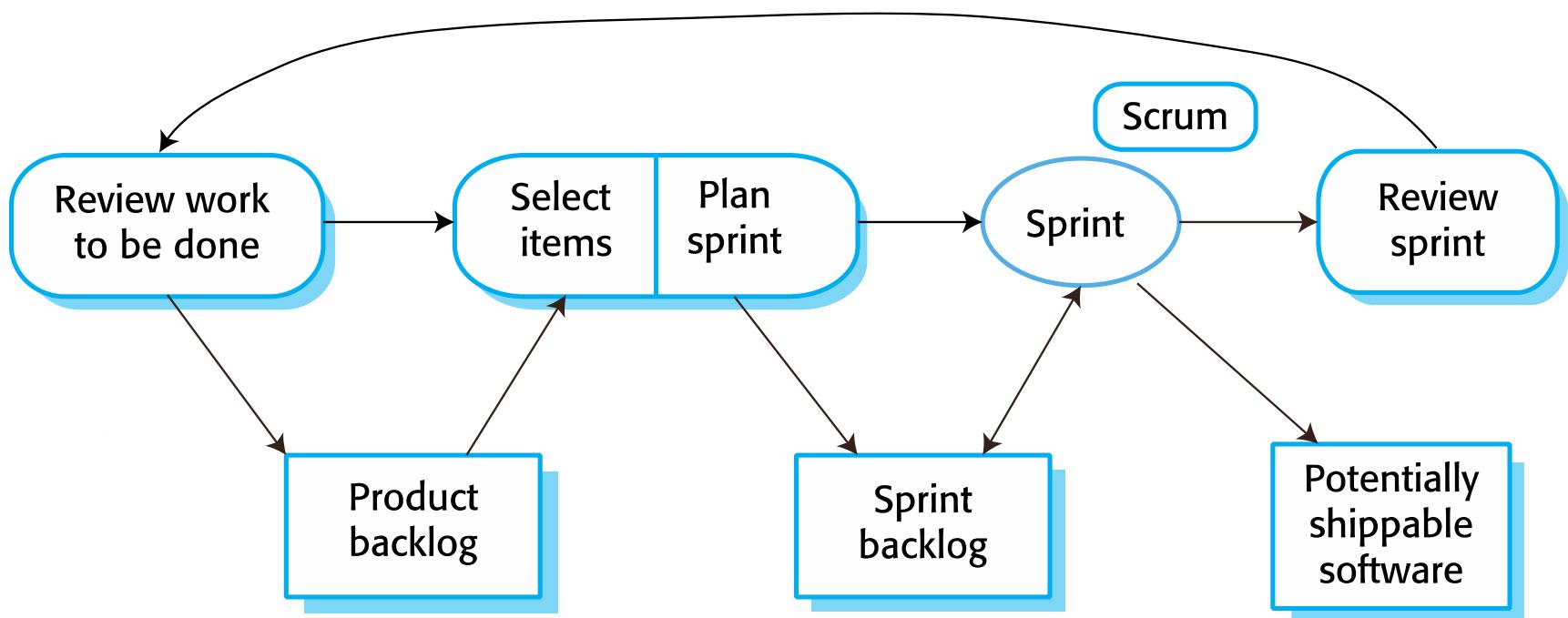
Scrum terminology (b)

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



Software Engineering
Ian Sommerville

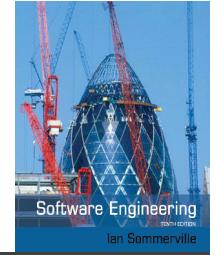


The Scrum sprint cycle



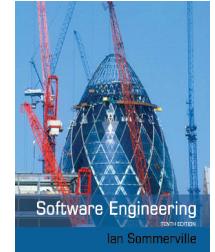
- ✧ Sprints are fixed length, normally 2–4 weeks.
- ✧ The starting point for planning is the product backlog, which is the list of work to be done on the project.
- ✧ The selection phase involves all of the project team who work with the customer to select the features and functionality from the product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



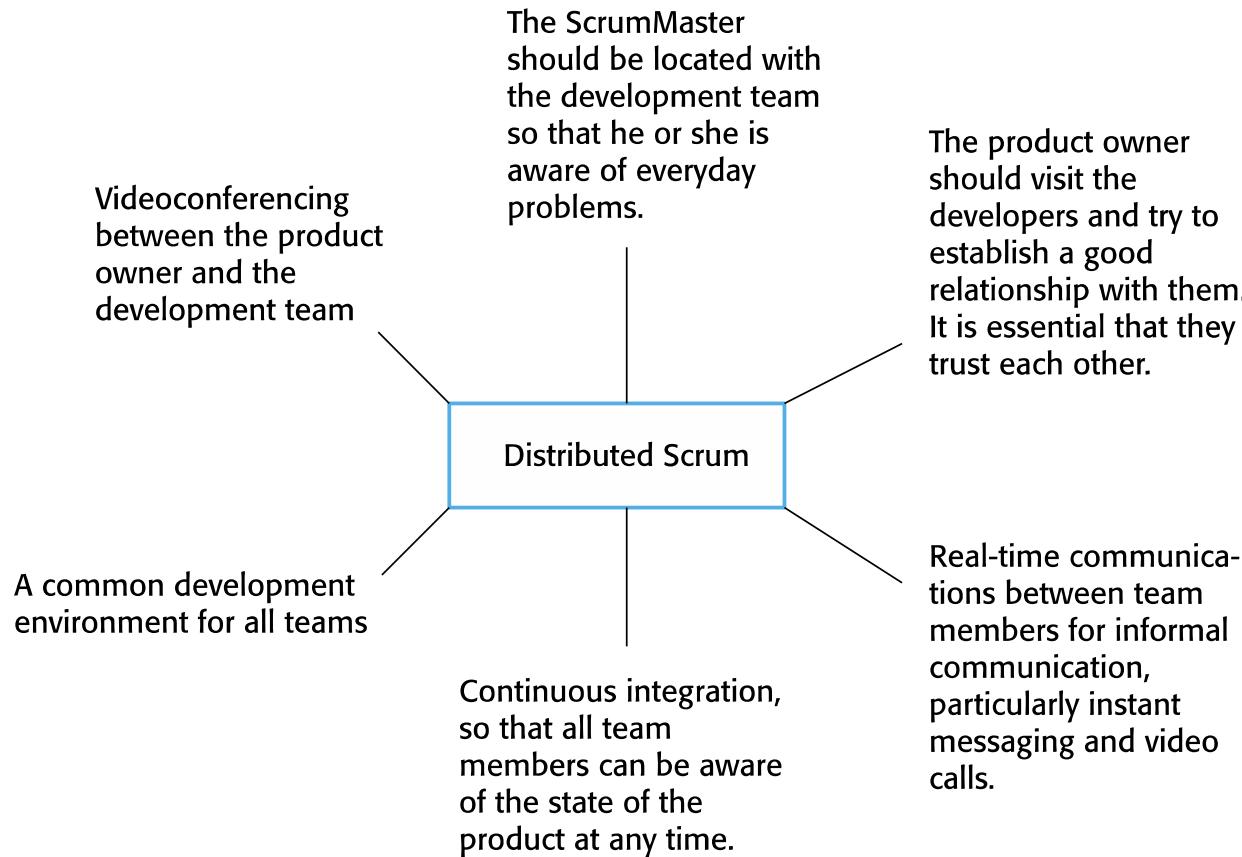
- ✧ The ‘Scrum master’ is a facilitator who arranges daily meetings, tracks the backlog of work to be done, records decisions, measures progress against the backlog and communicates with customers and management outside of the team.
- ✧ The whole team attends short daily meetings (Scrums) where all team 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.

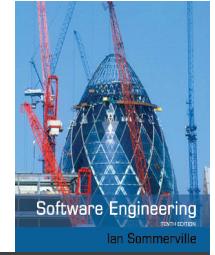
Distributed Scrum





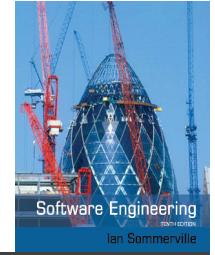
Scaling agile methods

Scaling agile methods



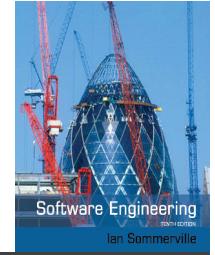
- ✧ Agile methods have proved to be successful for small and medium sized projects that can be developed by a small co-located team.
- ✧ It is sometimes argued that the success of these methods comes because of improved communications which is possible when everyone is working together.
- ✧ Scaling up 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



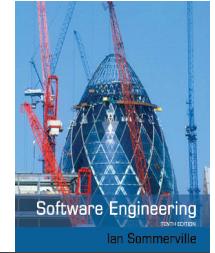
- ✧ Most software contracts for custom systems are based around a specification, which sets out what has to be implemented by the system developer for the system customer.
- ✧ However, this precludes interleaving specification and development as is the norm in agile development.
- ✧ A contract that pays 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



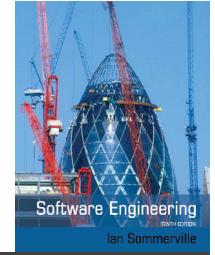
- ✧ Most organizations spend more on maintaining existing software than they do on new software development. So, if agile methods are to be successful, they have to support maintenance as well as original development.
- ✧ Two key issues:
 - Are systems that are developed using an agile approach maintainable, 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



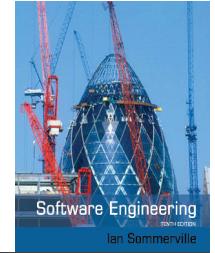
- ✧ Most projects include elements of plan-driven and agile processes. Deciding on the balance depends on:
 - Is it important to have a very detailed specification and design before moving to implementation? If so, you probably need to use a plan-driven approach.
 - Is an incremental delivery strategy, 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



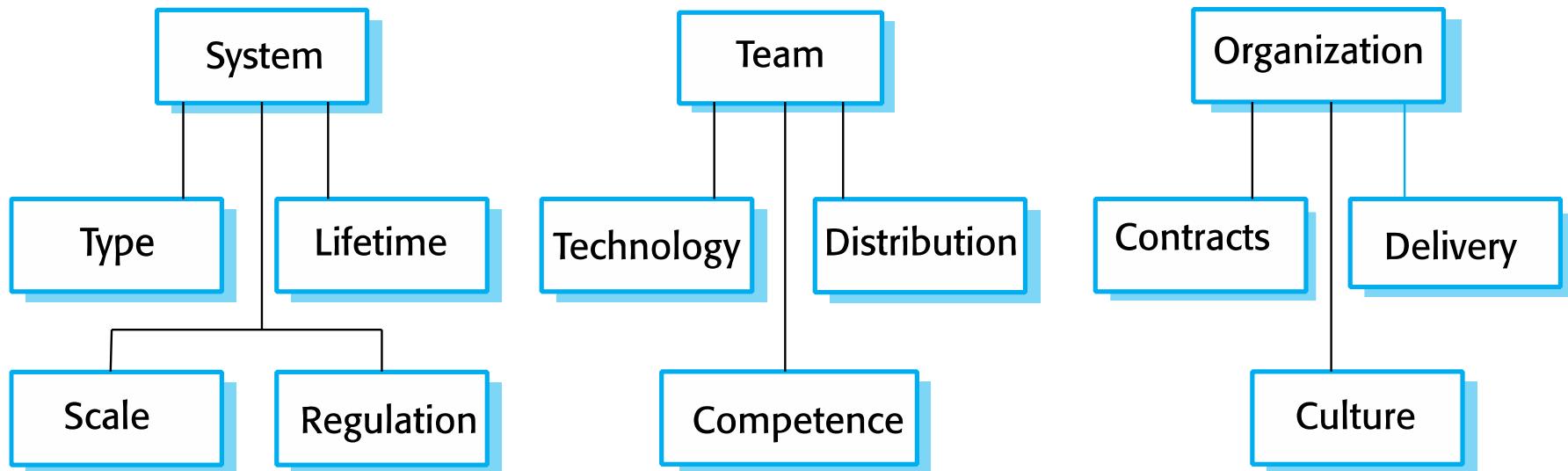
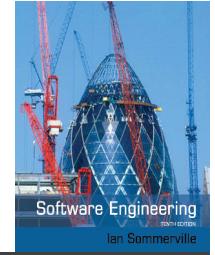
Principle	Practice
Customer involvement	<p>This depends on having a customer who is willing and able to spend time with the development team and who can represent all system stakeholders. Often, customer representatives have other demands on their time and cannot play a full part in the software development.</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

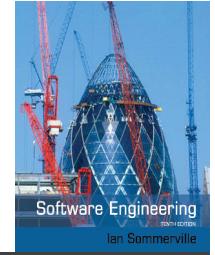




System issues

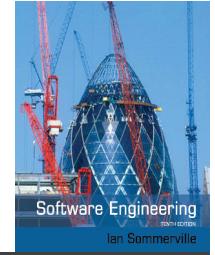
- ✧ How large is the system being developed?
 - Agile methods are most effective a relatively small co-located team who can communicate informally.
- ✧ What type of system is being developed?
 - Systems that require a lot of analysis before implementation need a fairly detailed design to carry out this 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



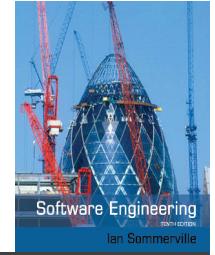
- ✧ Traditional engineering organizations have a culture of plan-based development, as this is the norm in engineering.
- ✧ Is it standard organizational practice to develop a detailed system specification?
- ✧ Will customer representatives be available to provide feedback of system increments?
- ✧ Can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

Agile methods for large systems

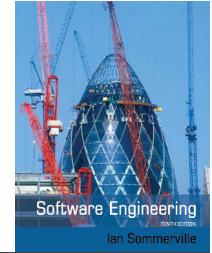


- ✧ Large systems are usually collections of separate, communicating systems, where separate teams develop each system. Frequently, these teams are working in different places, sometimes in different time zones.
- ✧ Large systems are ‘brownfield systems’, that is they include and interact with a number 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

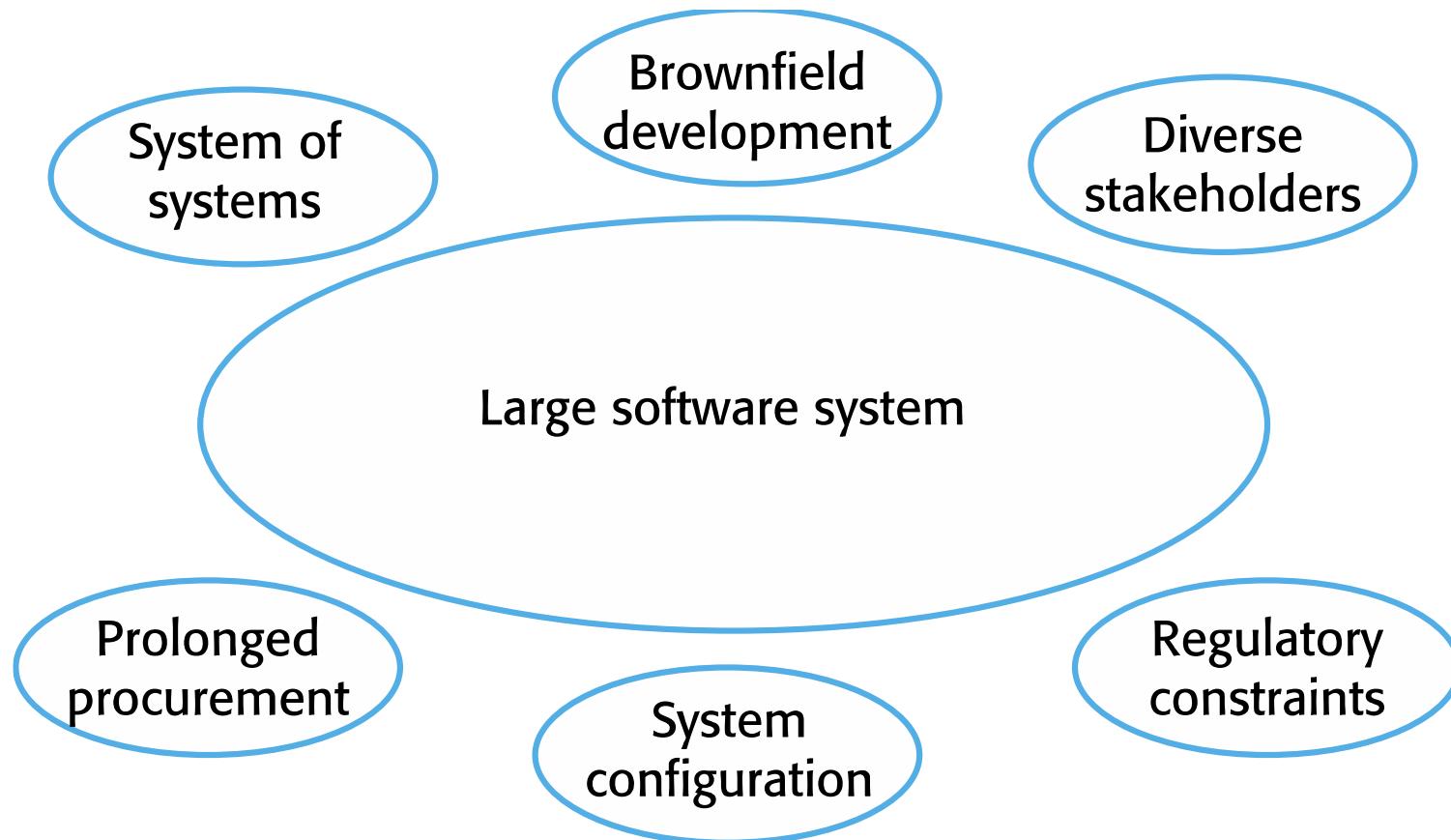


- ✧ Large systems and their development processes are often constrained by external rules and regulations limiting the way that they can be developed.
- ✧ Large systems have a long procurement and development time. It is difficult to maintain coherent teams who know about the system over that period 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.

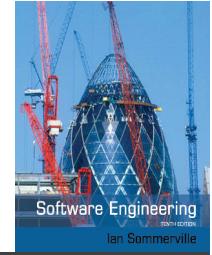


Factors in large systems

Software Engineering
Ian Sommerville

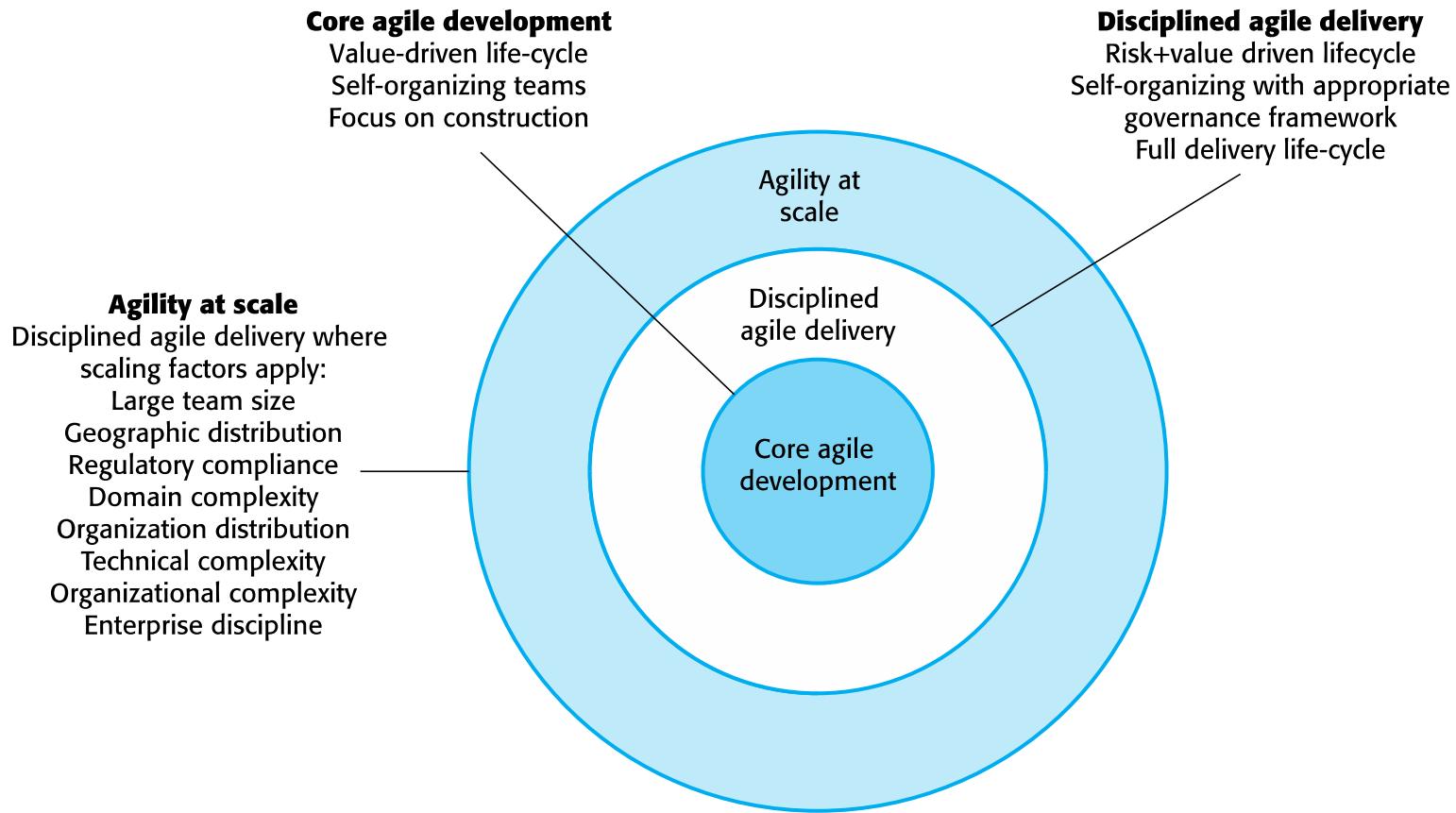


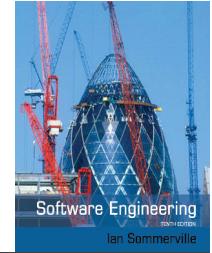
IBM's agility at scale model



Software Engineering

Ian Sommerville

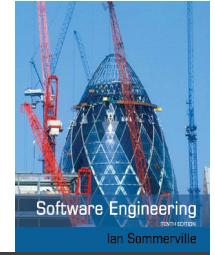




Scaling up to large systems

- ✧ 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.

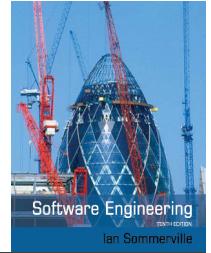
✧ *Scrum of Scrums*

- 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



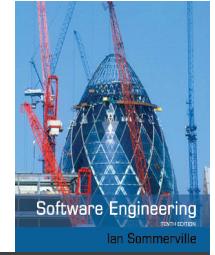
- ✧ Project managers who do not have experience of agile methods may be reluctant to accept the risk of a new approach.
- ✧ Large organizations often have quality procedures and standards that all projects are expected to follow and, because of their bureaucratic nature, these are likely to be incompatible 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

30/10/2014 Chapter 4 Requirements Engineering 1



Topics covered

- ✧ Functional and non-functional requirements
- ✧ Requirements engineering processes
- ✧ Requirements elicitation
- ✧ Requirements specification
- ✧ Requirements validation
- ✧ Requirements change

30/10/2014 Chapter 4 Requirements Engineering 2



Requirements engineering

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

30/10/2014 Chapter 4 Requirements Engineering 3



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.

30/10/2014 Chapter 4 Requirements Engineering 4



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."

30/10/2014 Chapter 4 Requirements Engineering 5



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.

30/10/2014 Chapter 4 Requirements Engineering 6

User and system requirements

User requirements definition

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

System requirements specification

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

30/10/2014 Chapter 4 Requirements Engineering 7

Readers of different types of requirements specification

```

graph LR
    UR[User requirements] --> CM[Client managers  
System end-users  
Client engineers  
Contractor managers  
System architects]
    SR[System requirements] --> SEU[System end-users  
Client engineers  
System architects  
Software developers]
  
```

30/10/2014 Chapter 4 Requirements Engineering 8

System stakeholders

- ✧ Any person or organization who is affected by the system in some way and so who has a legitimate interest
- ✧ Stakeholder types
 - End users
 - System managers
 - System owners
 - External stakeholders

30/10/2014 Chapter 4 Requirements Engineering 9

Stakeholders in the Mentcare system

- ✧ 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.

30/10/2014 Chapter 4 Requirements Engineering 10

Stakeholders in the Mentcare system

- ✧ 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.

30/10/2014 Chapter 4 Requirements Engineering 11

Agile methods and requirements

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

30/10/2014 Chapter 4 Requirements Engineering 12



Functional and non-functional requirements

30/10/2014 Chapter 4 Requirements Engineering 13

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

30/10/2014 Chapter 4 Requirements Engineering 14



Functional requirements

30/10/2014 Chapter 4 Requirements Engineering 15

Mentcare system: functional requirements

- ❖ 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.

30/10/2014 Chapter 4 Requirements Engineering 16



Requirements imprecision

30/10/2014 Chapter 4 Requirements Engineering 17

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, because of system and environmental complexity, it is impossible to produce a complete and consistent requirements document.

30/10/2014 Chapter 4 Requirements Engineering 18

Non-functional requirements

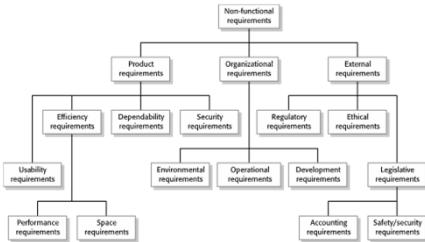
- ◊ These define system properties and constraints e.g. reliability, response time and storage requirements. Constraints are I/O device capability, system representations, etc.
- ◊ Process requirements may also be specified mandating a particular IDE, programming language or development method.
- ◊ Non-functional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

30/10/2014

Chapter 4 Requirements Engineering

19

Types of nonfunctional requirement



30/10/2014

Chapter 4 Requirements Engineering

20

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.

30/10/2014

Chapter 4 Requirements Engineering

21

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.

30/10/2014

Chapter 4 Requirements Engineering

22

Examples of nonfunctional requirements in the Mentcare system

Product requirement

The Mentcare system 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 Mentcare 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.

30/10/2014

Chapter 4 Requirements Engineering

23

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.

30/10/2014

Chapter 4 Requirements Engineering

24

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)

30/10/2014

Chapter 4 Requirements Engineering

25

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

30/10/2014

Chapter 4 Requirements Engineering

26

Requirements engineering processes



30/10/2014

Chapter 4 Requirements Engineering

27

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.

30/10/2014

Chapter 4 Requirements Engineering

28

A spiral view of the requirements engineering process



30/10/2014

Chapter 4 Requirements Engineering

29



Requirements elicitation



30/10/2014

Chapter 4 Requirements Engineering

30

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**.

30/10/2014 Chapter 4 Requirements Engineering 31

Requirements elicitation



30/10/2014 Chapter 4 Requirements Engineering 32

Requirements elicitation



◊ 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.

30/10/2014 Chapter 4 Requirements Engineering 33

Problems of requirements elicitation



◊ Stakeholders don't know what they really want.

◊ Stakeholders express requirements in their own terms.

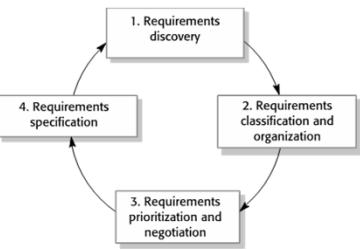
◊ Different stakeholders may have conflicting requirements.

◊ Organisational and political factors may influence the system requirements.

◊ The requirements change during the analysis process. New stakeholders may emerge and the business environment may change.

30/10/2014 Chapter 4 Requirements Engineering 34

The requirements elicitation and analysis process

30/10/2014 Chapter 4 Requirements Engineering 35

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.

30/10/2014 Chapter 4 Requirements Engineering 36

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.

30/10/2014

Chapter 4 Requirements Engineering

37

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.

30/10/2014

Chapter 4 Requirements Engineering

38

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.
- ✧ Interviewers need to be open-minded without pre-conceived ideas of what the system should do
- ✧ You need to prompt the user to talk about the system by suggesting requirements rather than simply asking them what they want.

30/10/2014

Chapter 4 Requirements Engineering

39

Problems with interviews



- ✧ Application specialists may use language to describe their work that isn't easy for the requirements engineer to understand.
- ✧ 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.

30/10/2014

Chapter 4 Requirements Engineering

40

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.

30/10/2014

Chapter 4 Requirements Engineering

41

Scope of ethnography



- ✧ Requirements that are derived from the way that people actually work rather than the way I 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.

30/10/2014

Chapter 4 Requirements Engineering

42

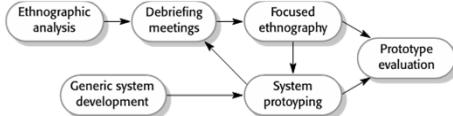
Focused ethnography



❖ 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.

30/10/2014 Chapter 4 Requirements Engineering 43

Ethnography and prototyping for requirements analysis

30/10/2014 Chapter 4 Requirements Engineering 44

Stories and scenarios



❖ Scenarios and user stories are real-life examples of how a system can be used.
 ❖ Stories and scenarios are a description of how a system may be used for a particular task.
 ❖ Because they are based on a practical situation, stakeholders can relate to them and can comment on their situation with respect to the story.

30/10/2014 Chapter 4 Requirements Engineering 45

Photo sharing in the classroom (iLearn)



❖ Jack is a primary school teacher in Ullapool (a village in northern Scotland). He has decided that a class project should be focused around the fishing industry in the area, looking at the history, development and economic impact of fishing. As part of this, pupils are asked to gather and share reminiscences from relatives, use newspaper archives and collect old photographs related to fishing and fishing communities in the area. Pupils use an iLearn wiki to gather together fishing stories and SCARAN (a history resources site) to access newspaper archives and photographs. However, Jack also needs a photo sharing site as he wants pupils to take and comment on each others photos and to upload scans of old photographs that they may have in their families.

Jack sends an email to a primary school teachers group, which he is a member of to see if anyone can recommend an appropriate system. Two teachers reply and both suggest that he uses KidsTakePics, a photo sharing site that allows teachers to check and moderate content. As KidsTakePics is not integrated with the iLearn authentication service, he sets up a teacher and a class account. He uses the iLearn setup service to add KidsTakePics to the services seen by the pupils in his class so that when they log in, they can immediately use the system to upload photos from their mobile devices and class computers.

30/10/2014 Chapter 4 Requirements Engineering 46

Scenarios



❖ A structured form of user story
 ❖ Scenarios 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.

30/10/2014 Chapter 4 Requirements Engineering 47

Uploading photos iLearn



❖ **Initial assumption:** A user or a group of users have one or more digital photographs to be uploaded to the picture sharing site. These are saved on either a tablet or laptop computer. They have successfully logged on to KidsTakePics.

❖ **Normal:** The user chooses upload photos and they are prompted to select the photos to be uploaded on their computer and to select the project name under which the photos will be stored. They should also be given the option of inputting keywords that should be associated with each uploaded photo. Uploaded photos are named by creating a conjunction of the user name with the filename of the photo on the local computer.

❖ On completion of the upload, the system automatically sends an email to the project moderator asking them to check new content and generates an on-screen message to the user that this has been done.

30/10/2014 Chapter 4 Requirements Engineering 48

Uploading photos

Software Engineering

- ❖ **What can go wrong:**
- ❖ No moderator is associated with the selected project. An email is automatically generated to the school administrator asking them to nominate a project moderator. Users should be informed that there could be a delay in making their photos visible.
- ❖ Photos with the same name have already been uploaded by the same user. The user should be asked if they wish to re-upload the photos with the same name, rename the photos or cancel the upload. If they chose to re-upload the photos, the originals are overwritten. If they chose to rename the photos, a new name is automatically generated by adding a number to the existing file name.
- ❖ **Other activities:** The moderator may be logged on to the system and may approve photos as they are uploaded.
- ❖ **System state on completion:** User is logged on. The selected photos have been uploaded and assigned a status 'awaiting moderation'. Photos are visible to the moderator and to the user who uploaded them.

30/10/2014 Chapter 4 Requirements Engineering 49

Requirements specification

Software Engineering

30/10/2014 Chapter 4 Requirements Engineering 50

Requirements specification

Software Engineering

- ❖ 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.

30/10/2014 Chapter 4 Requirements Engineering 51

Ways of writing a system requirements specification

Software Engineering

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.

30/10/2014 Chapter 4 Requirements Engineering 52

Requirements and design

Software Engineering

- ❖ 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.

30/10/2014 Chapter 4 Requirements Engineering 53

Natural language specification

Software Engineering

- ❖ 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.

30/10/2014 Chapter 4 Requirements Engineering 54

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.

30/10/2014

Chapter 4 Requirements Engineering

55

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.

30/10/2014

Chapter 4 Requirements Engineering

56

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.*)

30/10/2014

Chapter 4 Requirements Engineering

57

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.

30/10/2014

Chapter 4 Requirements Engineering

58

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.

30/10/2014

Chapter 4 Requirements Engineering

59

A structured specification of a requirement for an insulin pump



Insulin Pump/Control Software/SRS/3.3.2

Function Compute insulin dose: safe sugar level.

Description

Computes the dose of insulin to be delivered when the current measured sugar level is in the safe zone between 3 and 7 units.

Inputs Current sugar reading (r2); the previous two readings (r0 and r1).

Source Current sugar reading from sensor. Other readings from memory.

Outputs CompDose—the dose in insulin to be delivered.

Destination Main control loop.

30/10/2014

Chapter 4 Requirements Engineering

60

A structured specification of a requirement for an insulin pump

Action

CompDose is zero if the sugar level is stable or falling or if the level is increasing but the rate of increase is decreasing. If the level is increasing and the rate of increase is increasing, then CompDose is computed by dividing the difference between the current sugar level and the previous level by 4 and rounding the result. If the result, is rounded to zero then CompDose is set to the minimum dose that can be delivered.

Requirements

Two previous readings so that the rate of change of sugar level can be computed.

Pre-condition

The insulin reservoir contains at least the maximum allowed single dose of insulin.

Post-condition r0 is replaced by r1 then r1 is replaced by r2.

Side effects None.

30/10/2014

Chapter 4 Requirements Engineering

61

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 ($r2 < r1$)	CompDose = 0
Sugar level stable ($r2 = r1$)	CompDose = 0
Sugar level increasing and rate of increase decreasing $((r2 - r1) < (r1 - r0))$	CompDose = 0
Sugar level increasing and rate of increase stable or increasing $((r2 - r1) \geq (r1 - r0))$	$\text{CompDose} = \text{round}((r2 - r1)/4)$ If rounded result = 0 then $\text{CompDose} = \text{MinimumDose}$

30/10/2014

Chapter 4 Requirements Engineering

63

Use cases

- Use-cases are a kind of scenario that are included in the UML.

- Use cases identify the actors in an interaction and which describe the interaction itself.

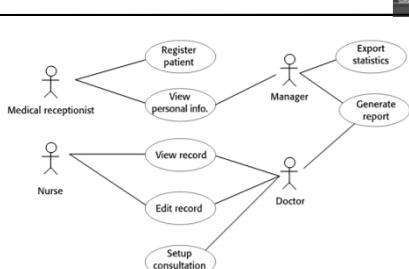
- A set of use cases should describe all possible interactions with the system.

- High-level graphical model supplemented by more detailed tabular description (see Chapter 5).

- UML 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 Mentcare system



30/10/2014

Chapter 4 Requirements Engineering

65

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.



Users of a requirements document

```

graph LR
    SC[System customers] --> S1[Specify the requirements and read them to check that they meet their needs. Customers specify changes to the requirements.]
    M[Managers] --> S2[Use the requirements document to plan a bid for the system and to plan the system development process.]
    SE[System engineers] --> S3[Use the requirements to understand what system is to be developed.]
    STE[System test engineers] --> S4[Use the requirements to develop validation tests for the system.]
    SME[System maintenance engineers] --> S5[Use the requirements to understand the system and the relationships between its parts.]
  
```

30/10/2014 Chapter 4 Requirements Engineering 67

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.

30/10/2014 Chapter 4 Requirements Engineering 68

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.

30/10/2014 Chapter 4 Requirements Engineering 69

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.

30/10/2014 Chapter 4 Requirements Engineering 70

Requirements validation

30/10/2014 Chapter 4 Requirements Engineering 71

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.

30/10/2014 Chapter 4 Requirements Engineering 72

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?

30/10/2014

Chapter 4 Requirements Engineering

73

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.

30/10/2014

Chapter 4 Requirements Engineering

74

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.

30/10/2014

Chapter 4 Requirements Engineering

75

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?

30/10/2014

Chapter 4 Requirements Engineering

76

Requirements change



30/10/2014

Chapter 4 Requirements Engineering

77

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.

30/10/2014

Chapter 4 Requirements Engineering

78

Changing requirements



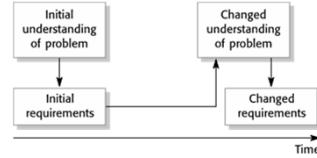
- ✧ Large systems usually have a diverse user community, with many users having different requirements and priorities that may be conflicting or contradictory.
- The final system requirements are inevitably a compromise between them and, with experience, it is often discovered that the balance of support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

30/10/2014

Chapter 4 Requirements Engineering

79

Requirements evolution



30/10/2014

Chapter 4 Requirements Engineering

80

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.

30/10/2014

Chapter 4 Requirements Engineering

81

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.

30/10/2014

Chapter 4 Requirements Engineering

82

Requirements change management



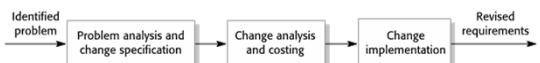
- ✧ Deciding if a requirements change should be accepted
 - *Problem analysis and change specification*
 - During this stage, the problem or the change proposal is analyzed to check that it is valid. This analysis is fed back to the change requestor who may respond with a more specific requirements change 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.

30/10/2014

Chapter 4 Requirements Engineering

83

Requirements change management



30/10/2014

Chapter 4 Requirements Engineering

84

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.

30/10/2014

Chapter 4 Requirements Engineering

85

Key points

- ✧ The requirements engineering process is an iterative process that includes requirements elicitation, specification and validation.
- ✧ Requirements elicitation is an iterative process that can be represented as a spiral of activities – requirements discovery, requirements classification and organization, requirements negotiation and requirements documentation.
- ✧ You can use a range of techniques for requirements elicitation including interviews and ethnography. User stories and scenarios may be used to facilitate discussions.

30/10/2014

Chapter 4 Requirements Engineering

86

Key points

- ✧ Requirements specification is the process of formally documenting the user and system requirements and creating a software requirements document.
- ✧ 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.

30/10/2014

Chapter 4 Requirements Engineering

87

Key points

- ✧ 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.

30/10/2014

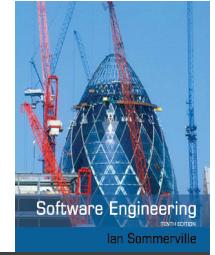
Chapter 4 Requirements Engineering

88



Chapter 5 – System Modeling

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 has now come to mean representing a system using some kind of graphical notation, which is now almost always based on notations in the Unified Modeling Language (UML).
- ✧ System modelling helps the analyst to understand the functionality of the system and models are used to communicate with customers.

Existing and planned system models



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

System perspectives



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



UML diagram types

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

Use of graphical models



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



Software Engineering

Ian Sommerville

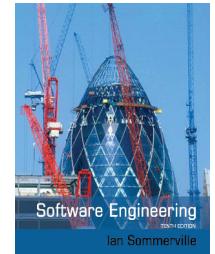
Context models

Context models



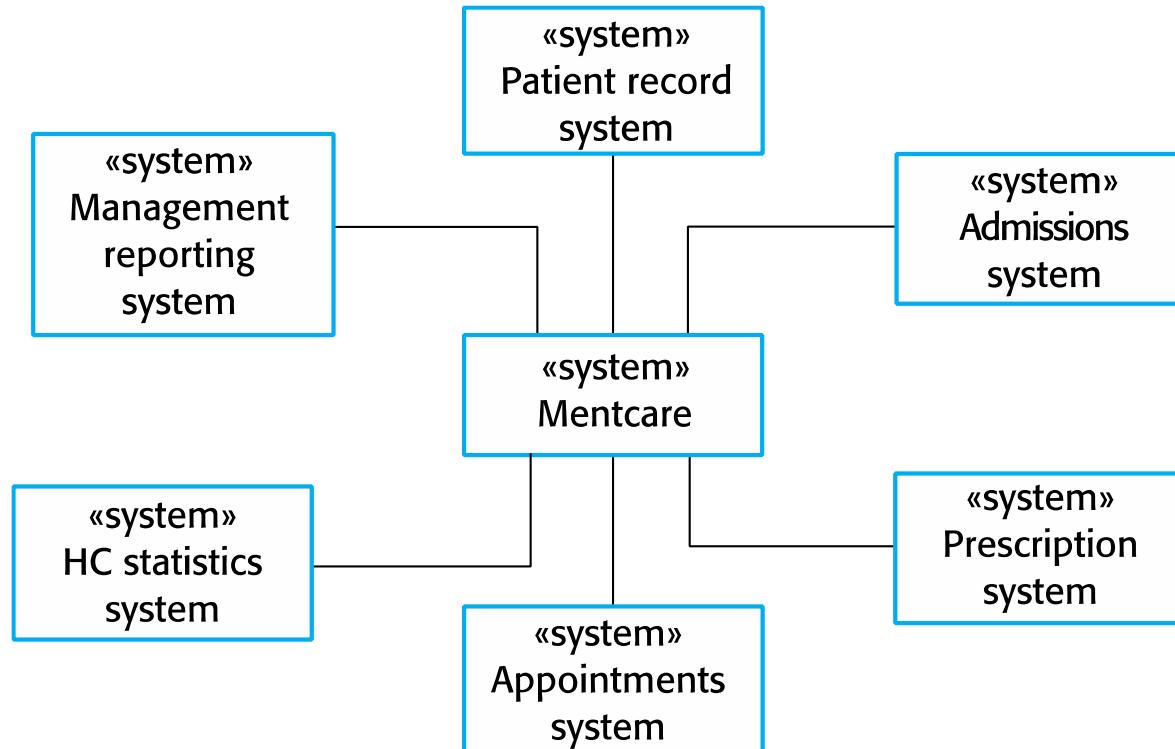
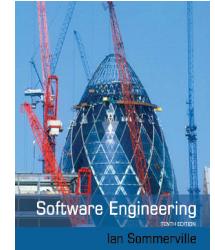
- ✧ Context models are used to illustrate the operational context of a system - they show what lies outside the system boundaries.
- ✧ Social and organisational concerns may affect the decision on where to position system boundaries.
- ✧ Architectural models show the system and its relationship with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

System boundaries

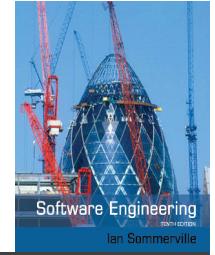


- ✧ System boundaries are established to define what is inside and what is outside the system.
 - They show other systems that are used or depend on the system being developed.
- ✧ The position of the system boundary has a profound effect on the system requirements.
- ✧ Defining a system boundary is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

The context of the Mentcare system

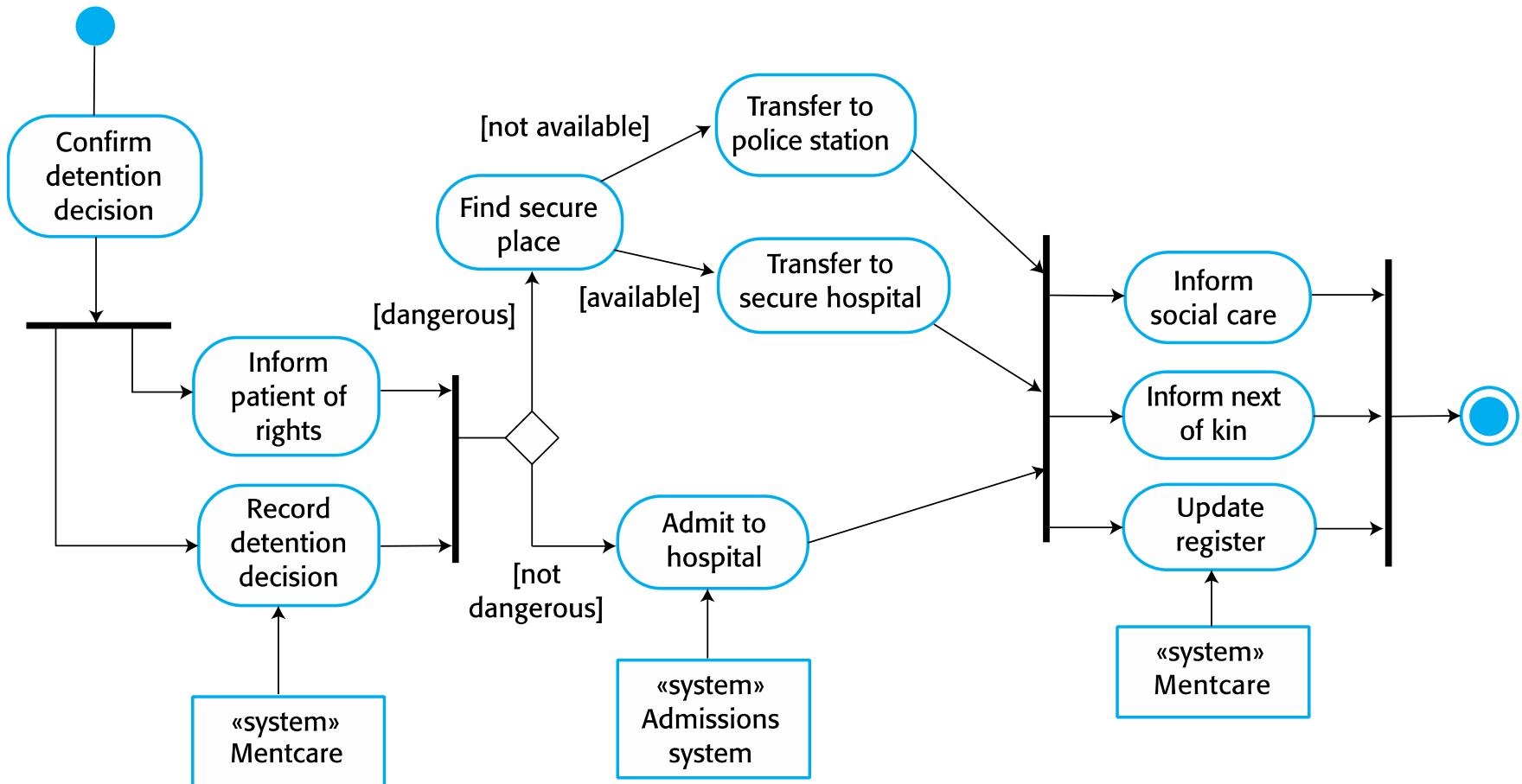
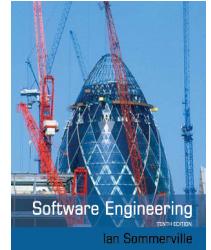


Process perspective



- ✧ 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.

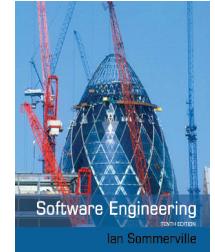
Process model of involuntary detention





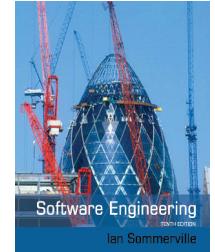
Interaction models

Interaction models



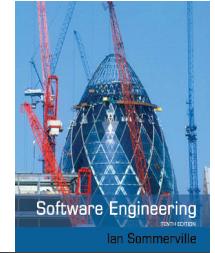
- ✧ Modeling user interaction is important as it helps to identify user requirements.
- ✧ Modeling system-to-system interaction highlights the communication problems that may arise.
- ✧ Modeling component interaction helps us understand if a proposed system structure is likely to deliver the required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

Use case modeling



- ✧ Use cases were developed originally to support requirements elicitation and now incorporated into the UML.
- ✧ 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

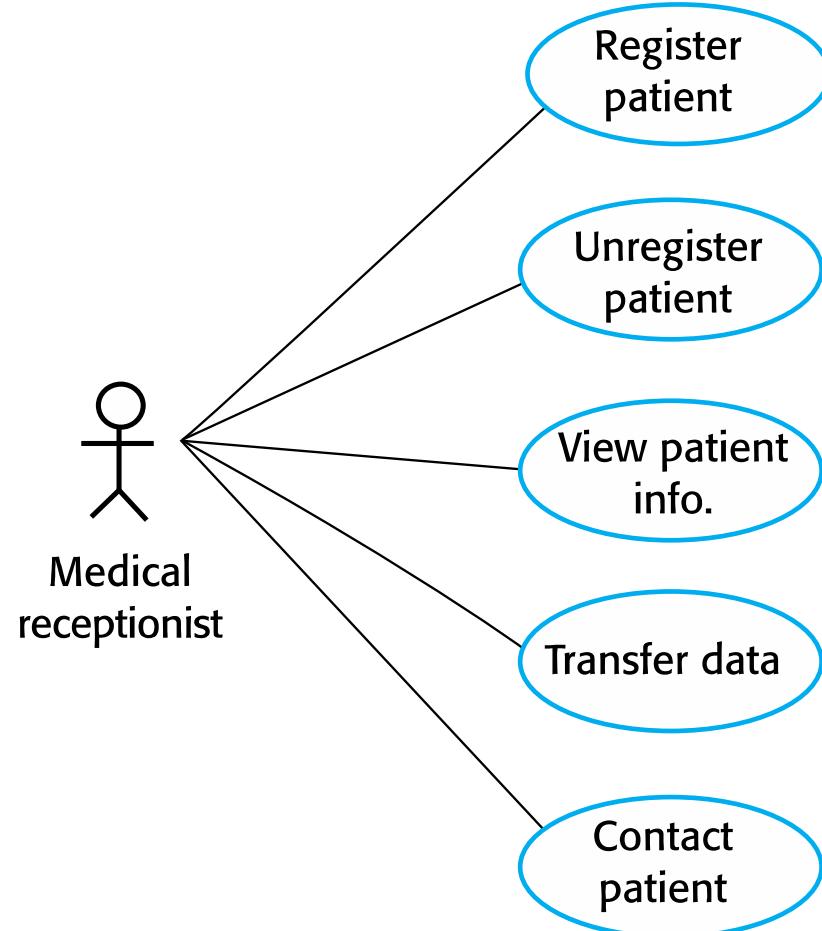
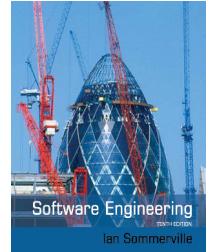


Tabular description of the ‘Transfer data’ use-case

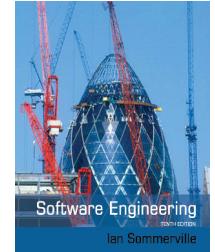


MHC-PMS: Transfer data	
Actors	Medical receptionist, patient records system (PRS)
Description	A receptionist may transfer data from the Mentcase system to a general patient record database that is maintained by a health authority. The information transferred may either be updated personal information (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 system involving the role 'Medical Receptionist'



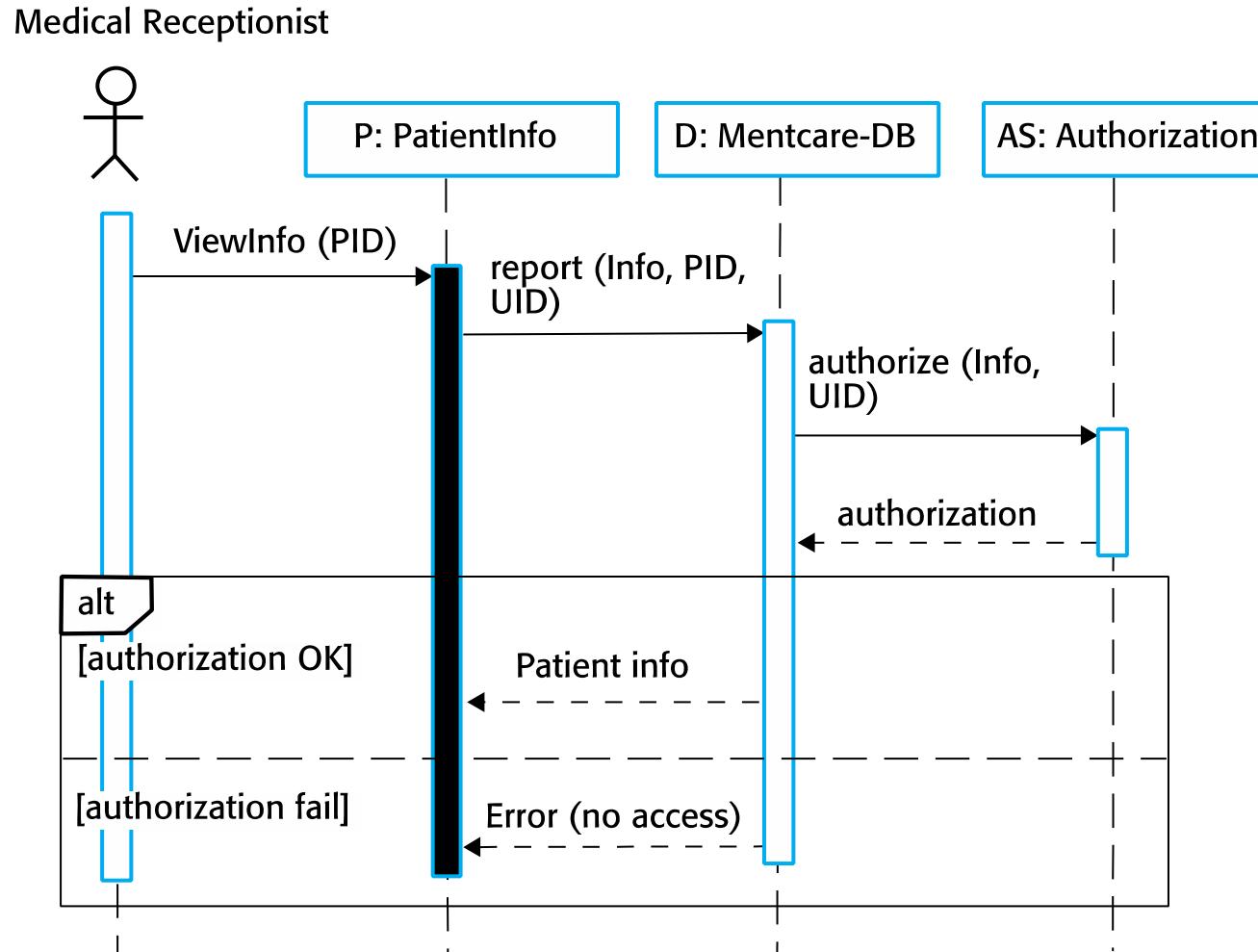
Sequence diagrams

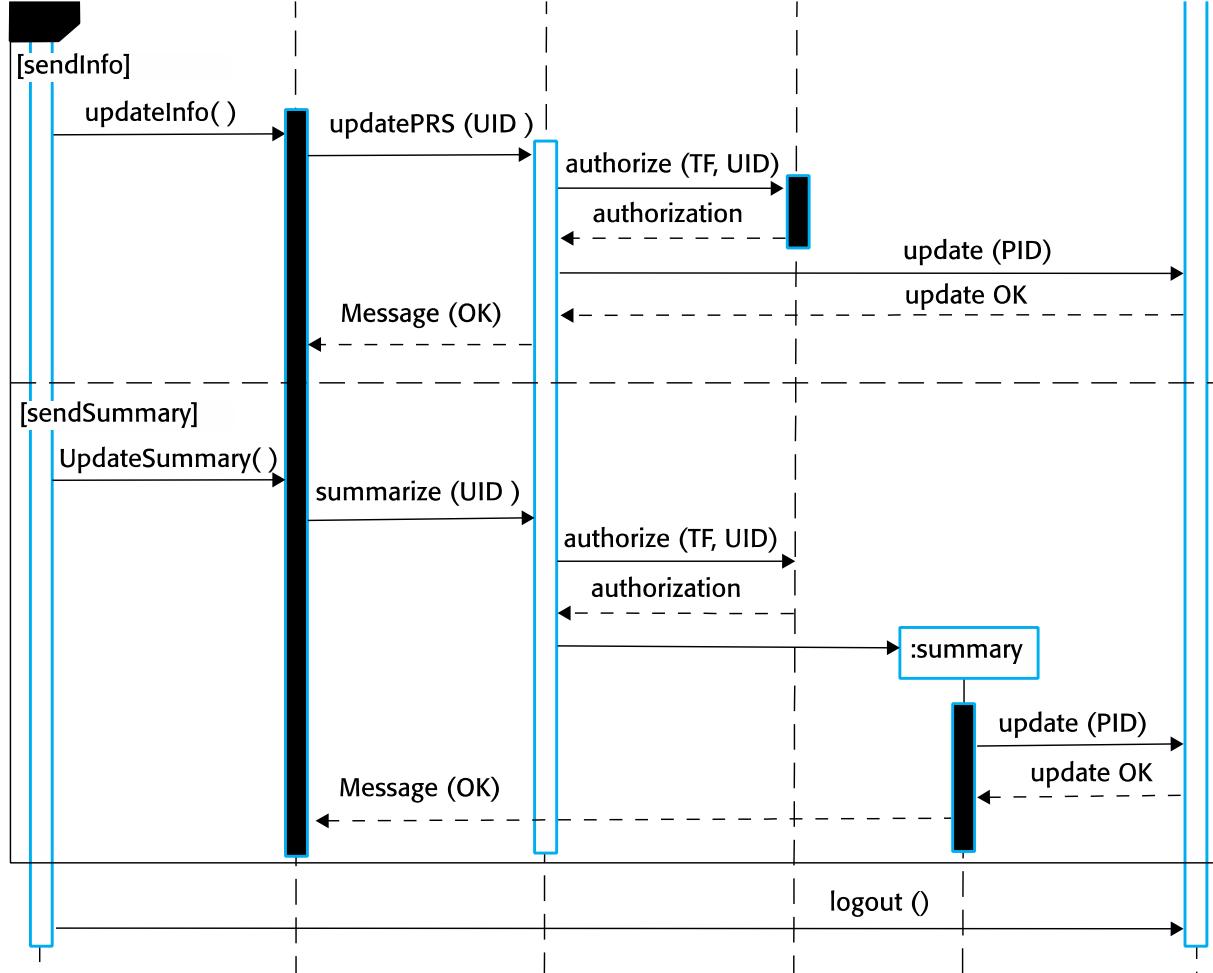
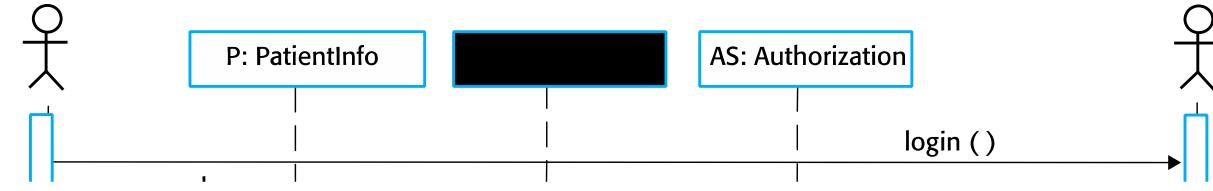


- ✧ Sequence diagrams are part of the UML and are used to model the interactions between the actors and the objects within a system.
- ✧ A sequence diagram shows the sequence of interactions that take place during a particular use case or use case instance.
- ✧ The objects and actors involved are listed 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

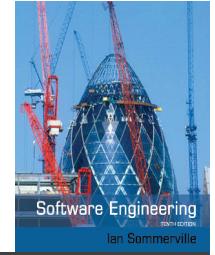


Software Engineering

Ian Sommerville

Structural models

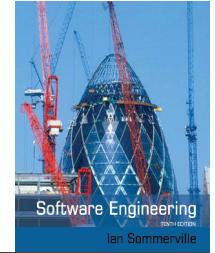
Structural models



Software Engineering
Ian Sommerville

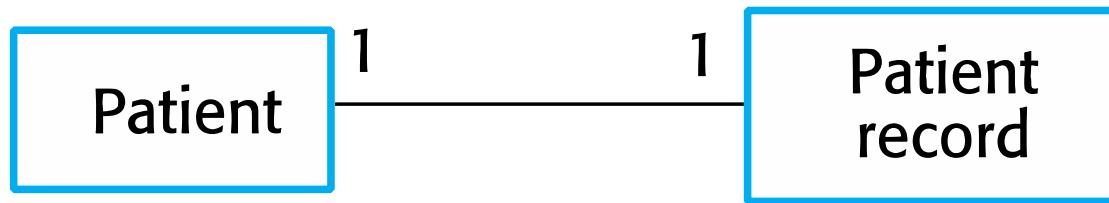
- ✧ Structural models of software display the organization of a system in terms of the components that make up that system and their relationships.
- ✧ Structural models may be static models, which show the structure of the system design, or dynamic models, which show the organization of the system when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

Class diagrams

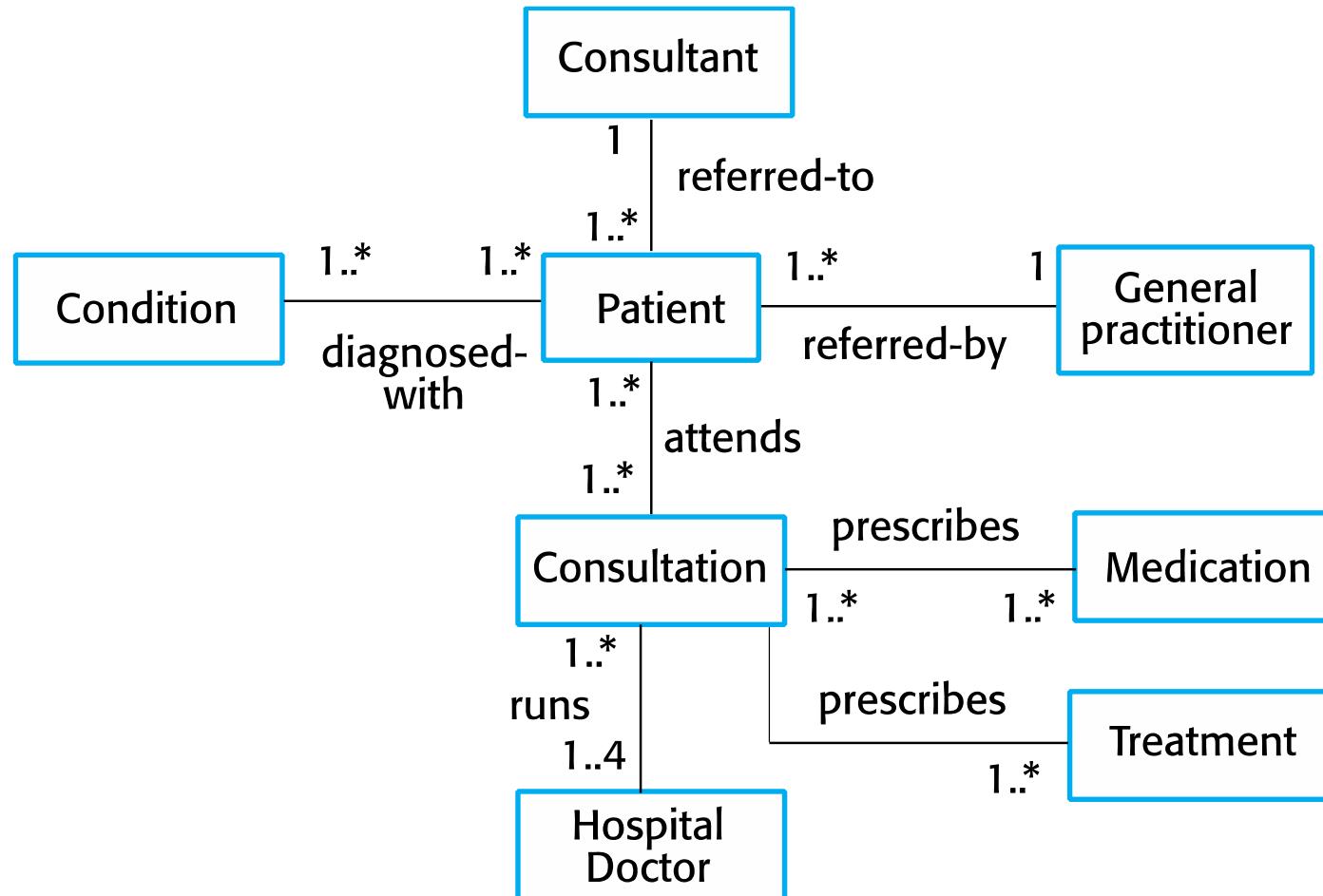


- ✧ Class diagrams are used when developing an object-oriented system model to show the classes in a system and the associations between these classes.
- ✧ An object class can be thought of as a general definition of one kind of system object.
- ✧ An association is a link between classes that indicates 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.

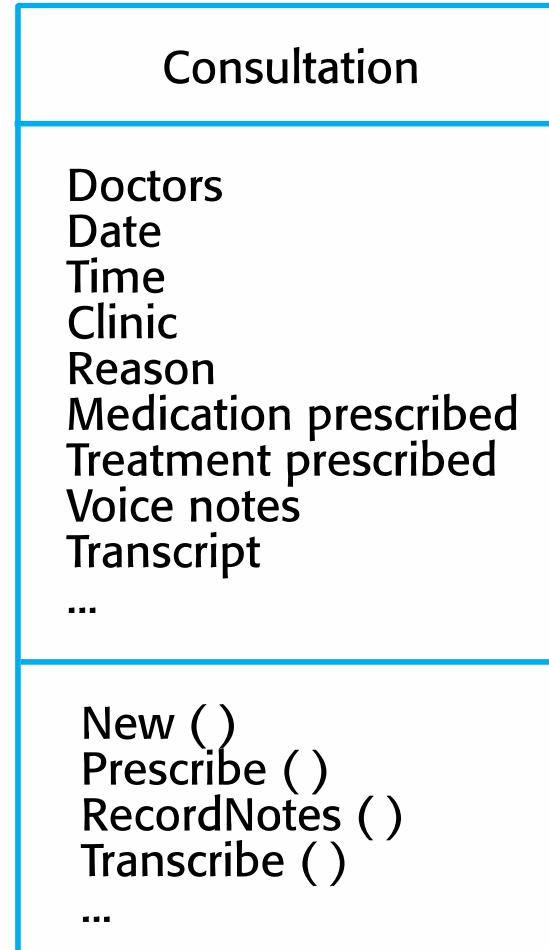
UML classes and association



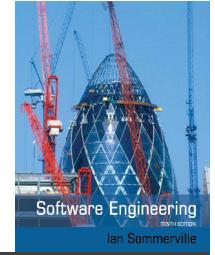
Classes and associations in the MHC-PMS



The Consultation class



Generalization

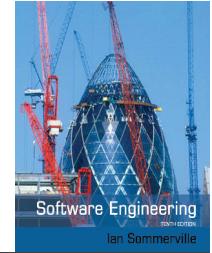


- ✧ Generalization is an everyday technique that we use to manage complexity.
- ✧ Rather than learn the detailed characteristics of every entity that we experience, we place these entities in more general classes (animals, cars, houses, etc.) and learn the characteristics of these classes.
- ✧ This allows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

Generalization



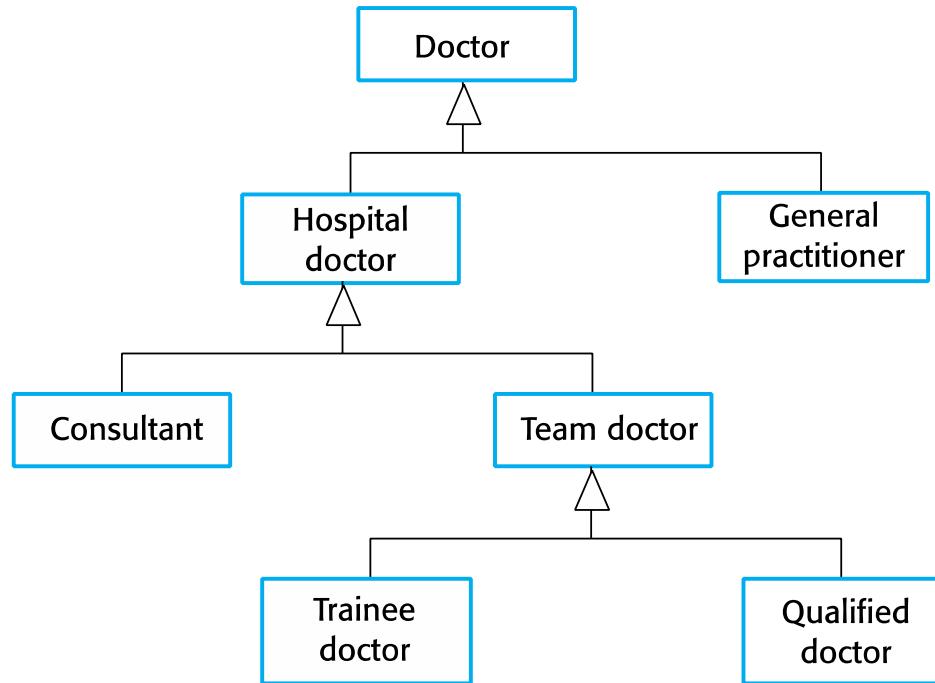
- ✧ In modeling systems, it is often useful to examine the classes in a system to see if there is scope for generalization. If changes are proposed, then you do not have to look at all classes in the system to see if they are affected by the change.
- ✧ In object-oriented languages, such as Java, generalization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



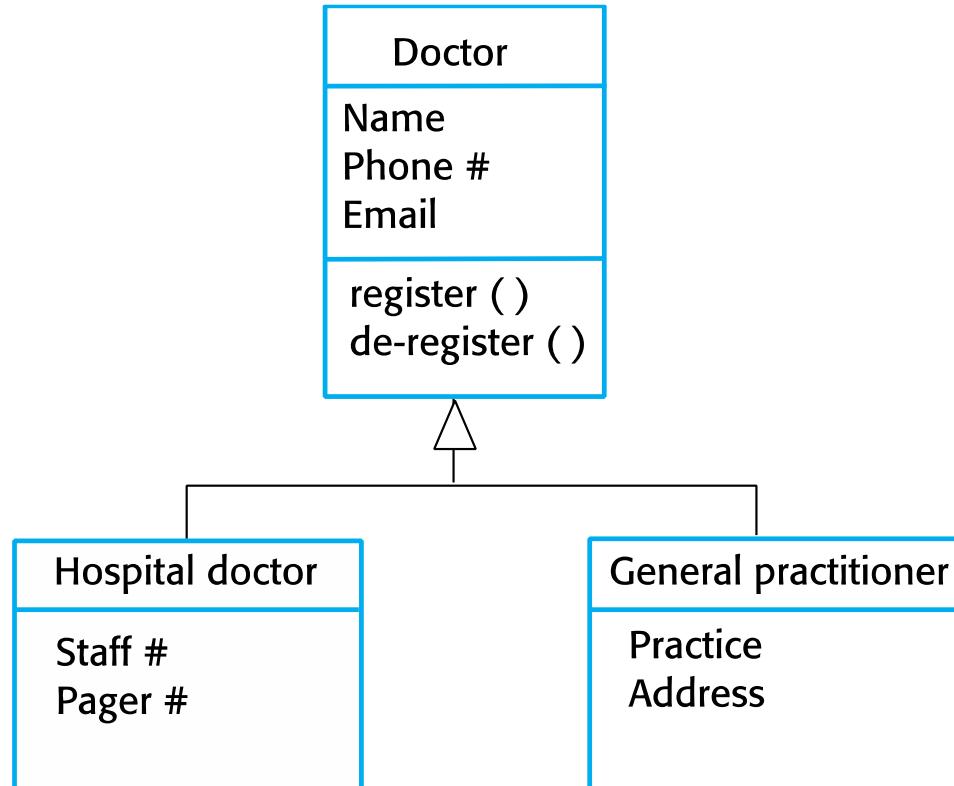
A generalization hierarchy

Software Engineering

Ian Sommerville



A generalization hierarchy with added detail

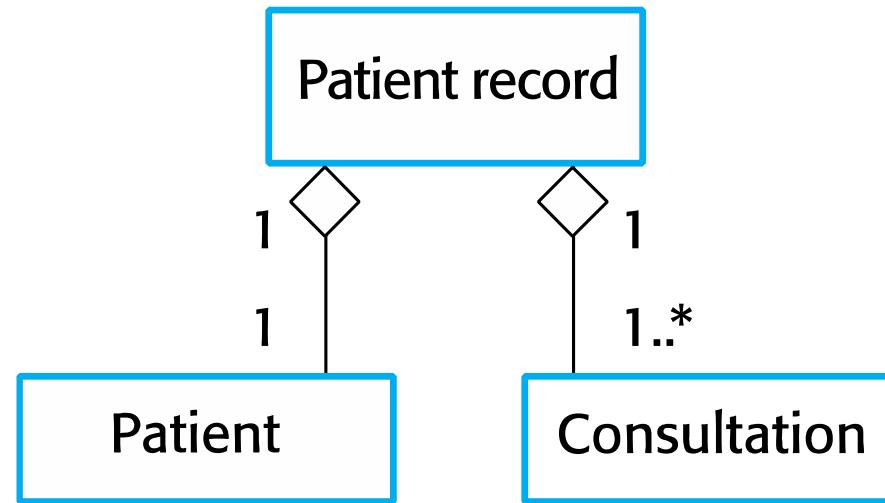


Object class aggregation models



- ✧ 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.

The aggregation association





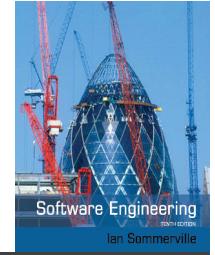
Behavioral models

Behavioral models



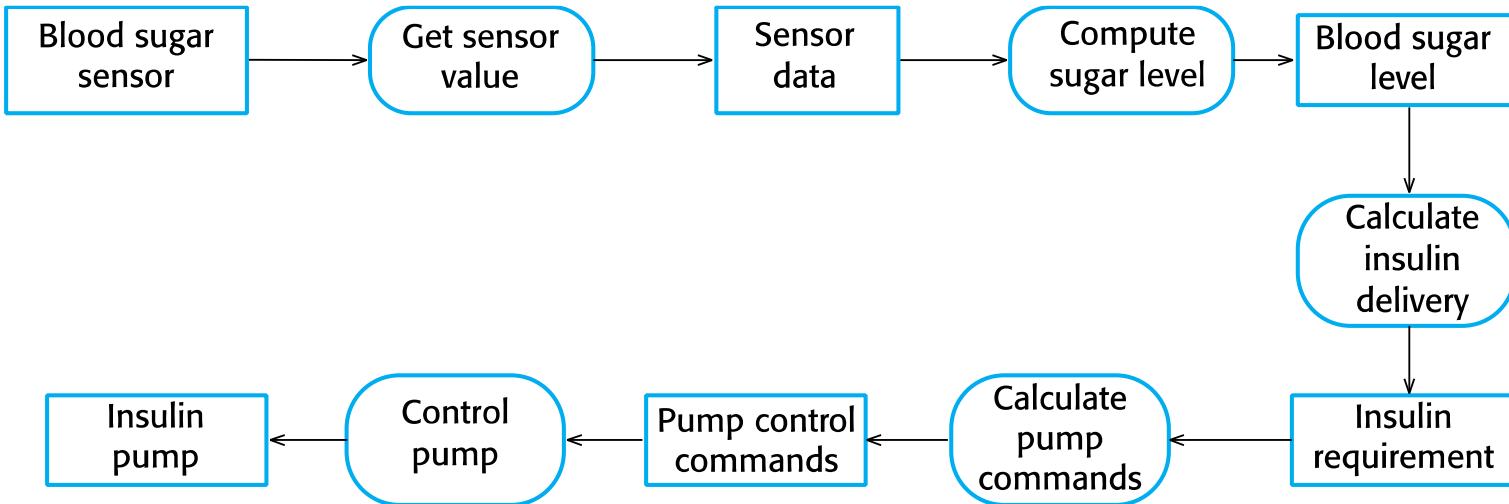
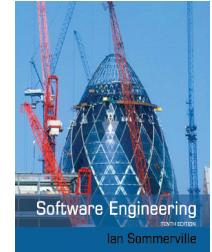
- ✧ Behavioral models are models of the dynamic behavior of a system as it is executing. They show what happens or what is supposed to happen when a system responds to a stimulus from its environment.
- ✧ You can think of these stimuli as being of two types:
 - **Data** Some data arrives that has to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

Data-driven modeling

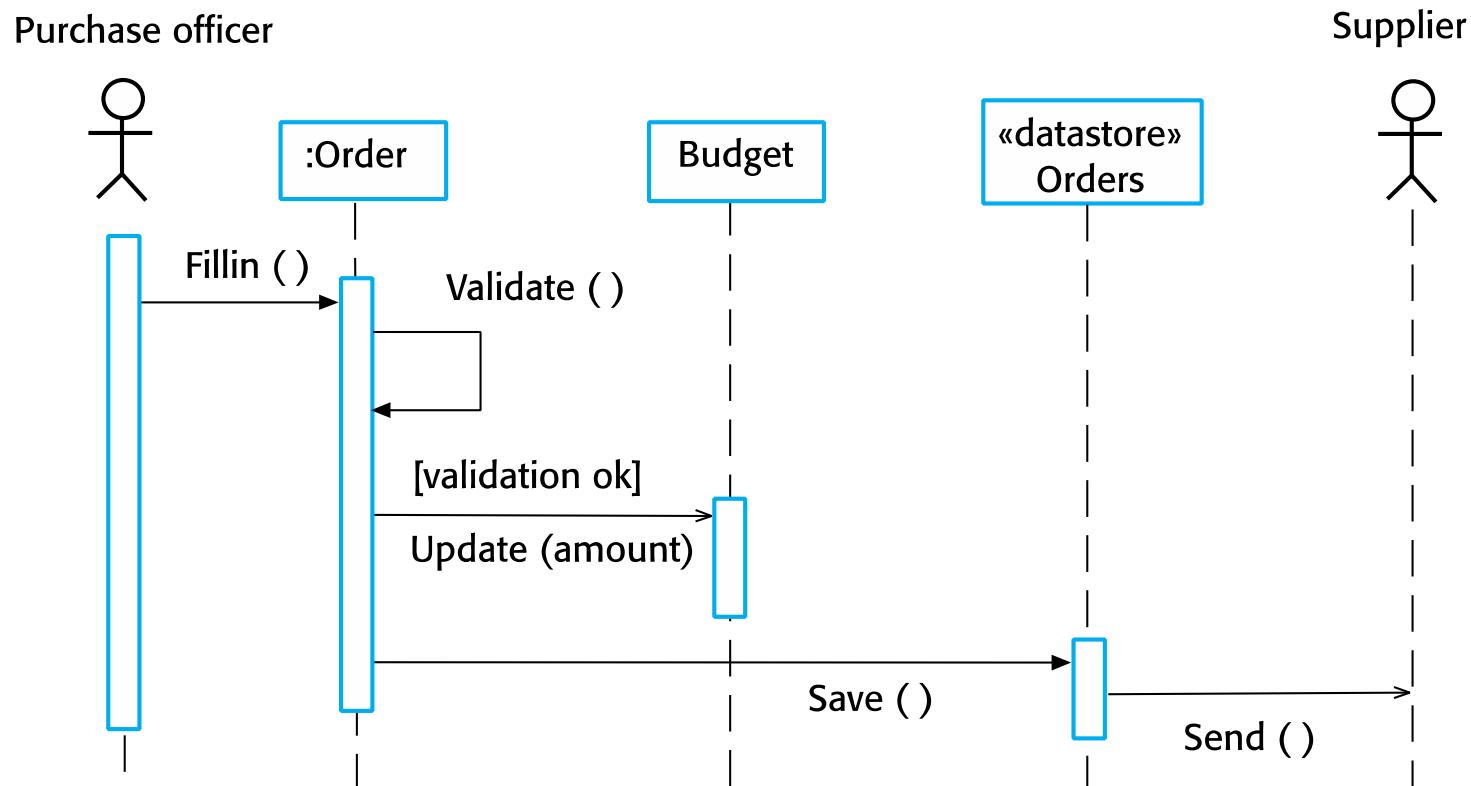


- ✧ Many business systems are data-processing systems that are primarily driven by data. They are controlled by the data input to the system, with relatively little external event processing.
- ✧ Data-driven models show the sequence of actions involved in processing input data and generating an associated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

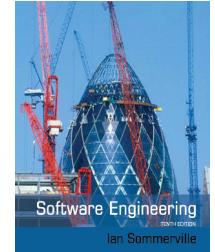
An activity model of the insulin pump's operation



Order processing

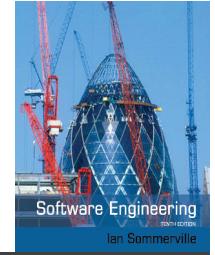


Event-driven modeling



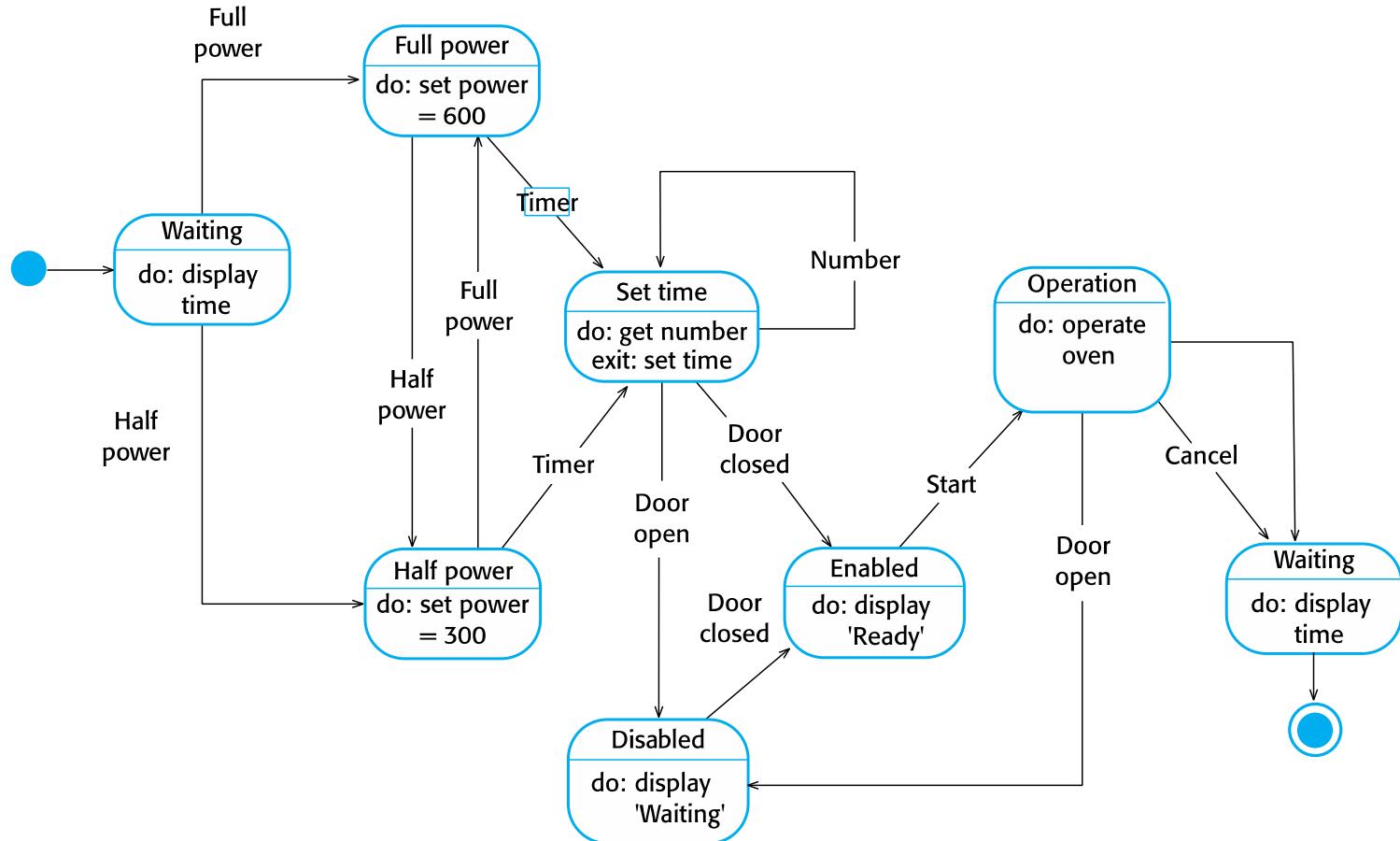
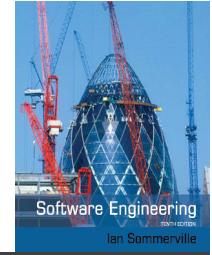
- ✧ Real-time systems are often event-driven, with minimal data processing. For example, a landline phone switching system responds to events such as 'receiver off hook' by generating a dial tone.
- ✧ Event-driven modeling shows how a system responds to external and internal events.
- ✧ It is based on the assumption 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

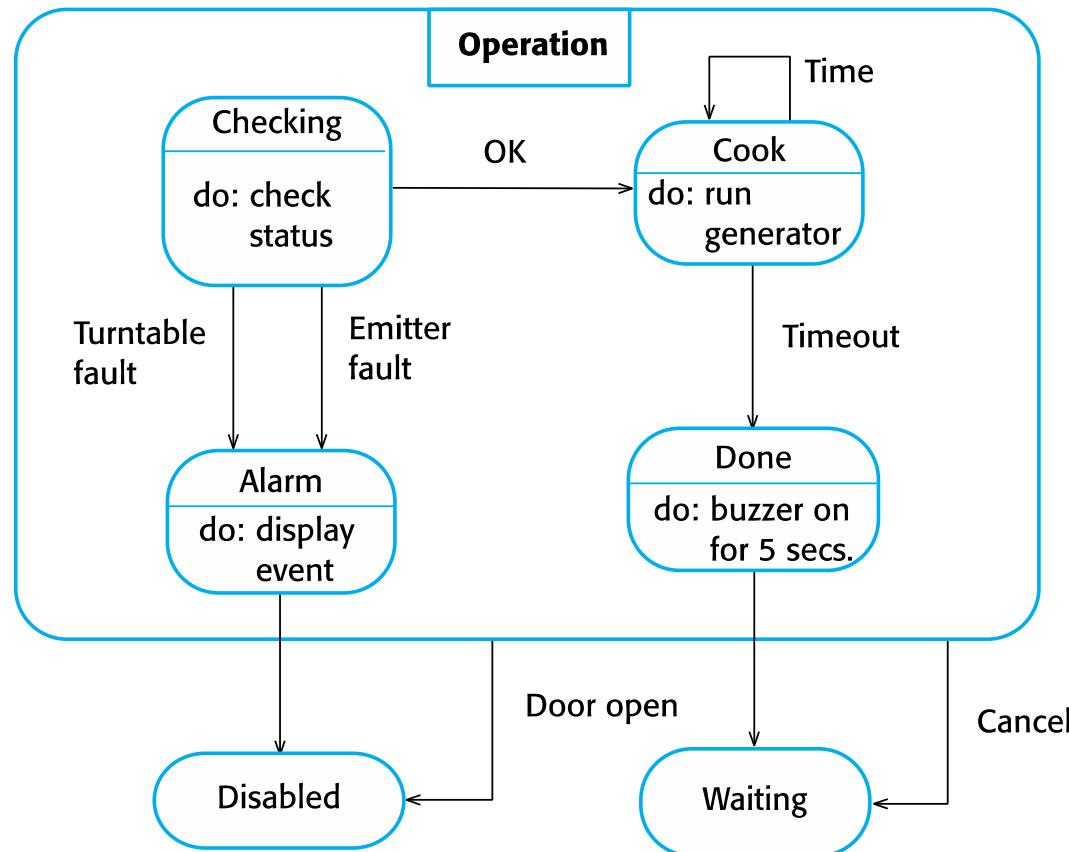
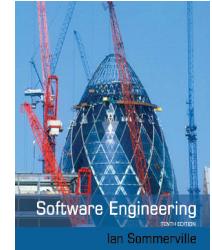


- ✧ These model the behaviour of the system in response to external and internal events.
- ✧ They show the system's responses to stimuli so are often used for modelling real-time systems.
- ✧ State machine models show system states as nodes and events as arcs between these nodes. When an event occurs,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



Microwave oven operation





Software Engineering

Ian Sommerville

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.



Software Engineering

Ian Sommerville

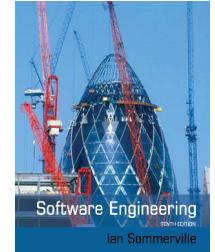
Model-driven engineering

Model-driven engineering



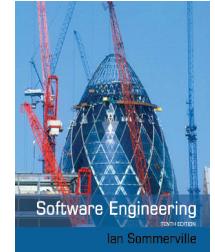
- ✧ Model-driven engineering (MDE) is an approach to software development where models rather than programs are the principal outputs of the development process.
- ✧ The programs that execute on a hardware/software platform are then generated automatically from the models.
- ✧ Proponents of MDE argue 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 or not 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



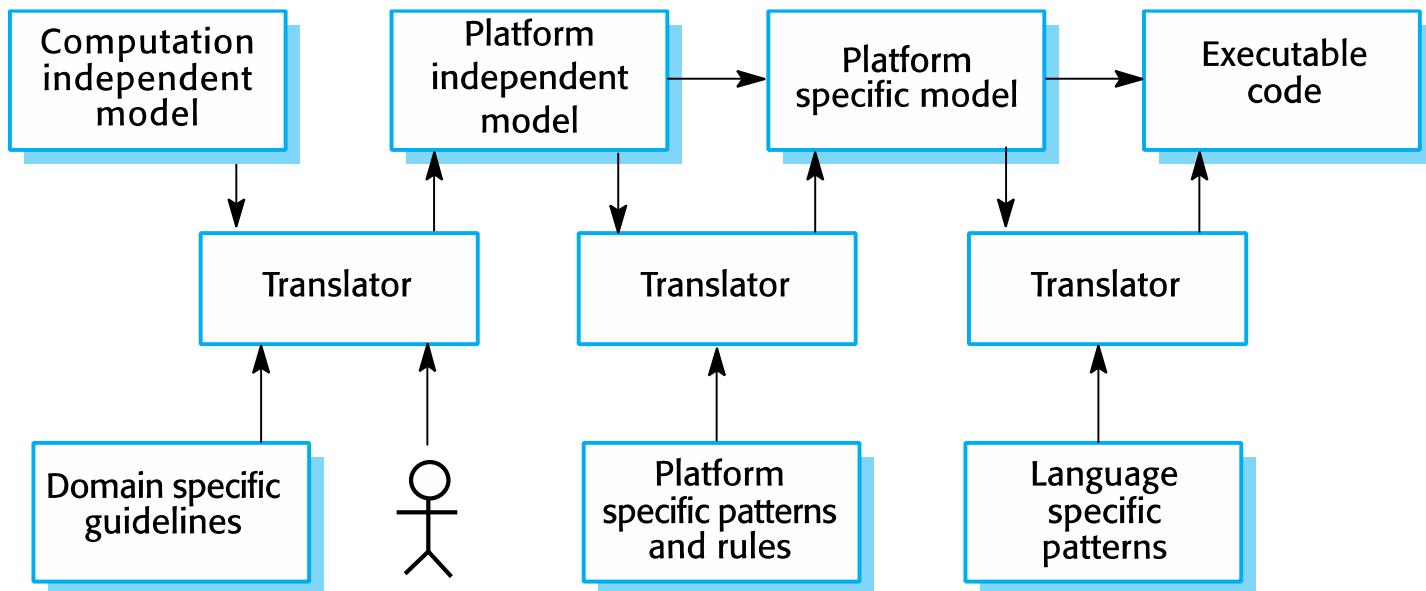
- ✧ Model-driven architecture (MDA) was the precursor of more general model-driven engineering
- ✧ MDA is a model-focused approach to software design and implementation that uses a subset of UML models to describe a system.
- ✧ Models at different levels of abstraction are created. From a high-level,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



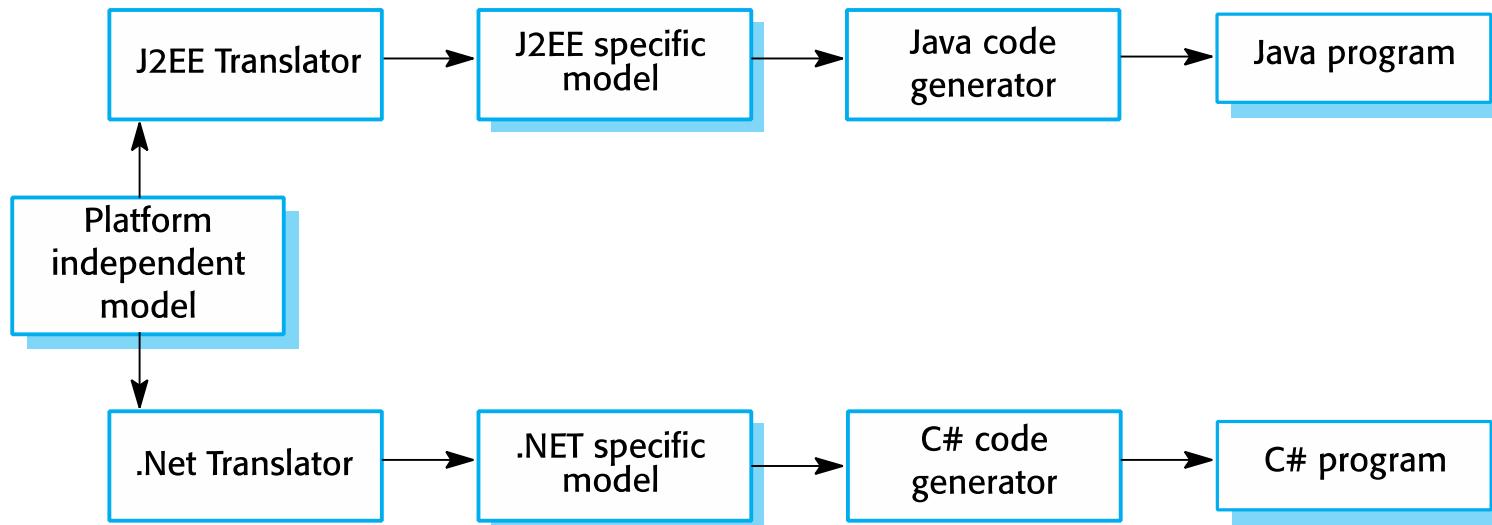
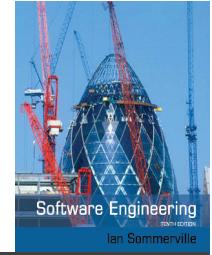
Types of model

- ✧ 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



Agile methods and MDA



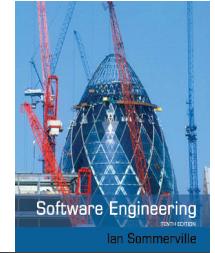
- ✧ The developers of MDA claim that it is intended to support an iterative approach to development and so can be used within agile methods.
- ✧ The notion of extensive up-front modeling contradicts the fundamental ideas in the agile manifesto and I suspect that few agile developers feel comfortable with model-driven engineering.
- ✧ If transformations can be completely automated and a complete program generated from a PIM, then, in principle, MDA could be used in an agile development process as no separate coding would be required.

Adoption of MDA



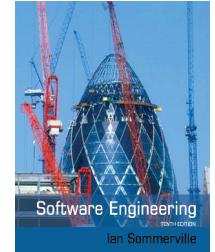
- ✧ A range of factors has limited the adoption of MDE/MDA
- ✧ Specialized tool support is required to convert models from one level to another
- ✧ There is limited tool availability and organizations may require tool adaptation and customisation to their environment
- ✧ For the long-lifetime systems developed using MDA, companies are reluctant to develop their own tools or rely on small companies that may go out of business

Adoption of MDA



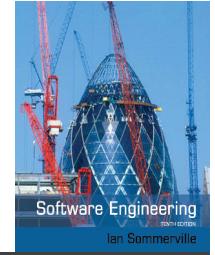
- ✧ Models are a good way of facilitating discussions about a software design. However the abstractions that are useful for discussions may not be the right abstractions for implementation.
- ✧ For most complex systems, implementation is not the major problem – requirements engineering, security and dependability, integration with legacy systems and testing are all more significant.

Adoption of MDA



- ✧ The arguments for platform-independence are only valid for large, long-lifetime systems. For software products and information systems, the savings from the use of MDA are likely to be outweighed by the costs of its introduction and tooling.
- ✧ The widespread adoption of agile methods over the same period that MDA was evolving has diverted attention away from model-driven approaches.

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.

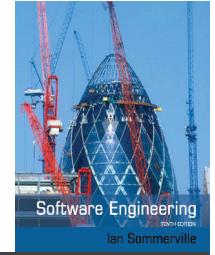


Software Engineering

Ian Sommerville

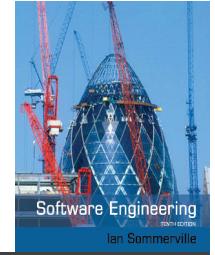
Chapter 6 – Architectural Design

Topics covered



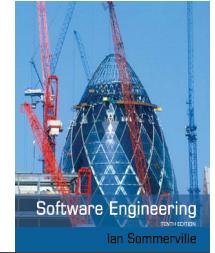
- ✧ Architectural design decisions
- ✧ Architectural views
- ✧ Architectural patterns
- ✧ Application architectures

Architectural design



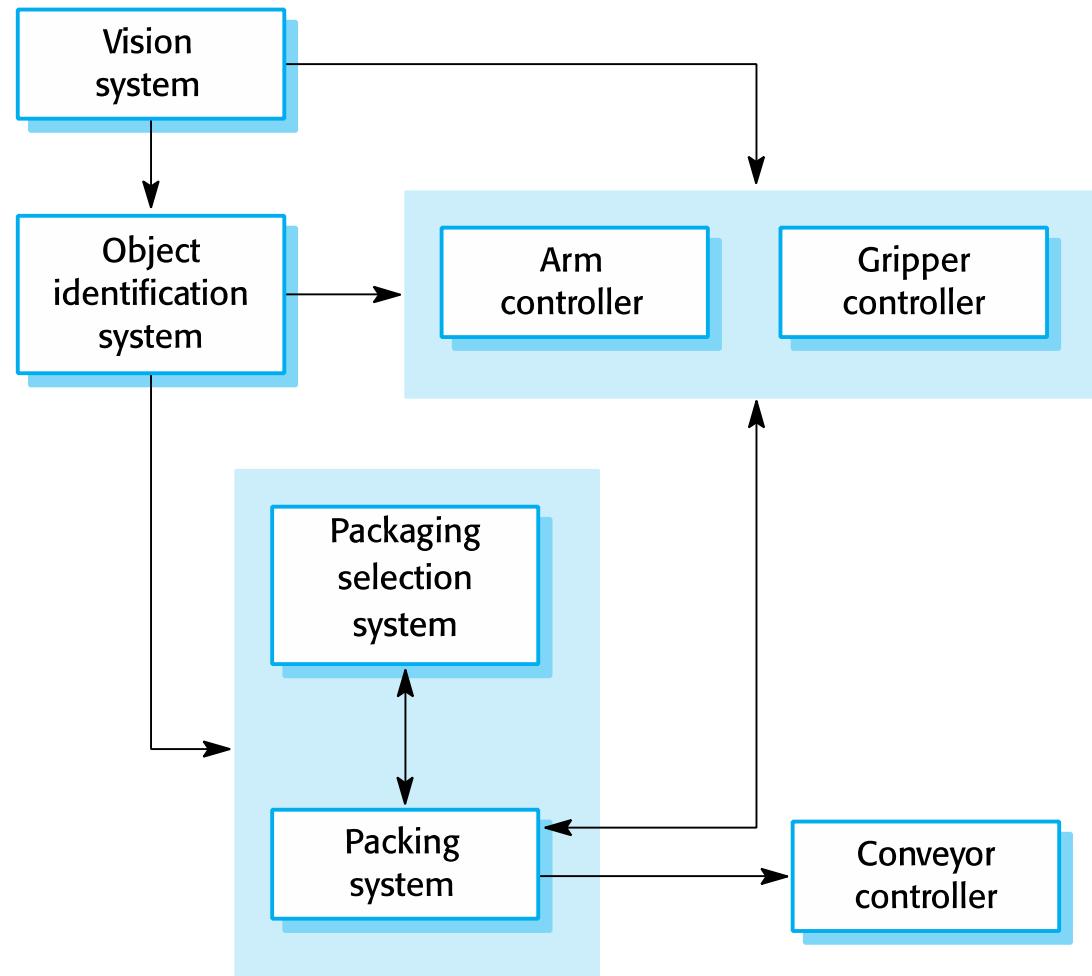
- ✧ Architectural design is concerned with understanding how a software system should be organized and designing the overall structure of that system.
- ✧ Architectural design is the critical link between design and requirements engineering, as it identifies the main structural components in a system 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

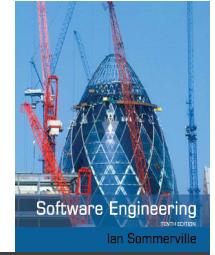


- ✧ It is generally accepted that an early stage of agile processes is to design an overall systems architecture.
- ✧ Refactoring the system architecture is usually expensive because it affects so many components in the system

The architecture of a packing robot control system

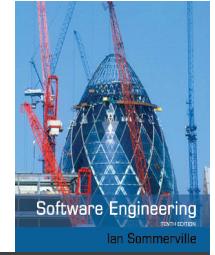


Architectural abstraction



- ✧ 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.

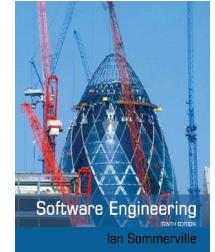
✧ System analysis

- 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

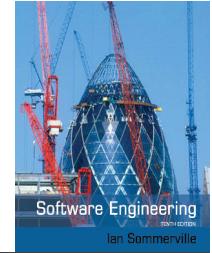


- ✧ Simple, informal block diagrams showing entities and relationships are the most frequently used method for documenting software architectures.
- ✧ But these have been criticised because they lack semantics, do not show the types of relationships between entities nor the visible properties of entities 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.



Software Engineering

Ian Sommerville

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?

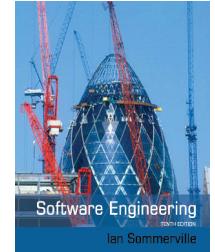
How should the architecture of the system be documented?

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



Software Engineering
Ian Sommerville

✧ 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.

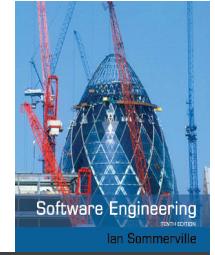


Software Engineering

Ian Sommerville

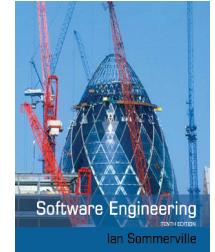
Architectural views

Architectural views

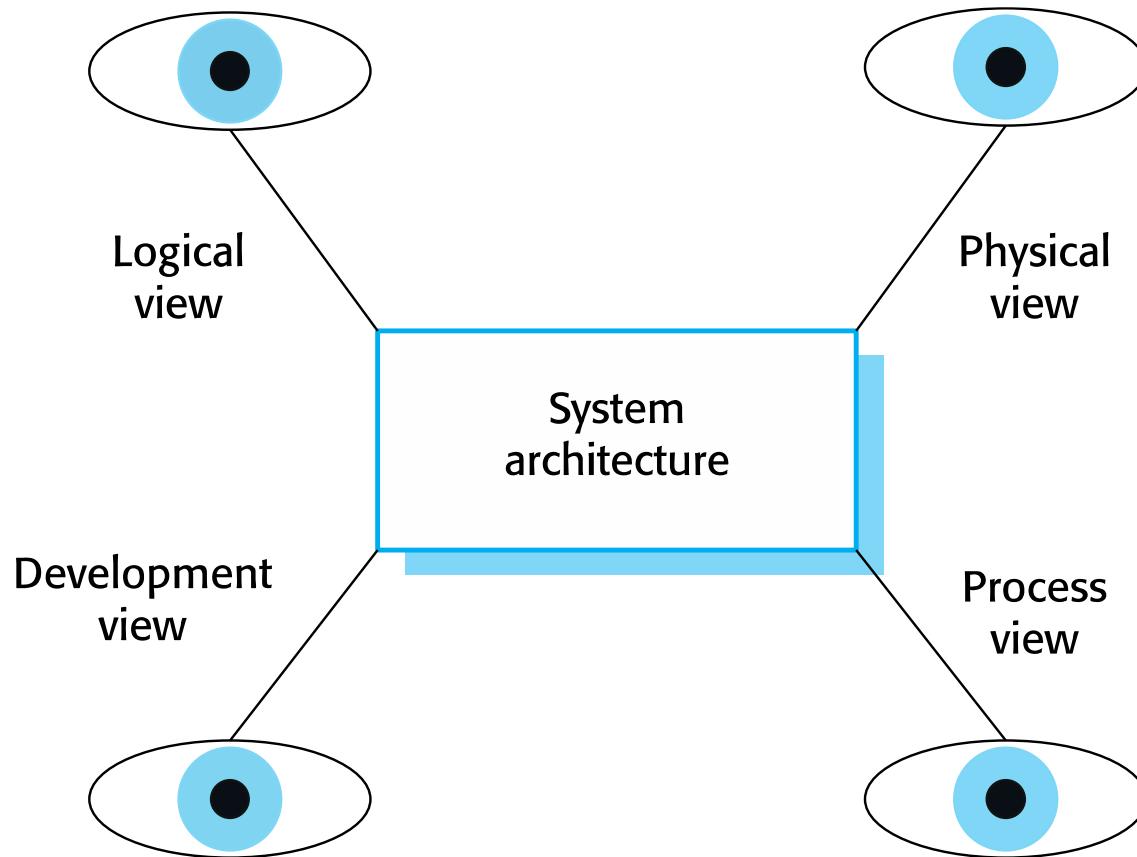


- ✧ What views or perspectives are useful when designing and documenting a system's architecture?
- ✧ What notations should be used for describing architectural models?
- ✧ Each architectural model only shows one view or perspective of the system.
 - It might show how a system is decomposed into modules, 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



Software Engineering
Ian Sommerville

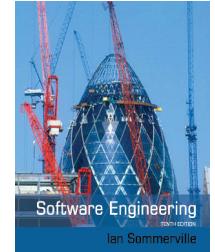


4 + 1 view model of software architecture



- ✧ 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

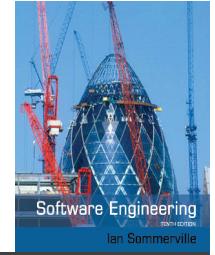


Software Engineering

Ian Sommerville

Architectural patterns

Architectural patterns



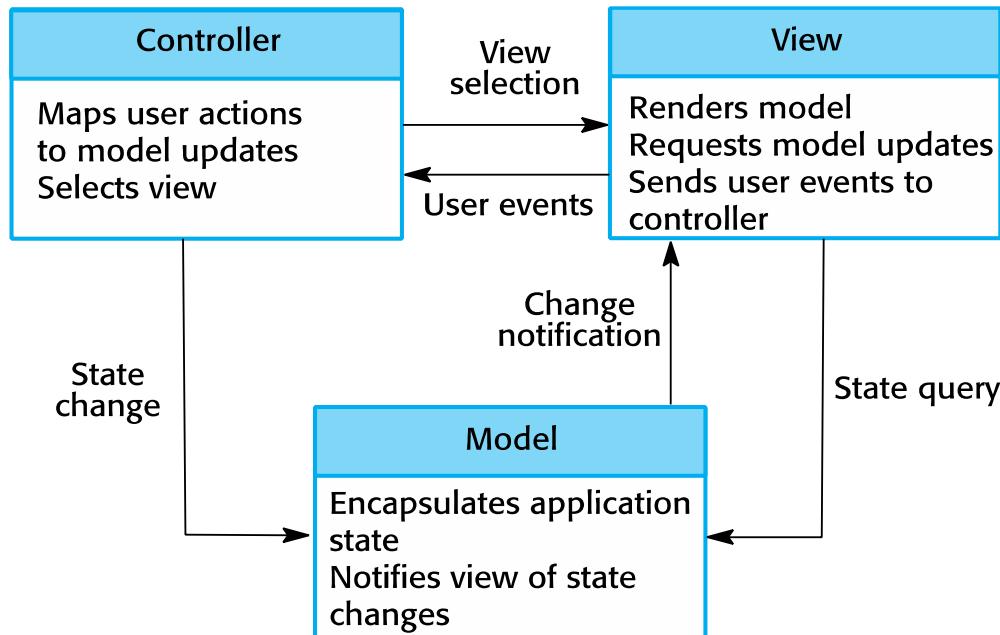
- ✧ Patterns are a means of representing, sharing and reusing knowledge.
- ✧ An architectural pattern is a stylized description of good design practice, which has been tried and tested in different environments.
- ✧ Patterns should include information about when they are and when they are not useful.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

The Model-View-Controller (MVC) pattern

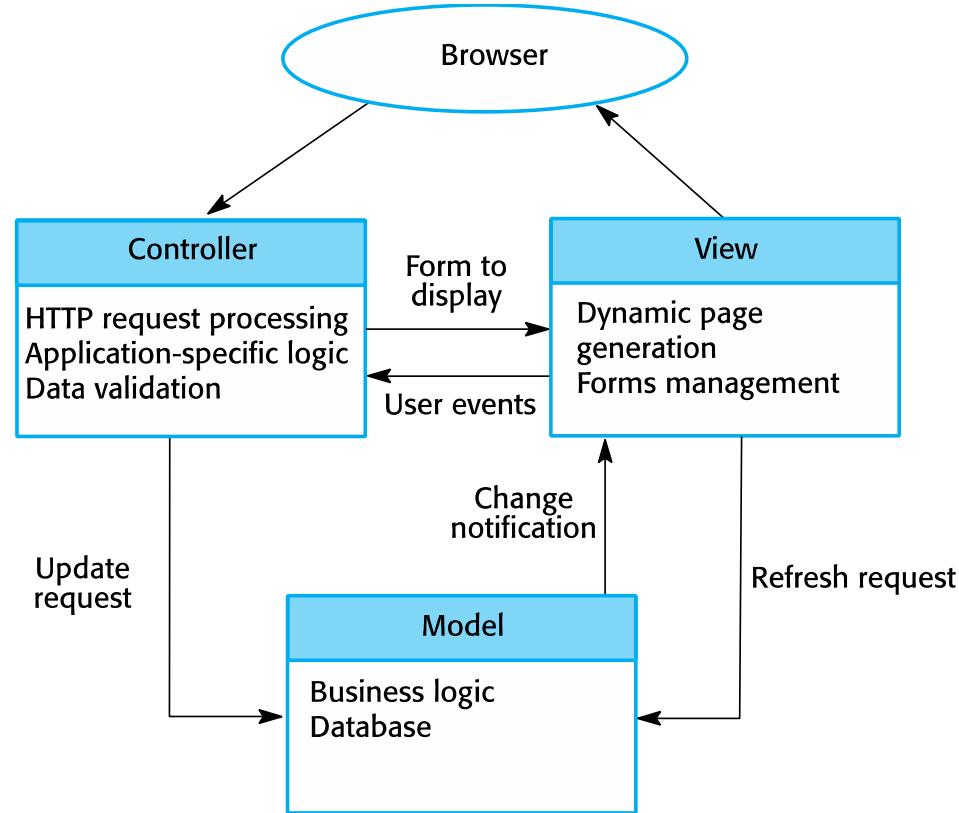


Name	MVC (Model-View-Controller)
Description	Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines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



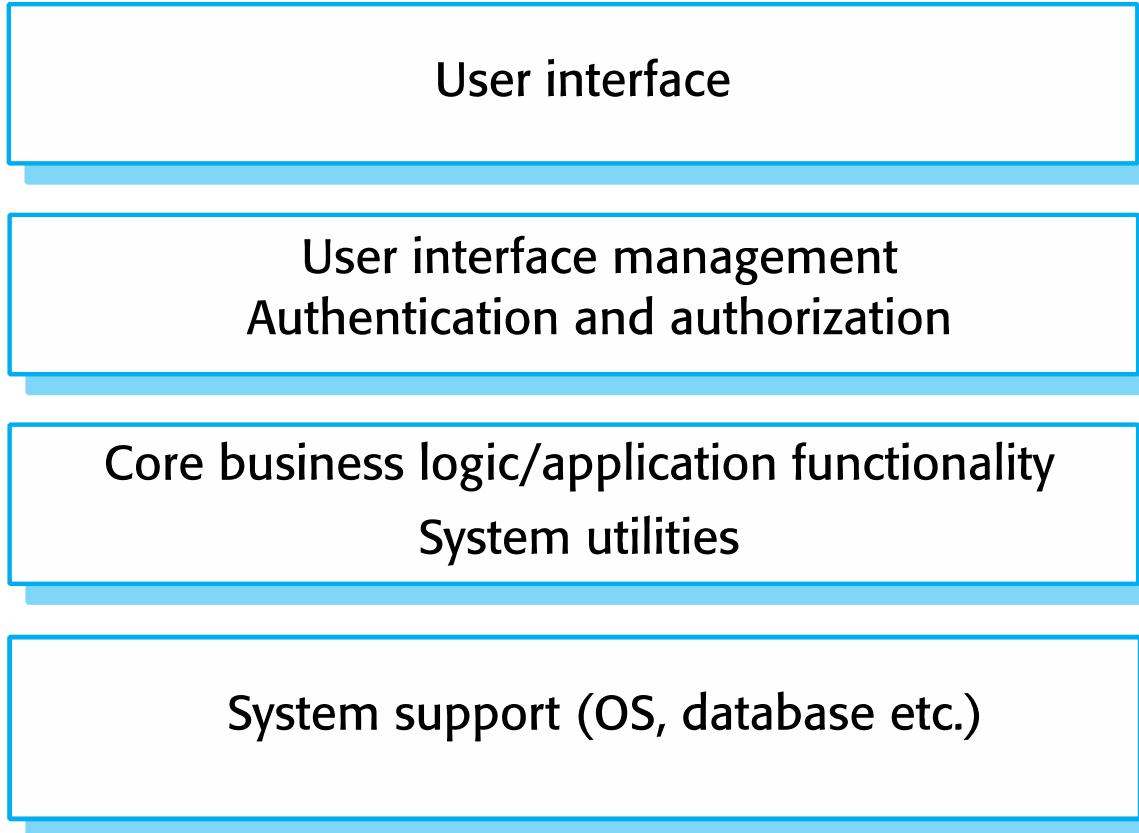
- ✧ Used to model the interfacing of sub-systems.
- ✧ Organises the system into a set of layers (or abstract machines) each of which provide a set of services.
- ✧ Supports the incremental development of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected.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

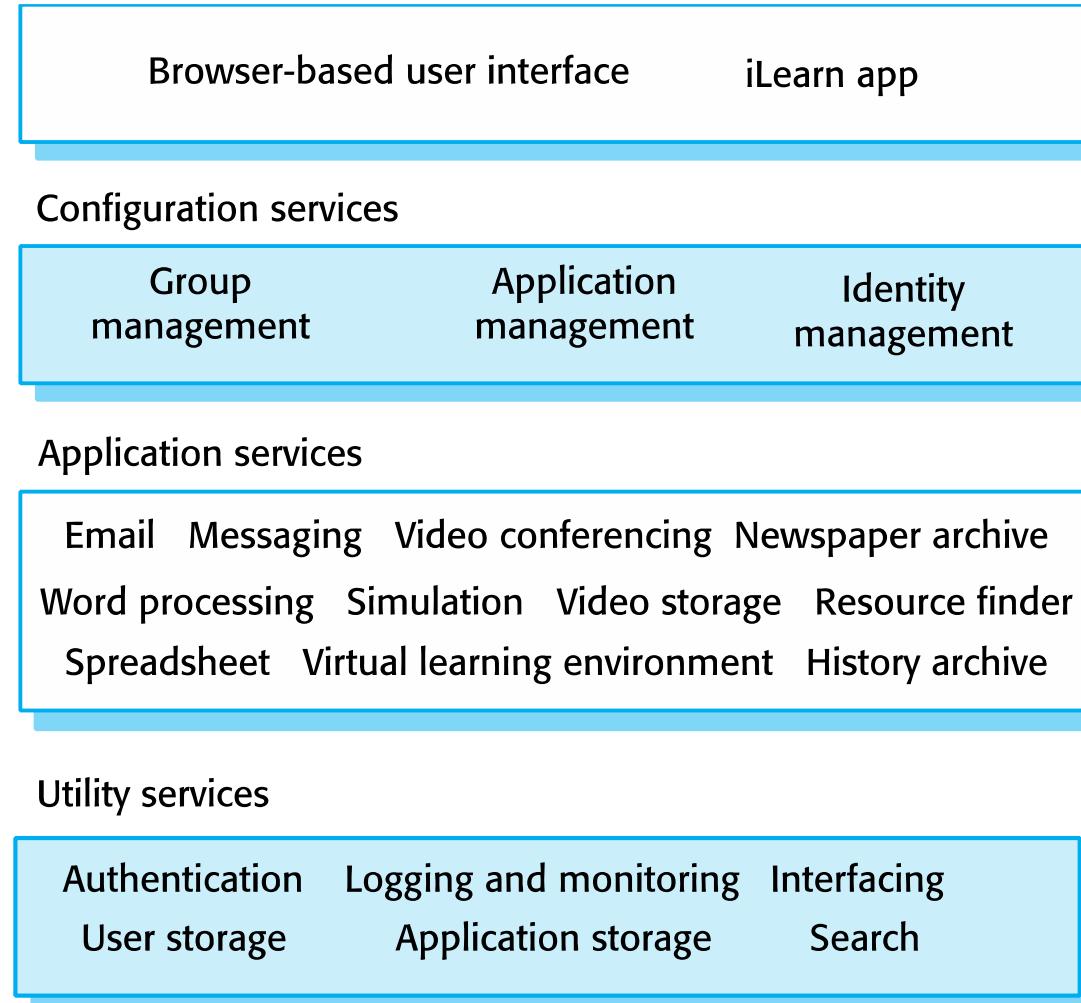




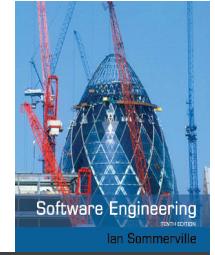
Software Engineering

Ian Sommerville

The architecture of the iLearn system



Repository architecture



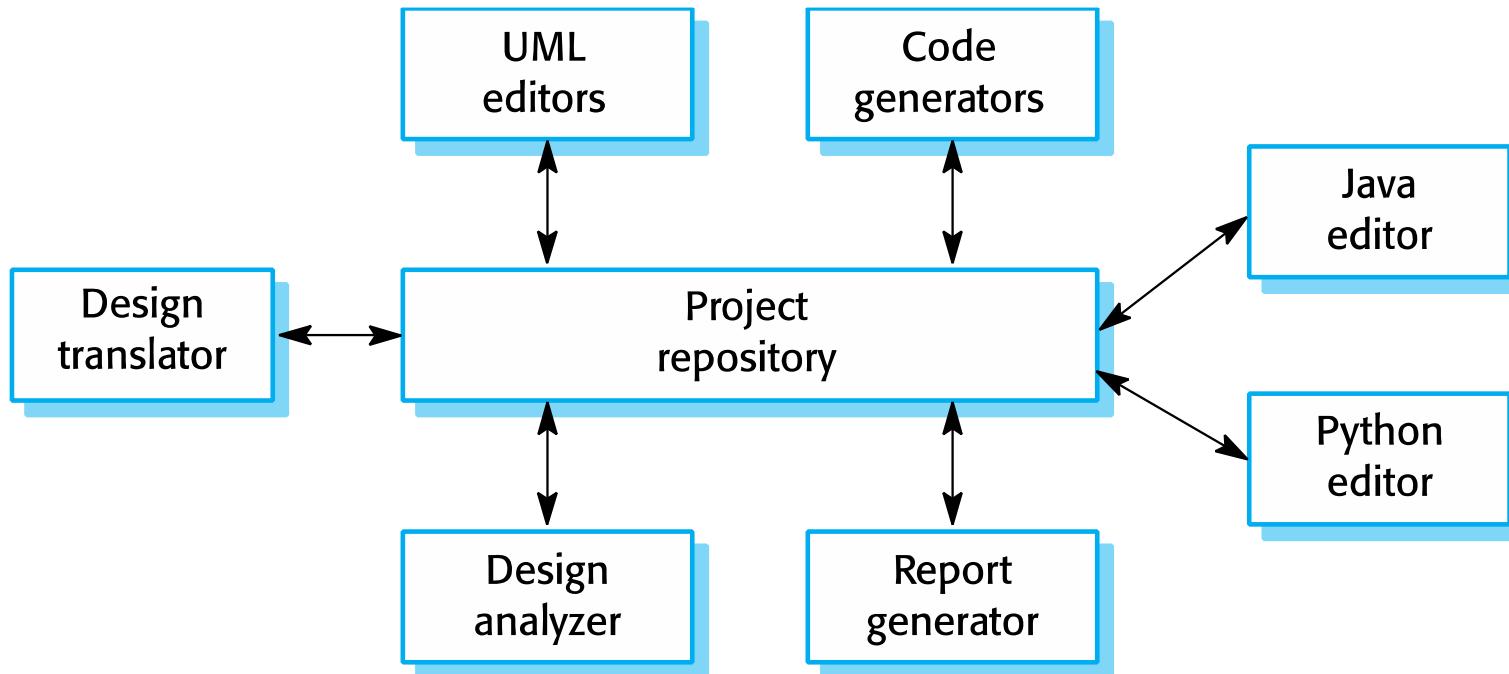
- ✧ Sub-systems must exchange data. This may be done in two ways:
 - Shared data is held in a central database or repository and may be accessed by all sub-systems;
 - Each sub-system maintains its own database and passes data explicitly to other sub-systems.
- ✧ When large amounts of data are to be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

The Repository pattern

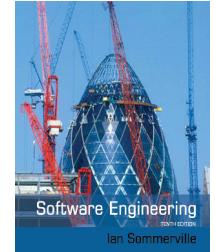


Name	Repository
Description	All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.
Example	Figure 6.9 is an example of an IDE where the components use a repository of system design information. 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.

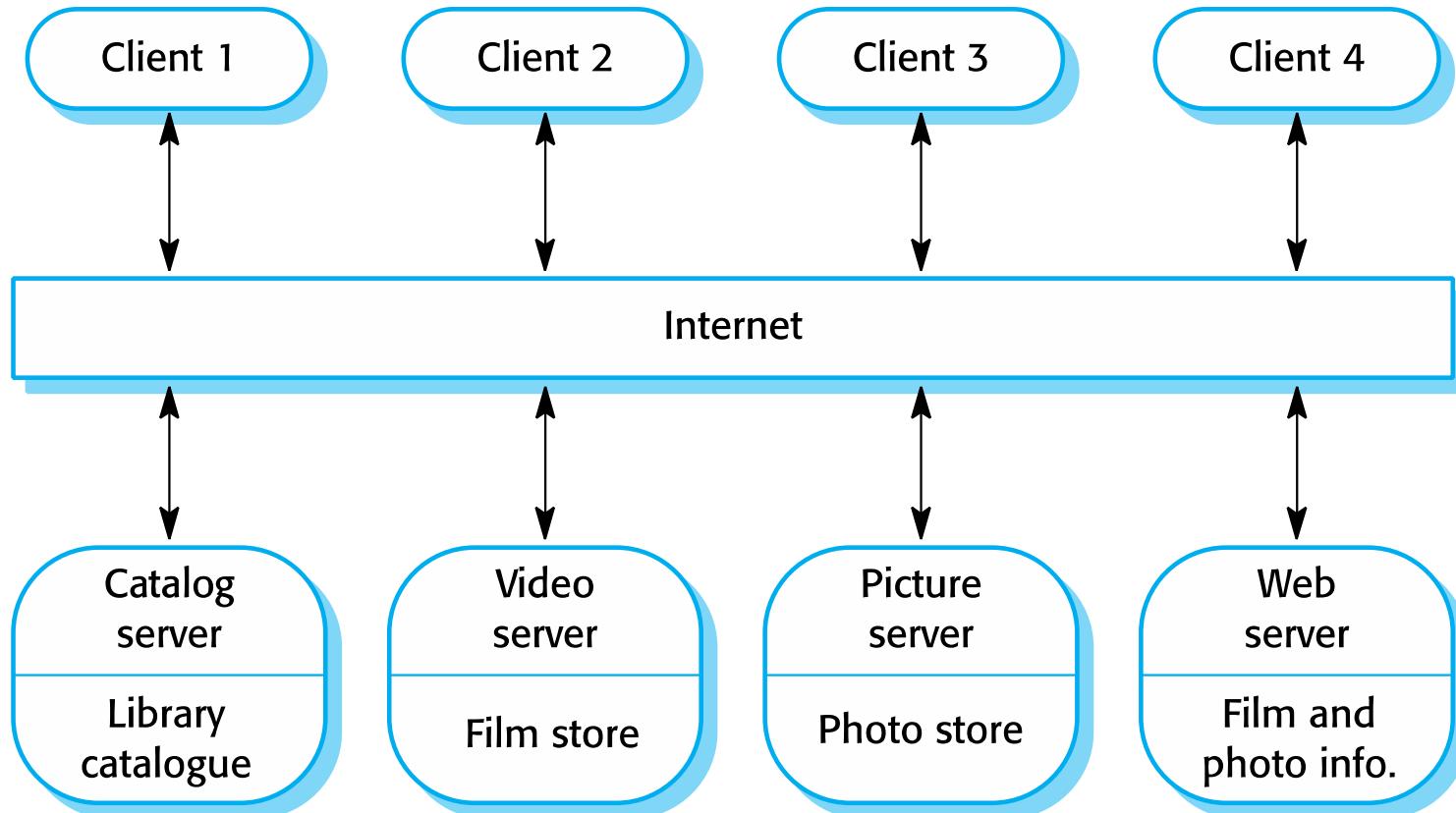


The Client–server pattern

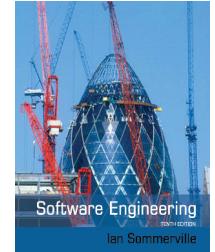
Name	Client-server
Description	In a client–server architecture, the functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.
Example	Figure 6.11 is an example of a film and video/DVD 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.



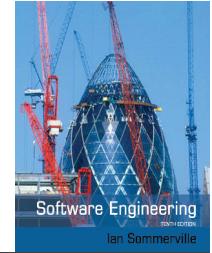
A client–server architecture for a film library



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.

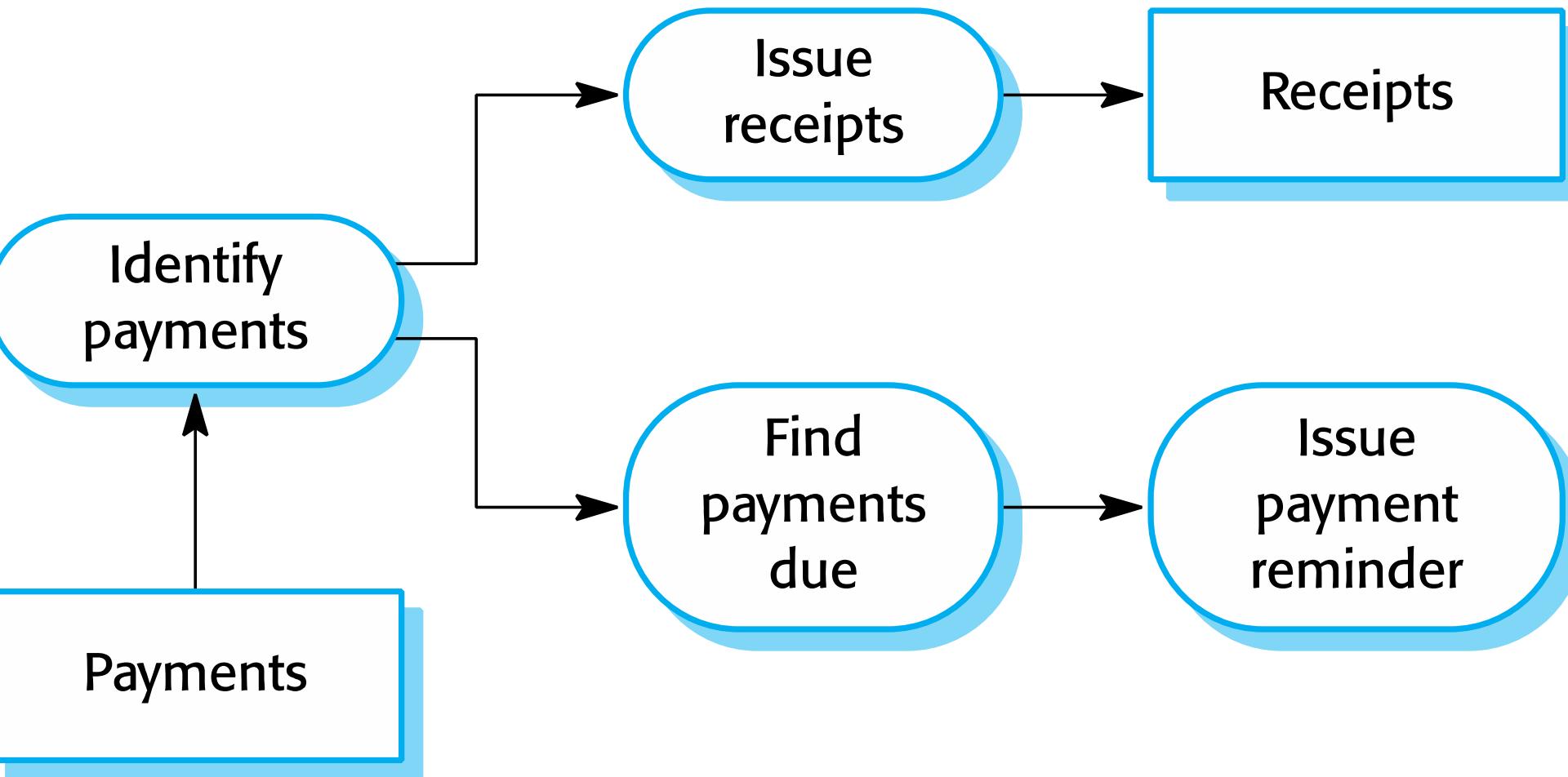


The pipe and filter pattern

Software Engineering
Ian Sommerville

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.

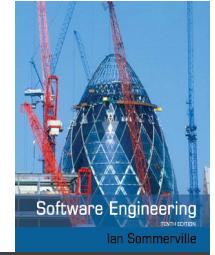
An example of the pipe and filter architecture used in a payments system





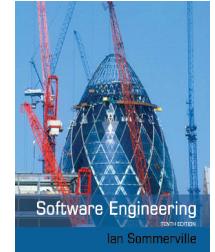
Application architectures

Application architectures

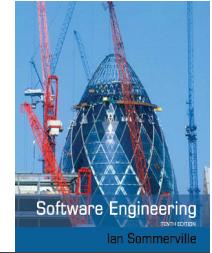


- ✧ Application systems are designed to meet an organisational need.
- ✧ As businesses have much in common, their application systems also tend to have a common architecture that reflects the application requirements.
- ✧ A generic application architecture is an architecture for a type of software system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

Use of application architectures



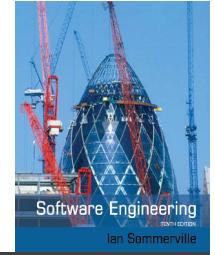
- ✧ As a starting point for architectural design.
- ✧ As a design checklist.
- ✧ As a way of organising the work of the development team.
- ✧ As a means of assessing components for reuse.
- ✧ As a vocabulary for talking about application types.



Examples of application types

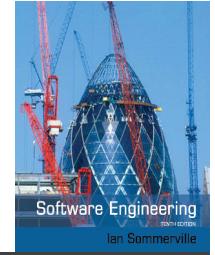
- ✧ Data processing applications
 - Data driven applications that process data in batches without explicit user intervention during the processing.
- ✧ Transaction processing applications
 - Data-centred applications that process user requests and update information in a system database.
- ✧ Event processing 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



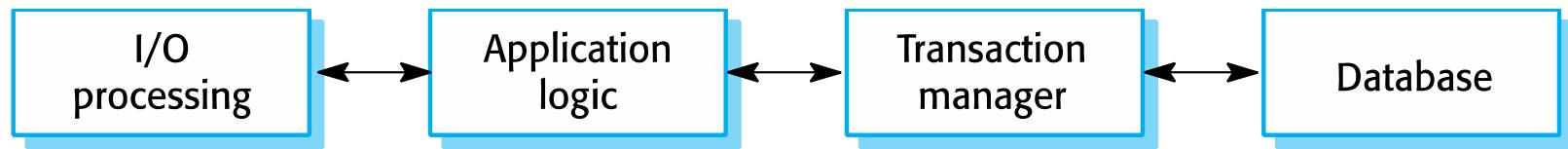
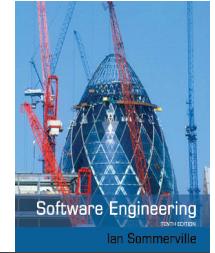
- ✧ Two very widely used generic application architectures are transaction processing systems and language processing systems.
- ✧ Transaction processing systems
 - E-commerce systems;
 - Reservation systems.
- ✧ Language processing systems
 - Compilers;
 - Command interpreters.

Transaction processing systems

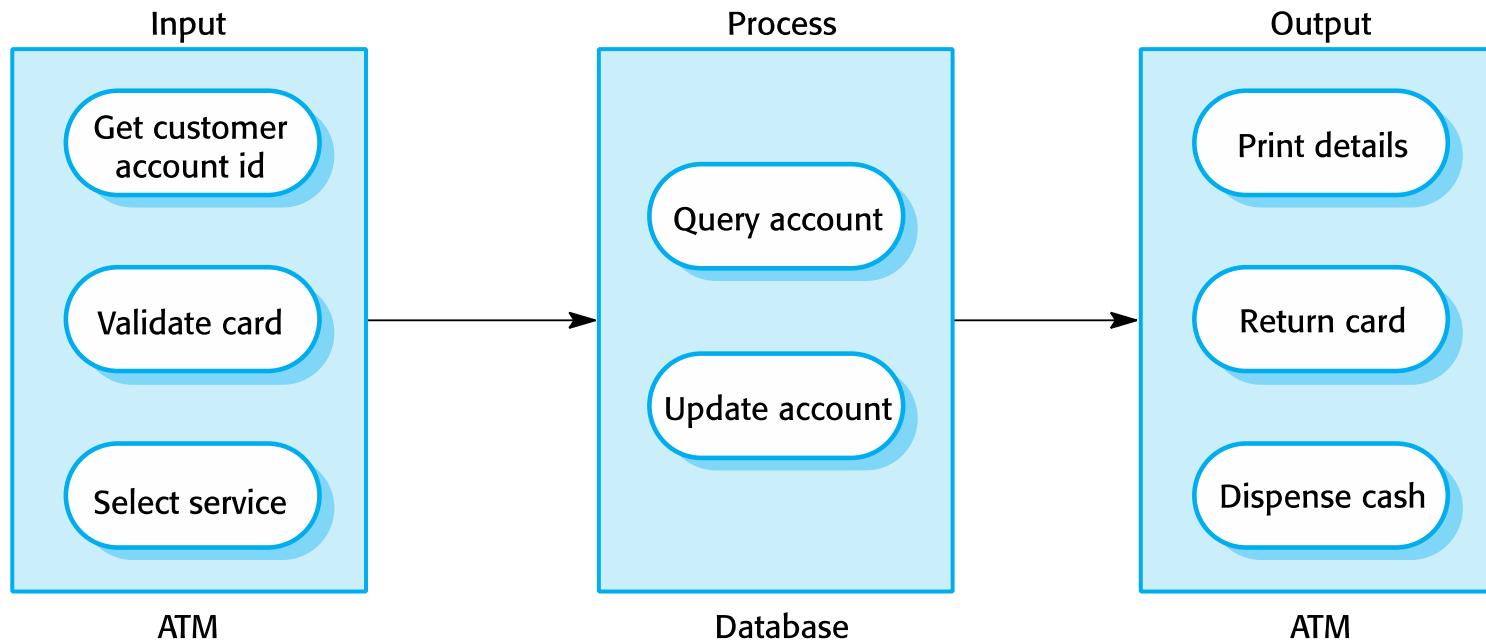


- ✧ Process user requests for information from a database or requests to update the database.
- ✧ From a user perspective a transaction is:
 - Any coherent sequence of operations that satisfies a goal;
 - For example - find the times of flights from London to Paris.
- ✧ Users make asynchronous requests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

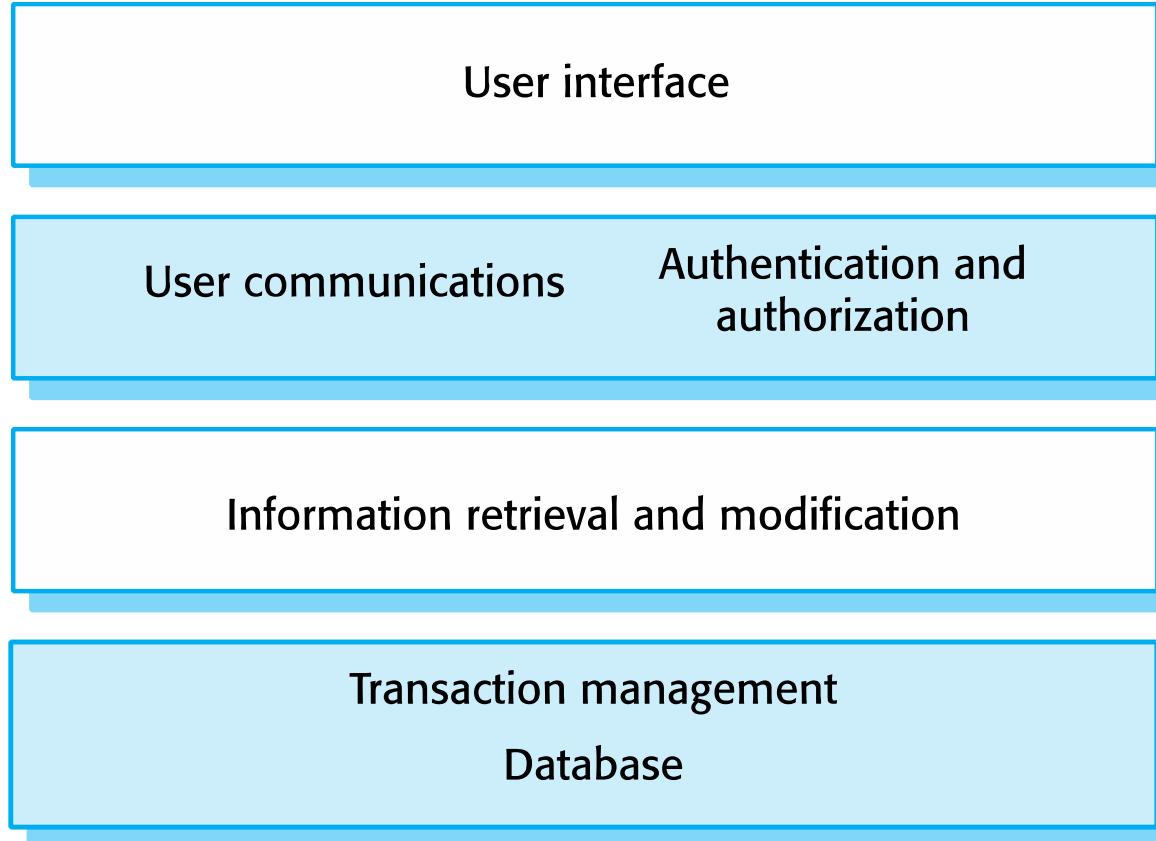


- ✧ Information systems have a generic architecture that can be organised as a layered architecture.
- ✧ These are transaction-based systems as interaction with these systems generally involves database transactions.
- ✧ Layers include:
 - The user interface
 - User communications
 - Information retrieval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



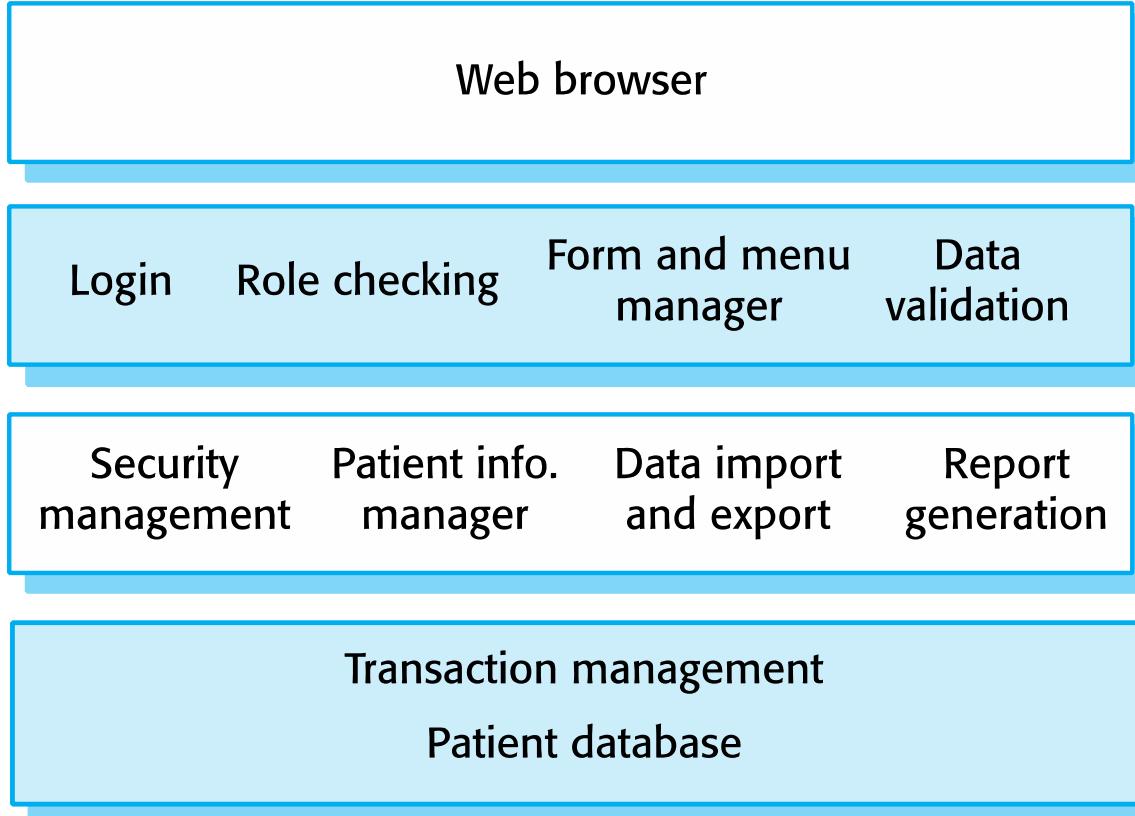
Layered information system architecture

Software Engineering
Ian Sommerville





The architecture of the Mentcare system

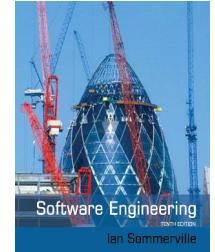


Web-based information systems



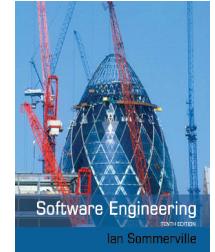
- ✧ Information and resource management systems are now usually web-based systems where the user interfaces are implemented using a web browser.
- ✧ For example, e-commerce systems are Internet-based resource management systems that accept electronic orders for goods or services and then arrange delivery 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



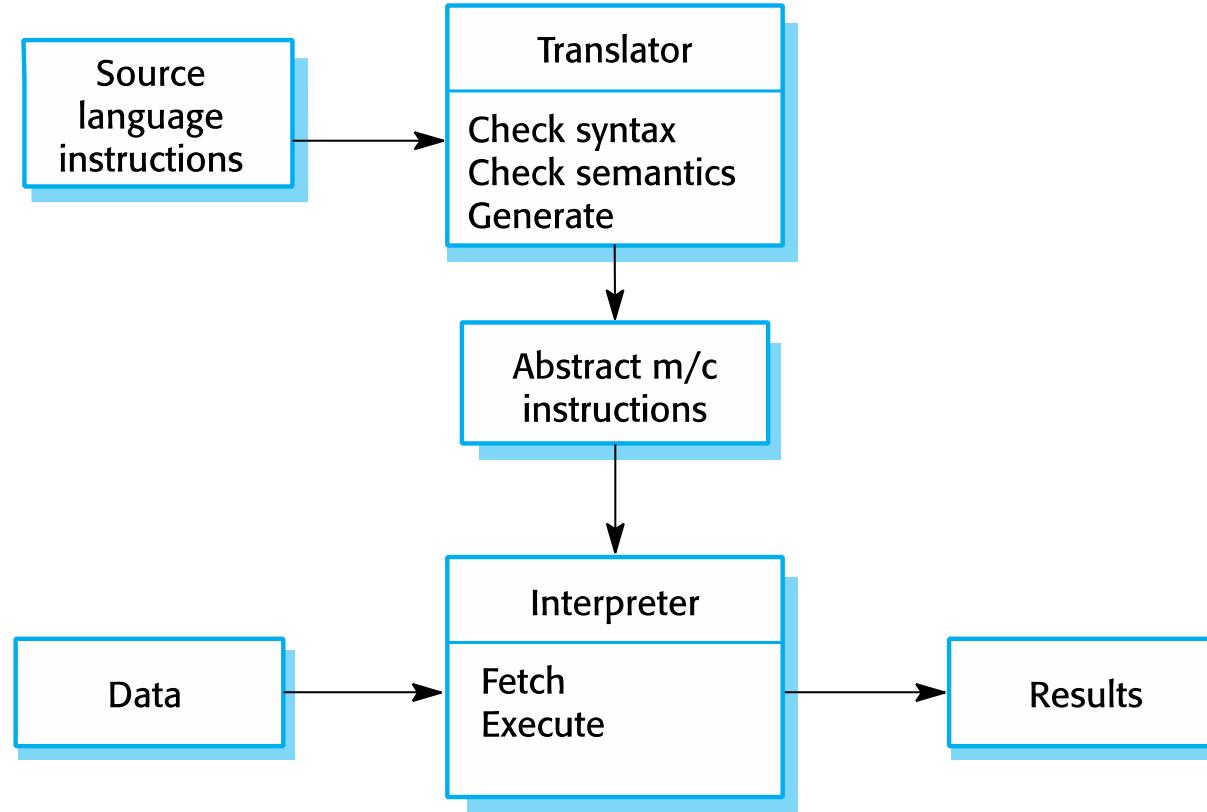
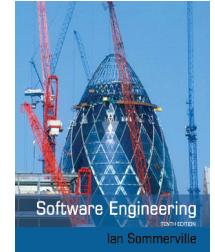
- ✧ These systems are often implemented as multi-tier client server/architectures (discussed in Chapter 17)
 - The web server is responsible for all user communications, with the user interface implemented using a web browser;
 - The application server is responsible for implementing application-specific 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

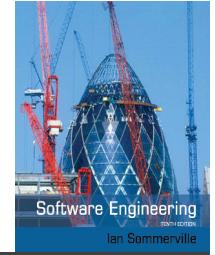


- ✧ Accept a natural or artificial language as input and generate some other representation of that language.
- ✧ May include an interpreter to act on the instructions in the language that is being processed.
- ✧ Used in situations where the easiest way to solve a problem is to describe an algorithm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

The architecture of a language processing system

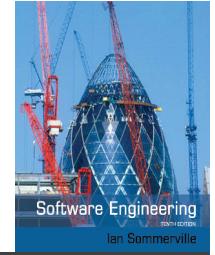


Compiler components



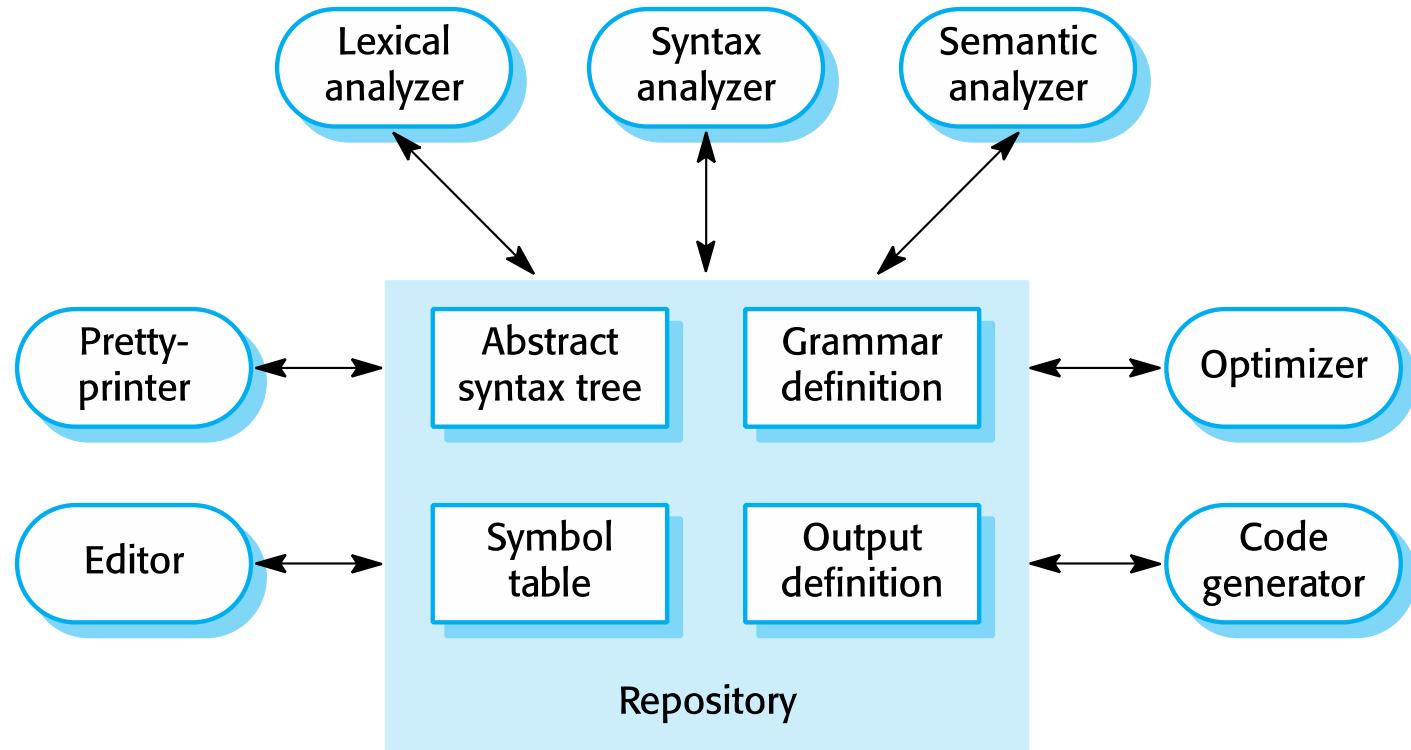
- ✧ A lexical analyzer, which takes input language tokens and converts them to an internal form.
- ✧ A symbol table, which holds information about the names of entities (variables, class names, object names, etc.) used in the text that is being translated.
- ✧ A syntax analyzer, which checks the syntax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

Compiler components

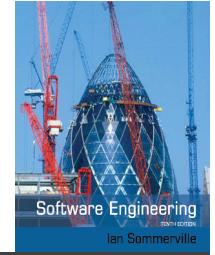


- ✧ A semantic analyzer that uses information from the syntax tree and the symbol table to check the semantic correctness of the input language text.
- ✧ A code generator that ‘walks’ the syntax tree and generates abstract machine code.

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



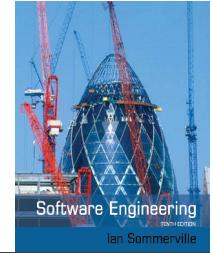
- ✧ Models of application systems architectures help us understand and compare applications, validate application system designs and assess large-scale components for reuse.
- ✧ Transaction processing systems are interactive systems that allow information in a database to be remotely accessed and modified 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.



Software Engineering

Ian Sommerville

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

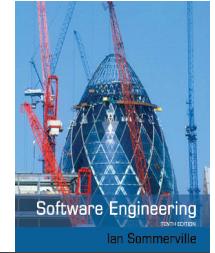


- ✧ In a wide range of domains, it is now possible to buy off-the-shelf systems (COTS) that can be adapted and tailored to the users' requirements.
 - For example, if you want to implement a medical records system, you can buy a package that is already used in hospitals. It can be cheaper and faster 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



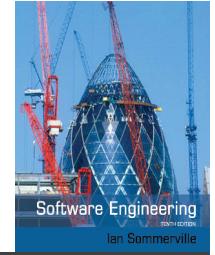
- ✧ There are a variety of different object-oriented design processes that depend on the organization using the process.
- ✧ Common activities in these processes include:
 - Define the context and modes of use of the system;
 - Design the system architecture;
 - Identify the principal system objects;
 - 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

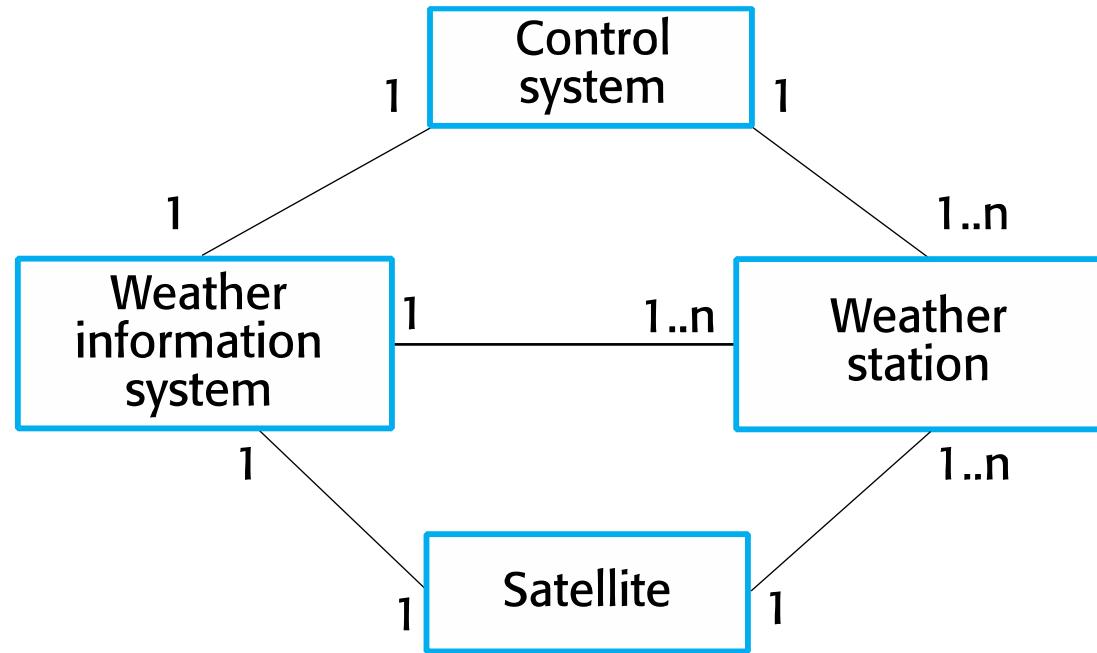


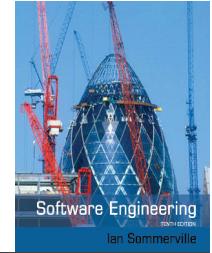
- ✧ A system context model is a structural model that demonstrates the other systems in the environment of the system being developed.
- ✧ An interaction model is a dynamic model that shows how the system interacts with its environment as it is used.

System context for the weather station

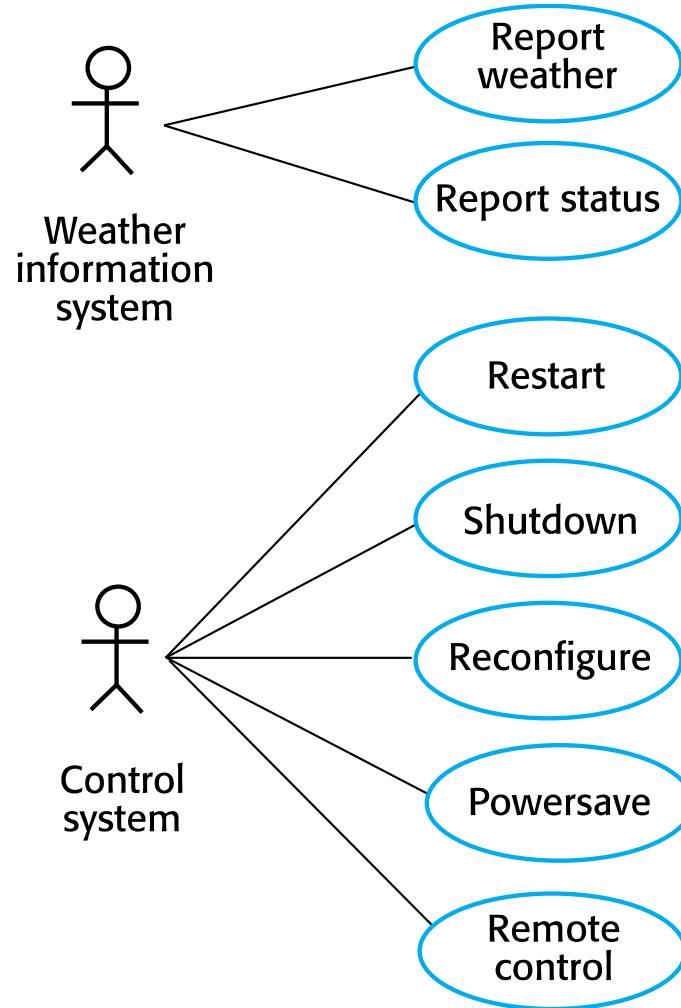


Software Engineering
Ian Sommerville





Weather station use cases

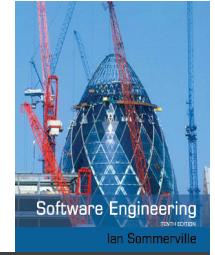




Use case description—Report weather

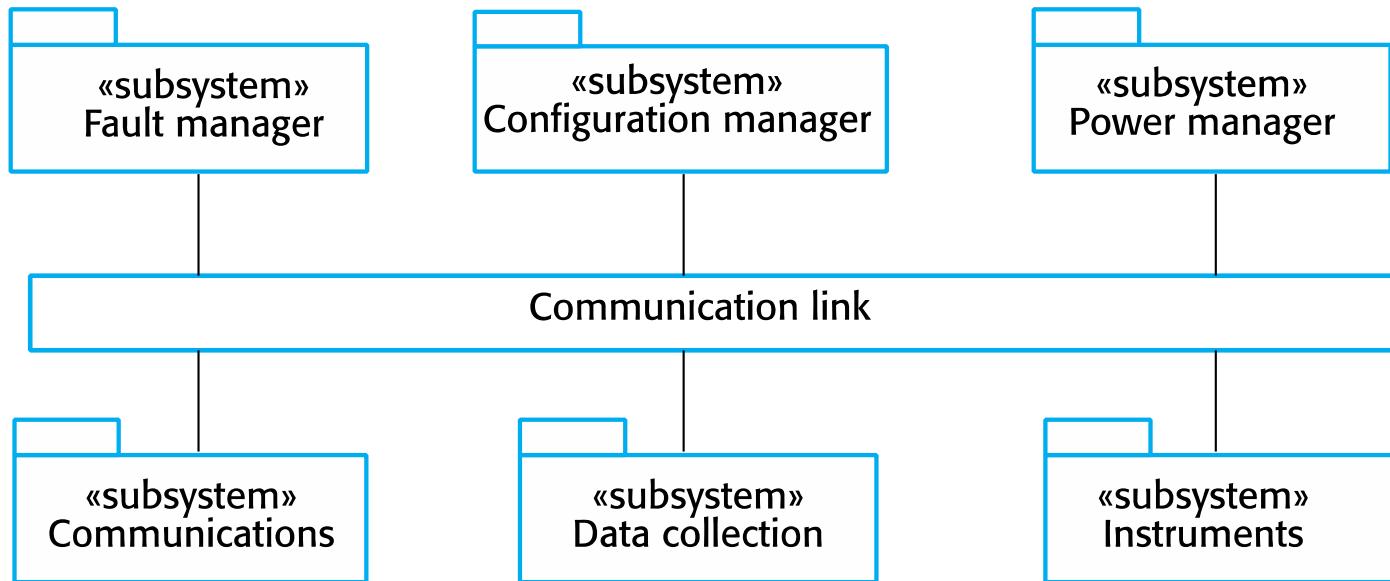
System	Weather station
Use case	Report weather
Actors	Weather information system, Weather station
Description	The weather station sends a summary of the weather data that has been collected from the instruments in the collection period to the weather information system. The data sent are the maximum, 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

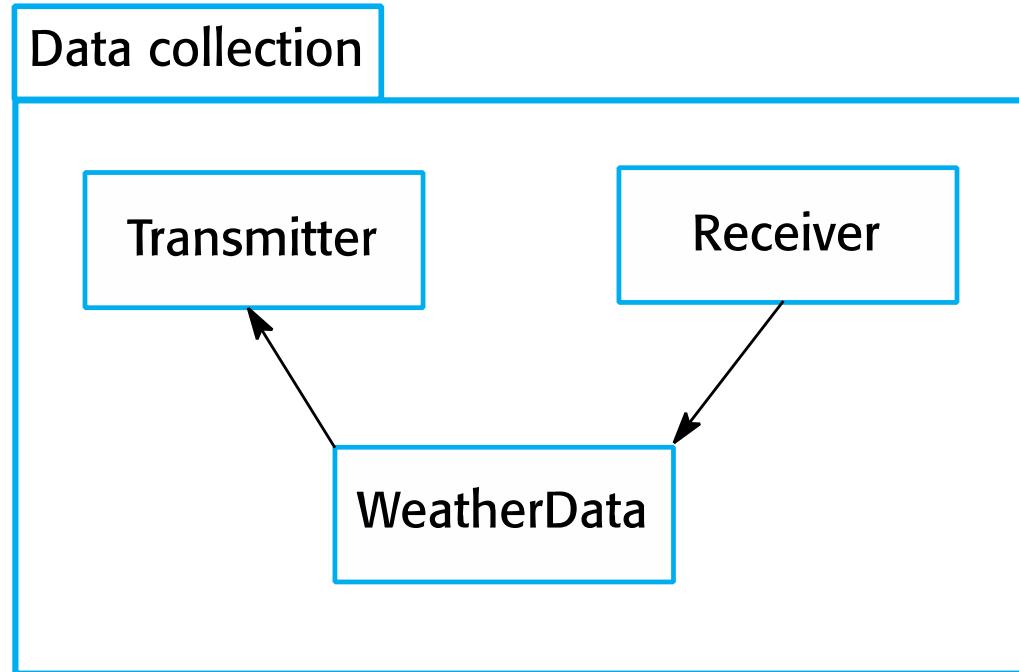
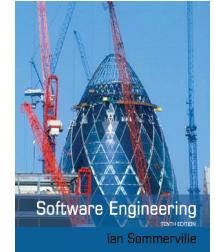


- ✧ Once interactions between the system and its environment have been understood, you use this information for designing the system architecture.
- ✧ You identify the major components that make up the system and their interactions, and then may organize the components using an architectural pattern 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

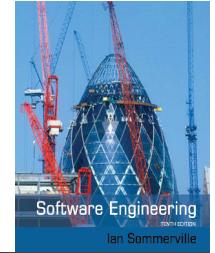


- ✧ Identifying object classes is often a difficult part of object oriented design.
- ✧ There is no 'magic formula' for object identification. It relies on the skill, experience and domain knowledge of system designers.
- ✧ Object identification is an iterative process. You are unlikely to get it right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



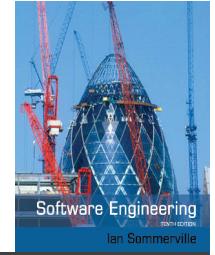
- ✧ Design models show the objects and object classes and relationships between these entities.
- ✧ There are two kinds of design model:
 - Structural models describe the static structure of the system in terms of object classes and relationships.
 - Dynamic models describe the dynamic interactions between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

Examples of design models



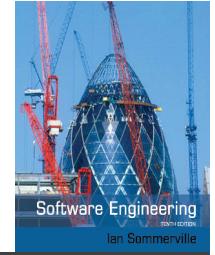
- ✧ Subsystem models that show logical groupings of objects into coherent subsystems.
- ✧ Sequence models that show the sequence of object interactions.
- ✧ State machine models that show how individual objects change their state in response to events.
- ✧ Other models include use-case models, aggregation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

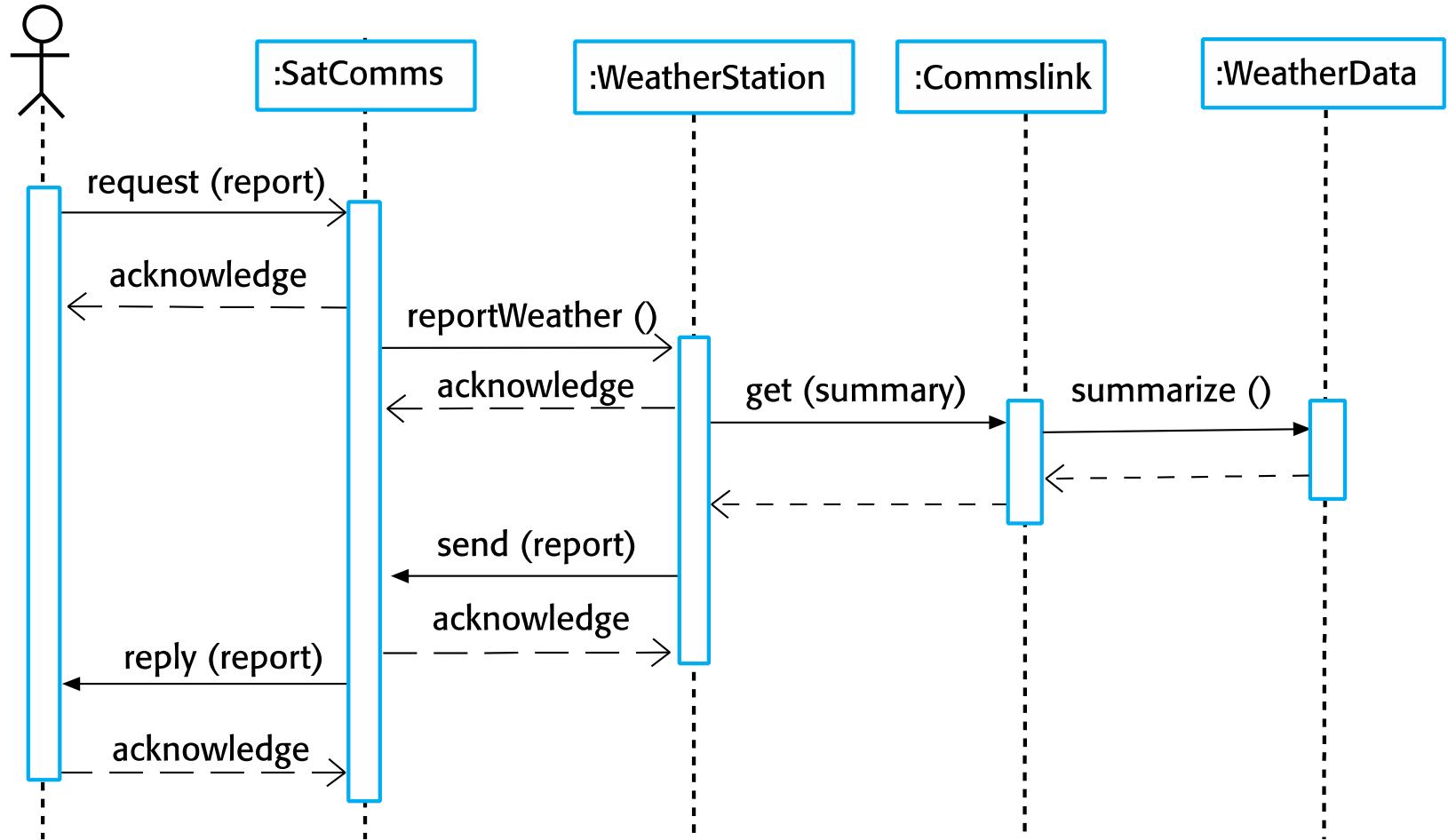


Software Engineering

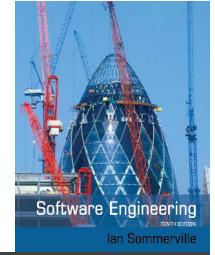
Ian Sommerville

Sequence diagram describing data collection

information system

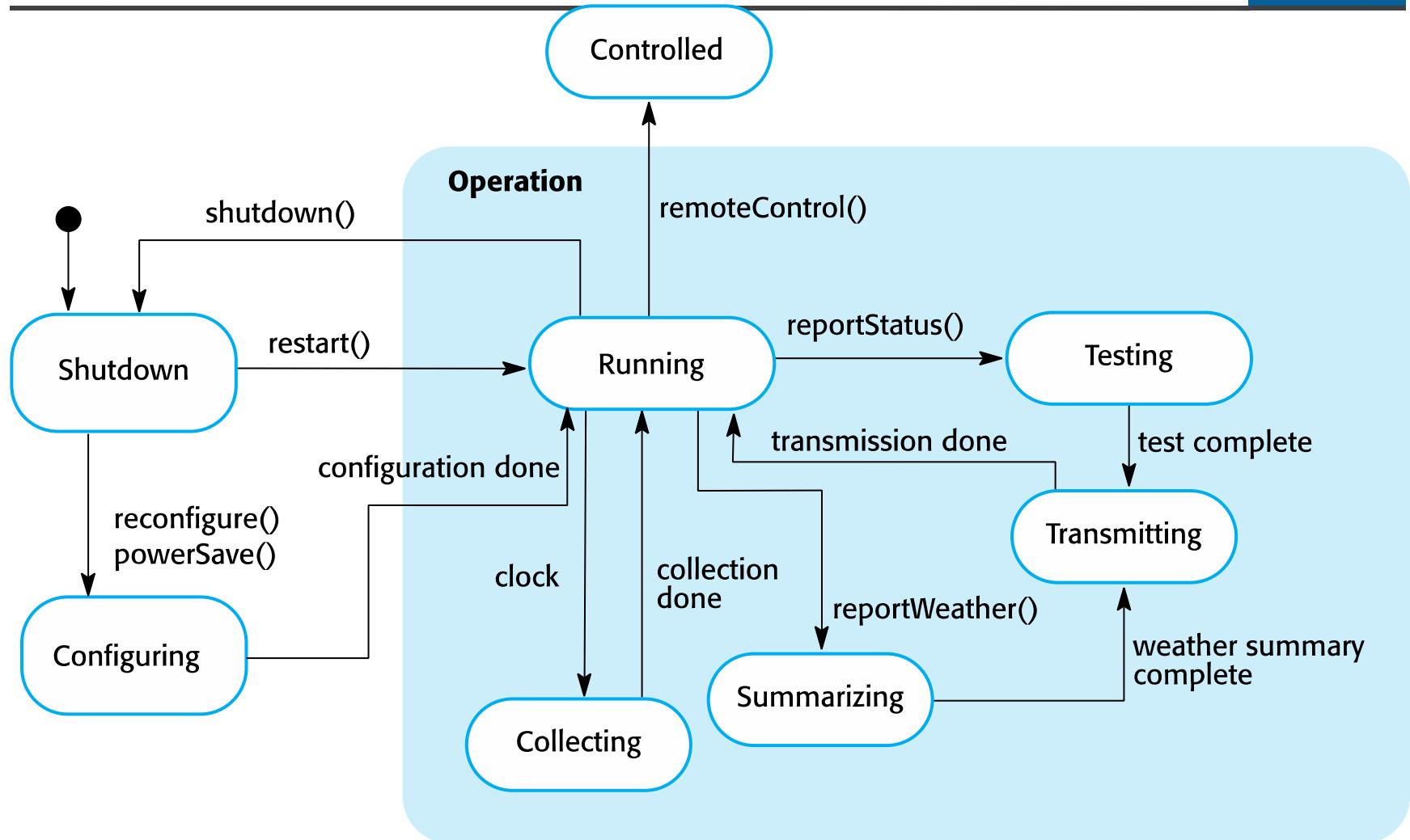
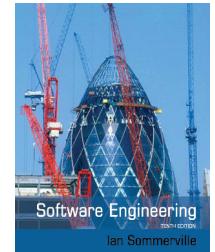


State diagrams



- ✧ 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



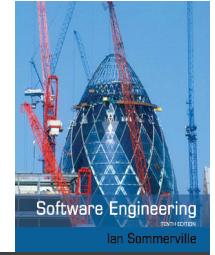
- ✧ A design pattern is a way of reusing abstract knowledge about a problem and its solution.
- ✧ A pattern is a description of the problem and the essence of its solution.
- ✧ It should be sufficiently abstract to be reused in different settings.
- ✧ Pattern descriptions usually make use of object-oriented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



Software Engineering

Ian Sommerville

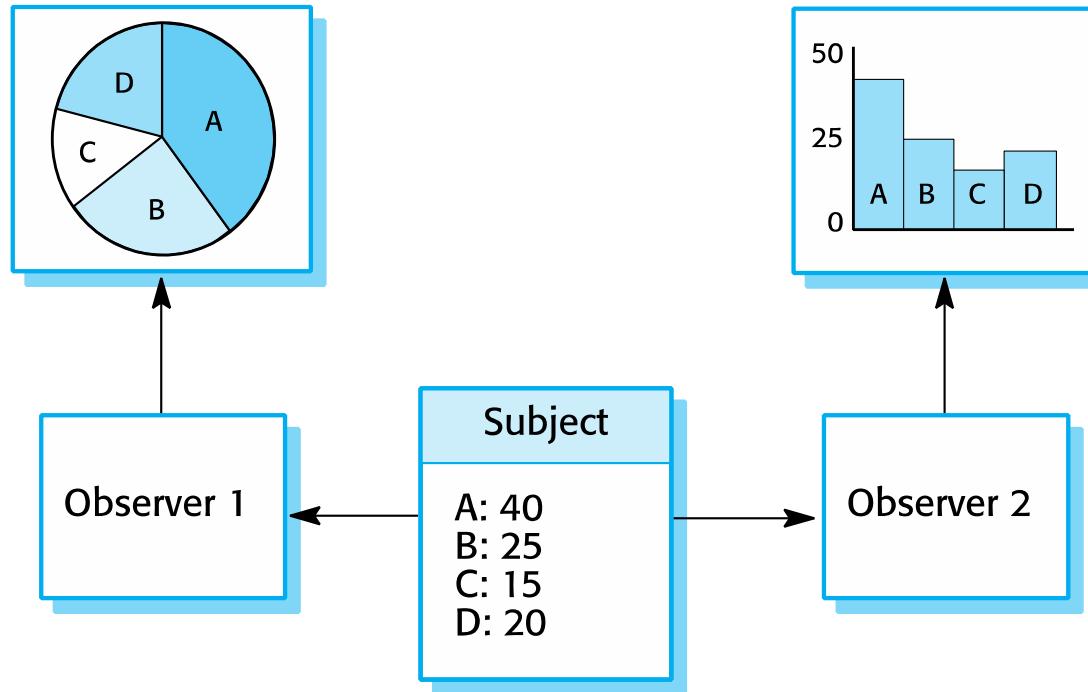
Pattern name	Observer
Description	<p>Separates the display of the state of an object from the object itself and allows alternative displays to be provided. When the object state changes, all displays are automatically notified and updated to reflect the change.</p>
Problem description	<p>In many situations, 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)

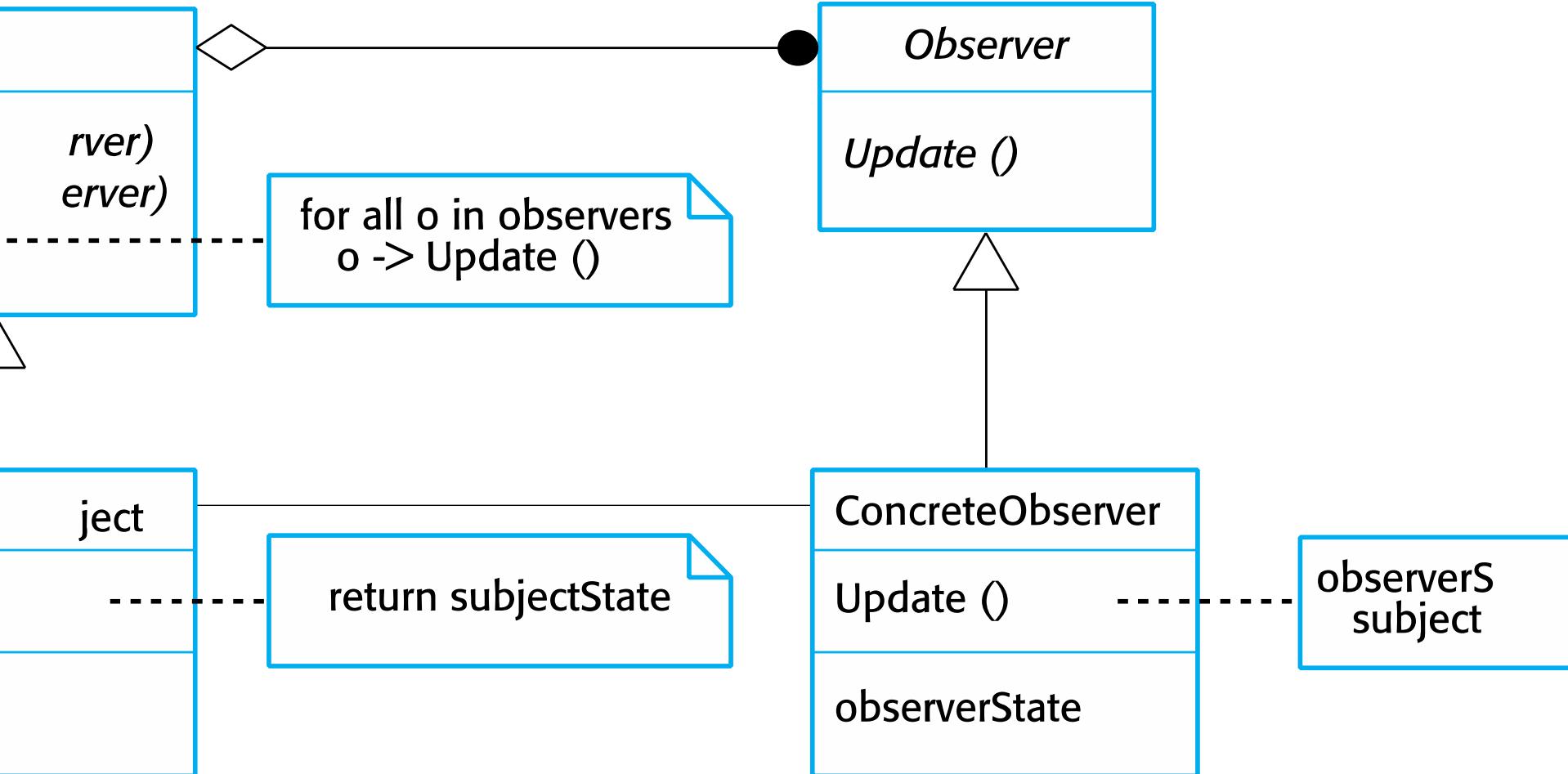
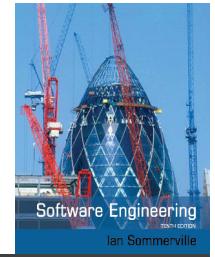


Pattern name	Observer
Solution description	<p>This involves two abstract objects, Subject and Observer, and two concrete objects, ConcreteSubject and ConcreteObject, which inherit the attributes of the related abstract objects. The abstract objects include general operations that are applicable in all 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



Software Engineering
Ian Sommerville

- ✧ 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



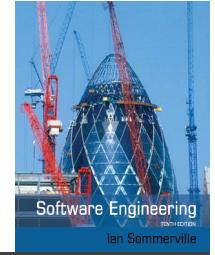
- ✧ Focus here is not on programming, although this is obviously important, but on other implementation issues that are often not covered in programming texts:
 - **Reuse** Most modern software is constructed by reusing existing components or systems. When you are developing software, you should make 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.

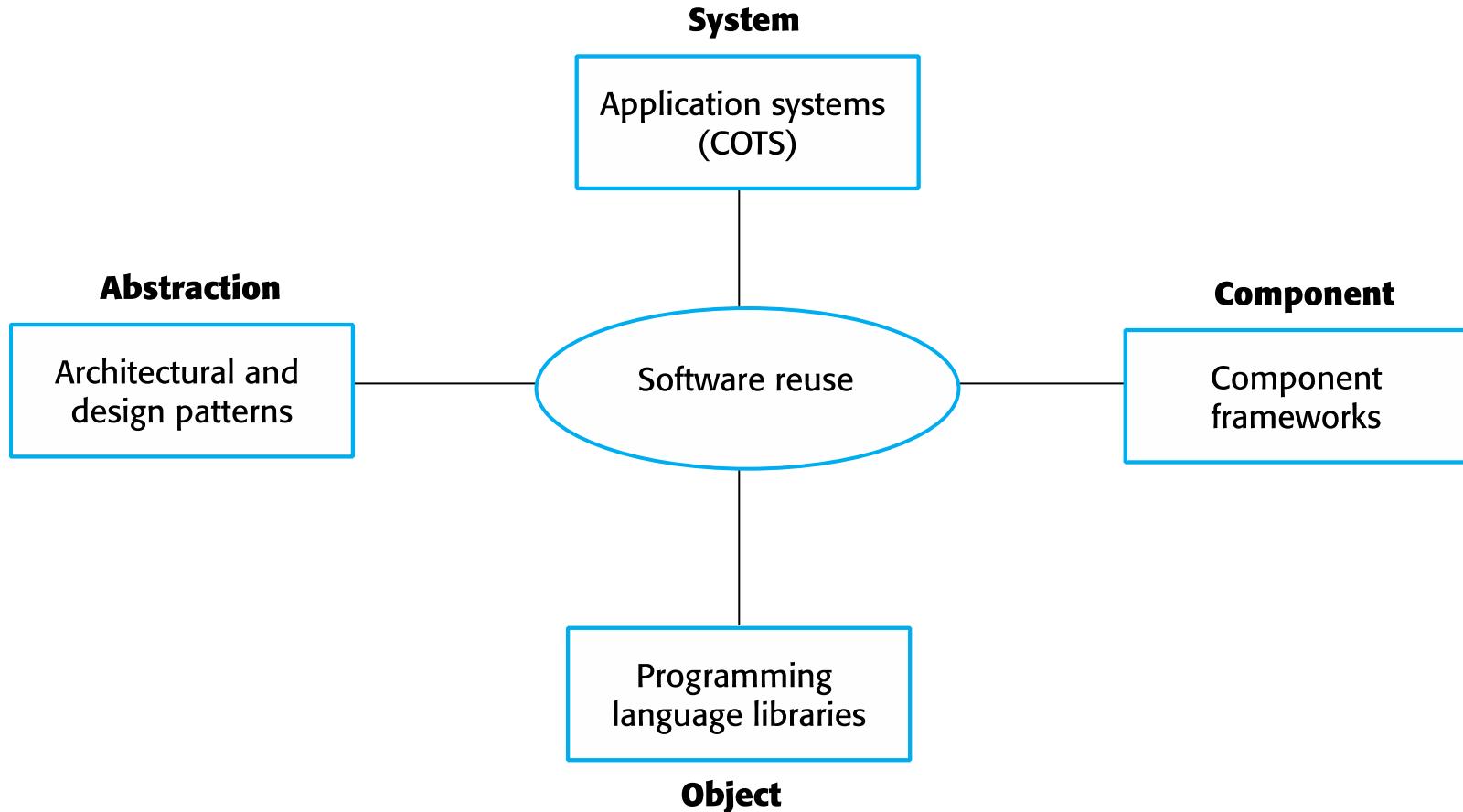
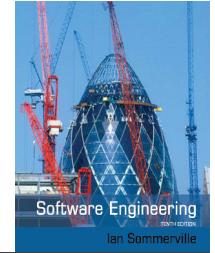
✧ The component level

- 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

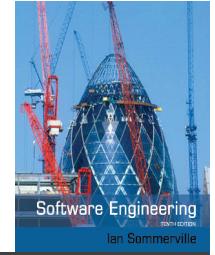


Reuse costs



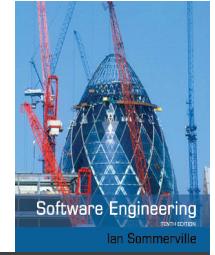
- ✧ The costs of the time spent in looking for software to reuse and assessing whether or not it meets your needs.
- ✧ Where applicable, the costs of buying the reusable software. For large off-the-shelf systems, these costs can be very high.
- ✧ The costs of adapting and configuring the reusable software 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



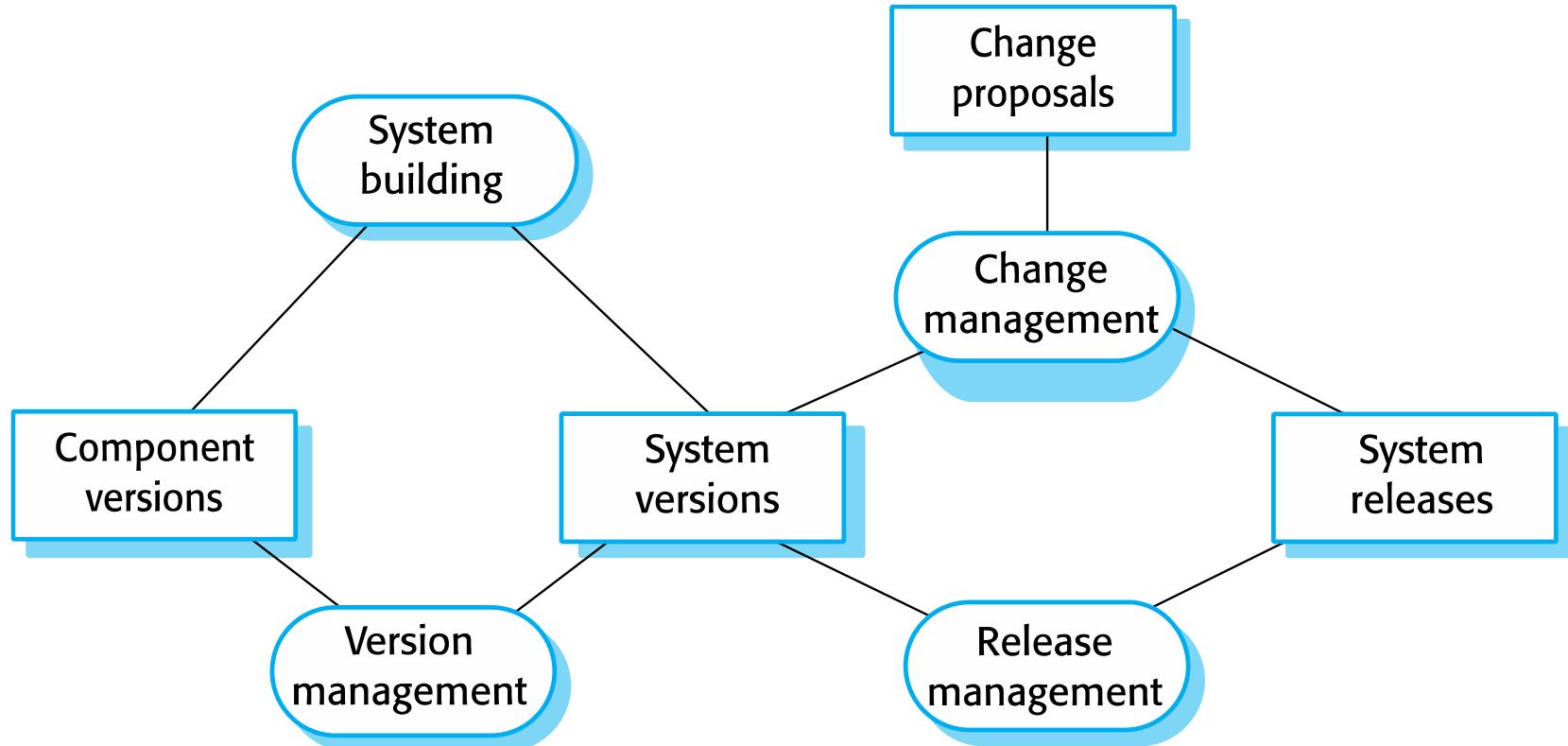
- ✧ Configuration management is the name given to the general process of managing a changing software system.
- ✧ The aim of configuration management is to support the system integration process so that all developers can access the project code and documents in a controlled way, find out what changes 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

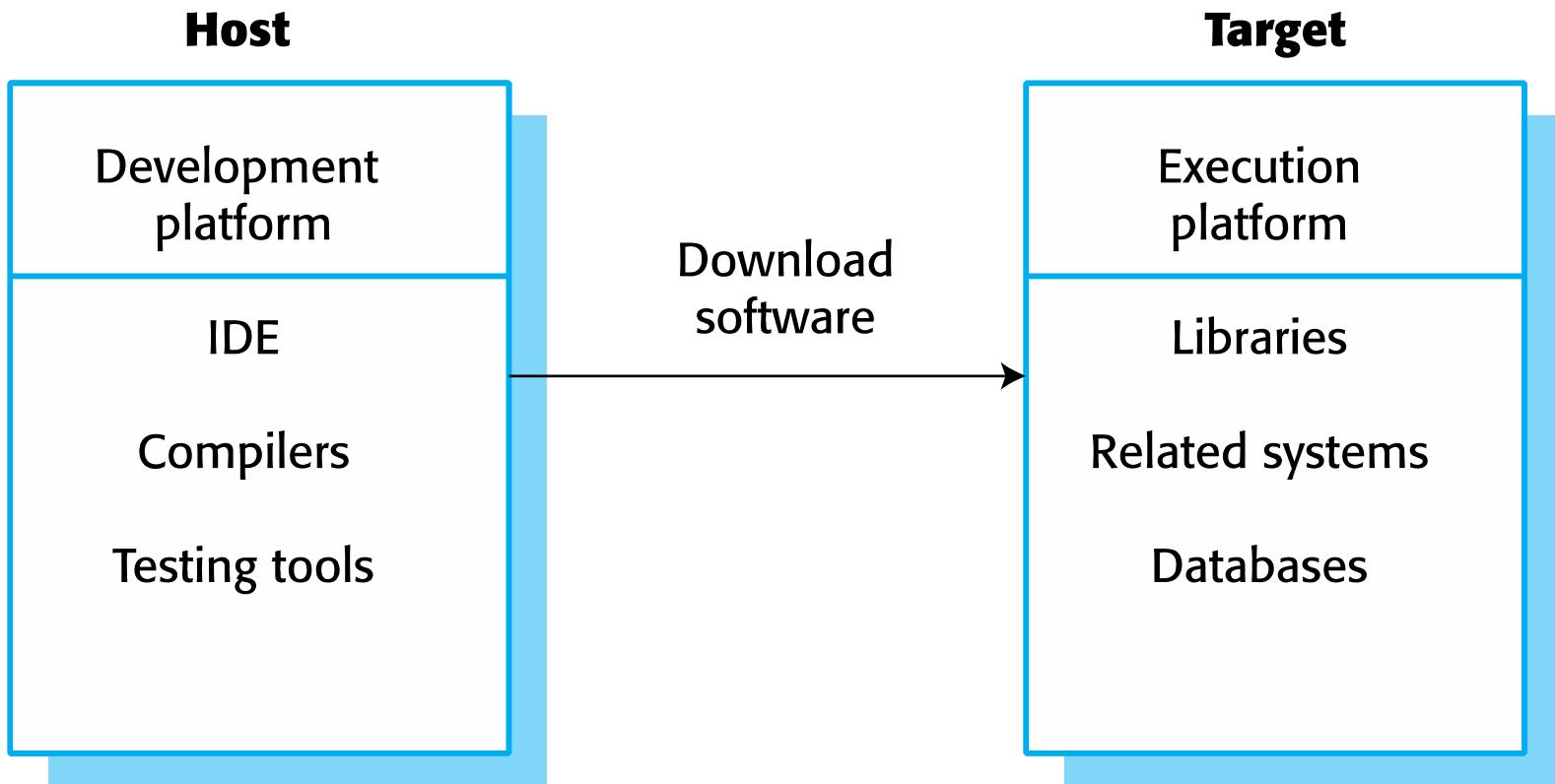
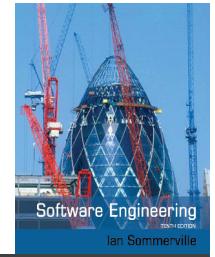


Host-target development



- ✧ 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.



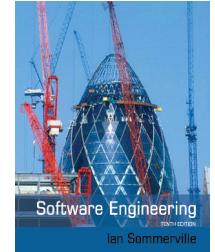
Open source development

Open source development



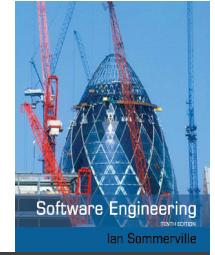
- ✧ Open source development is an approach to software development in which the source code of a software system is published and volunteers are invited to participate in the development process
- ✧ Its roots are in the Free Software Foundation (www.fsf.org), which advocates that source code should not 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



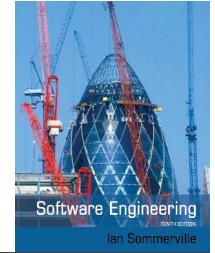
- ✧ The best-known open source product is, of course, the Linux operating system which is widely used as a server system and, increasingly, as a desktop environment.
- ✧ Other important open source products are Java, the Apache web server and the mySQL database management system.

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



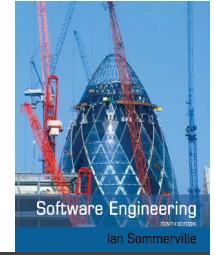
- ✧ More and more product companies are using an open source approach to development.
- ✧ Their business model is not reliant on selling a software product but on selling support for that product.
- ✧ They believe that involving the open source community will allow software to be developed more cheaply,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



Open source licensing

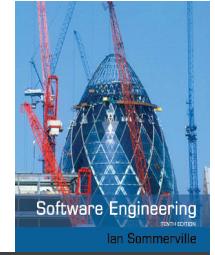
- ✧ A fundamental principle of open-source development is that source code should be freely available, this does not mean that anyone can do as they wish with that code.
 - Legally, the developer of the code (either a company or an individual) still owns the code. They can place restrictions on how it 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

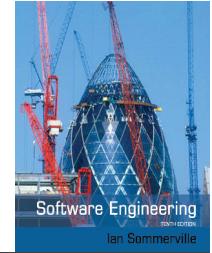


- ✧ The GNU General Public License (GPL). This is a so-called ‘reciprocal’ license that means that if you use open source software that is licensed under the GPL license, then you must make that software open source.
- ✧ The GNU Lesser General Public License (LGPL) is a variant of the GPL license where 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



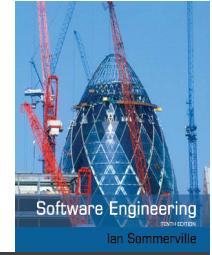
- ✧ Establish a system for maintaining information about open-source components that are downloaded and used.
- ✧ Be aware of the different types of licenses and understand how a component is licensed before it is used.
- ✧ Be aware of evolution pathways for components.
- ✧ Educate people about open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

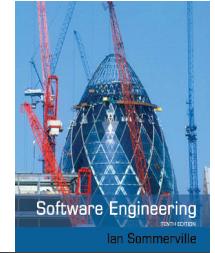


- ✧ When developing software, you should always consider the possibility of reusing existing software, either as components, services or complete systems.
- ✧ Configuration management is the process of managing changes to an evolving software system. It is essential when a team of people are cooperating 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.



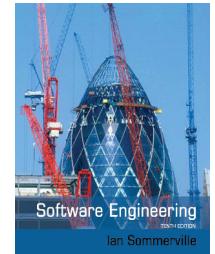
Chapter 8 – Software Testing

Topics covered

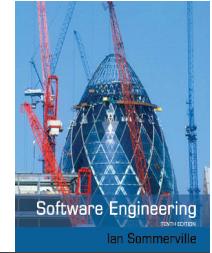


- ✧ Development testing
- ✧ Test-driven development
- ✧ Release testing
- ✧ User testing

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

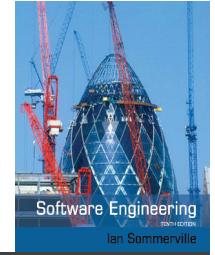
- ✧ To demonstrate to the developer and the customer that the software meets its requirements.
 - For custom software, this means that there should be at least one test for every requirement in the requirements document. For generic software products, it means that there should be tests for all of the 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



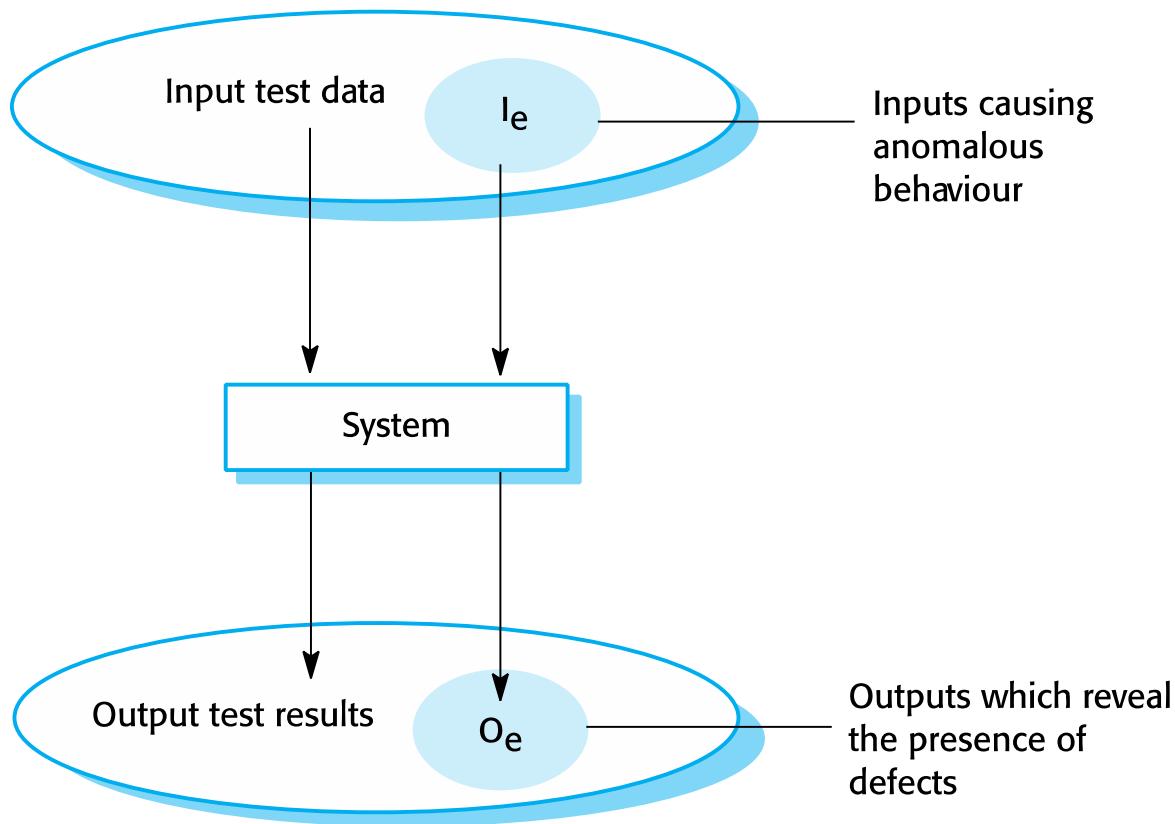
✧ Validation testing

- 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.

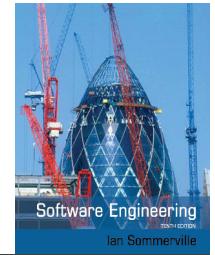
✧ Defect testing

- 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

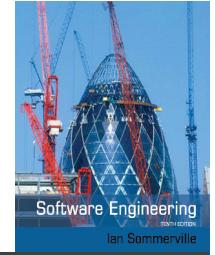


Verification vs validation



- ✧ 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



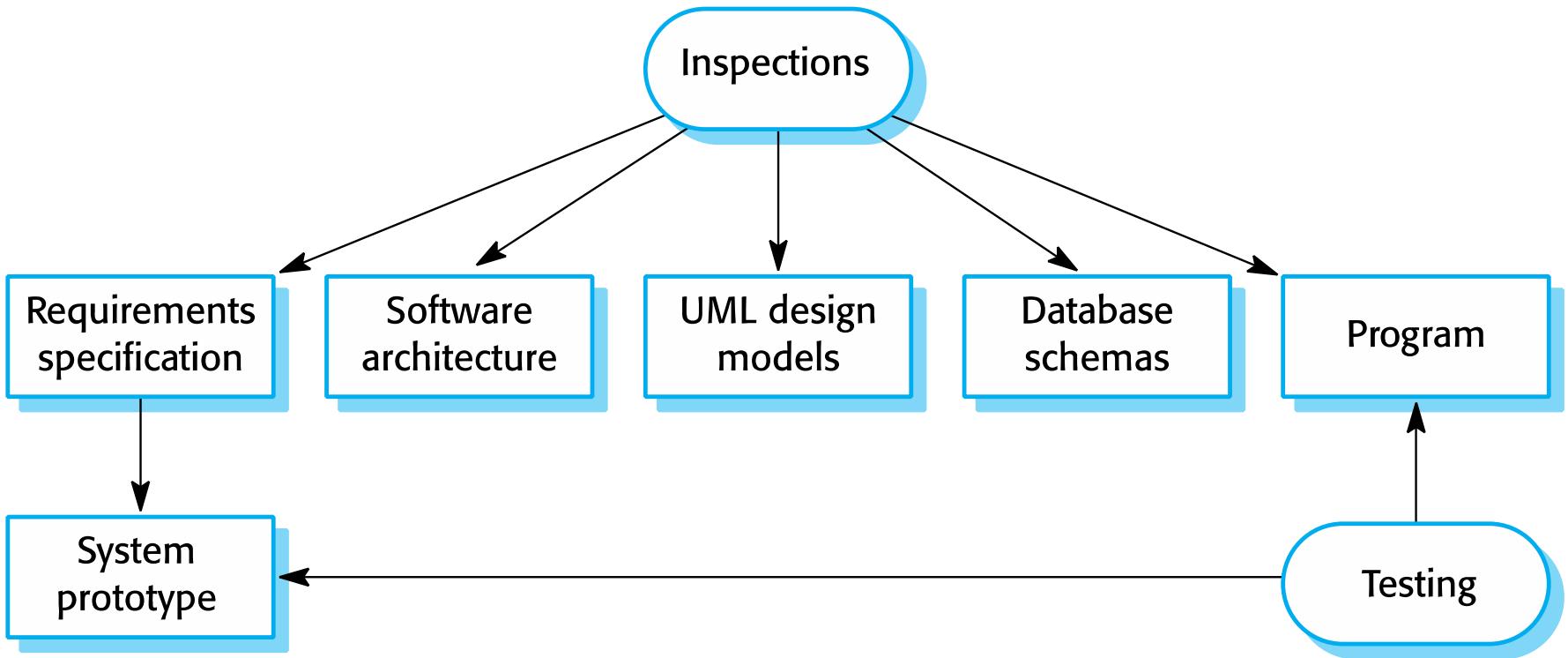
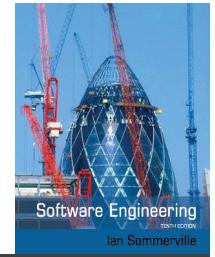
- ✧ Aim of V & V is to establish confidence that the system is ‘fit for purpose’.
- ✧ Depends on system’s purpose, user expectations and marketing environment
 - Software purpose
 - The level of confidence depends on how critical the software is to an organisation.
 - User expectations
 - Users may have 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

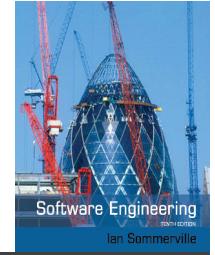


- ✧ **Software inspections** Concerned with analysis of the static system representation to discover problems (static verification)
 - May be supplement by tool-based document and code analysis.
 - Discussed in Chapter 15.
- ✧ **Software testing** Concerned with exercising and observing product behaviour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

Inspections and testing

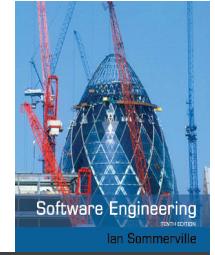


Software inspections



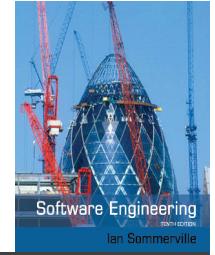
- ✧ These involve people examining the source representation with the aim of discovering anomalies and defects.
- ✧ Inspections not require execution of a system so may be used before implementation.
- ✧ They may be applied to any representation of the system (requirements, design, configuration data,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

Advantages of inspections



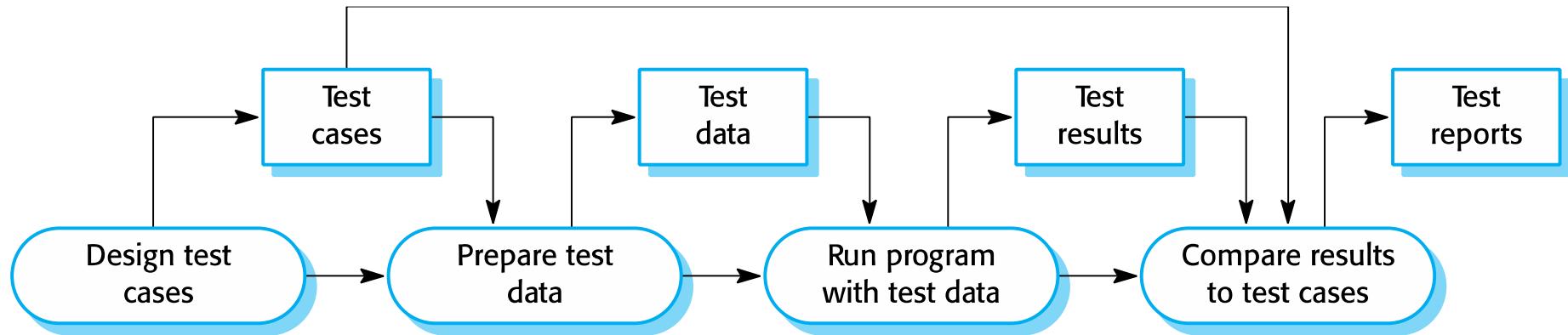
- ✧ During testing, errors can mask (hide) other errors.
Because inspection is a static process, you don't have to be concerned with interactions between errors.
- ✧ Incomplete versions of a system can be inspected without additional costs. If a program is incomplete, then you need to develop specialized 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

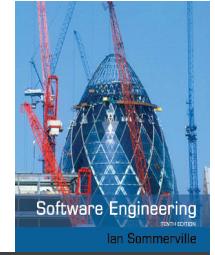


- ✧ Inspections and testing are complementary and not opposing verification techniques.
- ✧ Both should be used during the V & V process.
- ✧ Inspections can check conformance with a specification but not conformance with the customer's real requirements.
- ✧ Inspections cannot check non-functional characteristics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

A model of the software testing process



Stages of testing

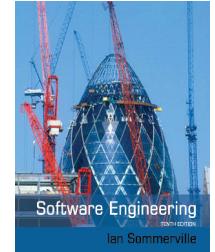


- ✧ Development testing, where the system is tested during development to discover bugs and defects.
- ✧ Release testing, where a separate testing team test a complete version of the system before it is released to users.
- ✧ User testing, where users or potential users of a system test the system in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



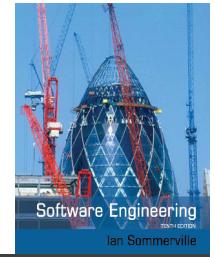
Development testing

Development testing



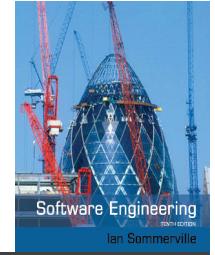
- ✧ Development testing includes all testing activities that are carried out by the team developing the system.
 - Unit testing, where individual program units or object classes are tested. Unit testing should focus on testing the functionality of objects or methods.
 - Component testing, where several 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



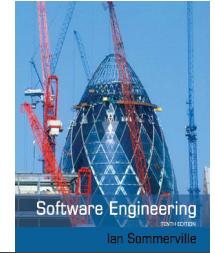
- ✧ Need to define test cases for reportWeather, calibrate, test, startup and shutdown.
- ✧ Using a state model, identify sequences of state transitions to be tested and the event sequences to cause these transitions
- ✧ For example:
 - Shutdown -> Running-> Shutdown
 - Configuring-> Running-> Testing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

Choosing unit test cases



- ✧ 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

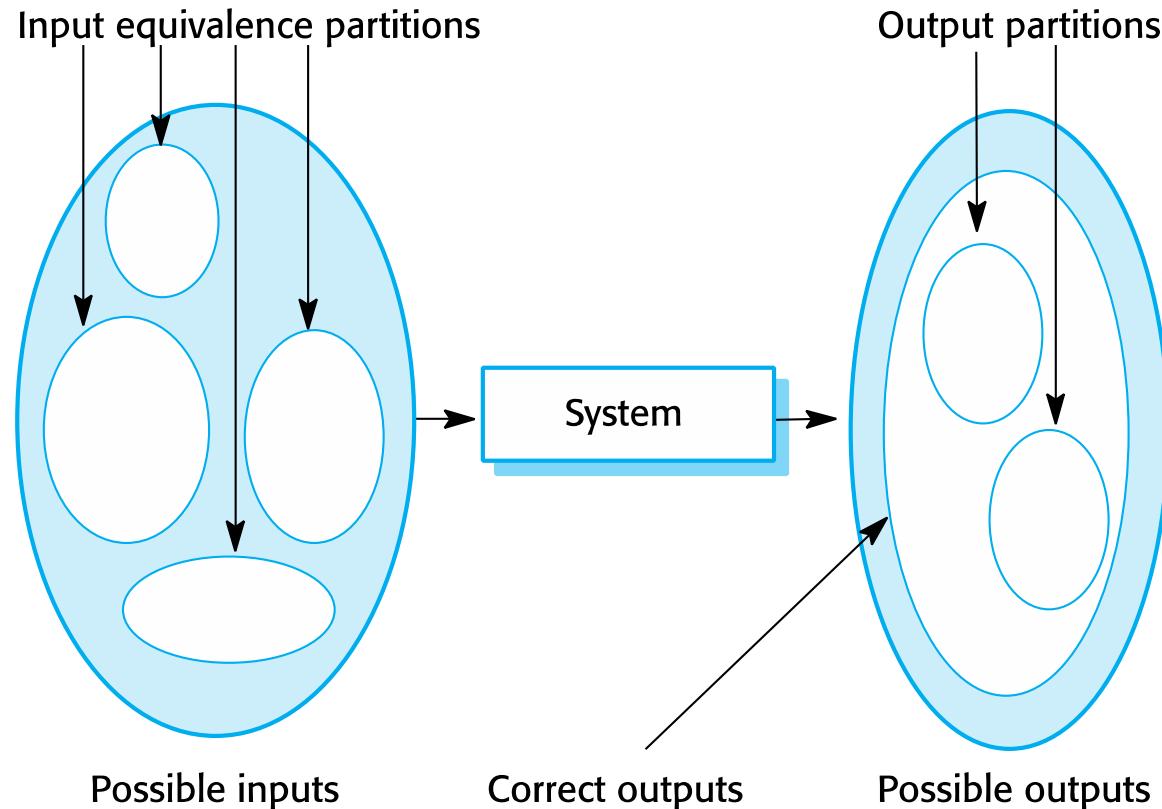


- ✧ Input data and output results often fall into different classes where all members of a class are related.
- ✧ Each of these classes is an equivalence partition or domain where the program behaves in an equivalent way for each class member.
- ✧ Test cases should be chosen from each partition.

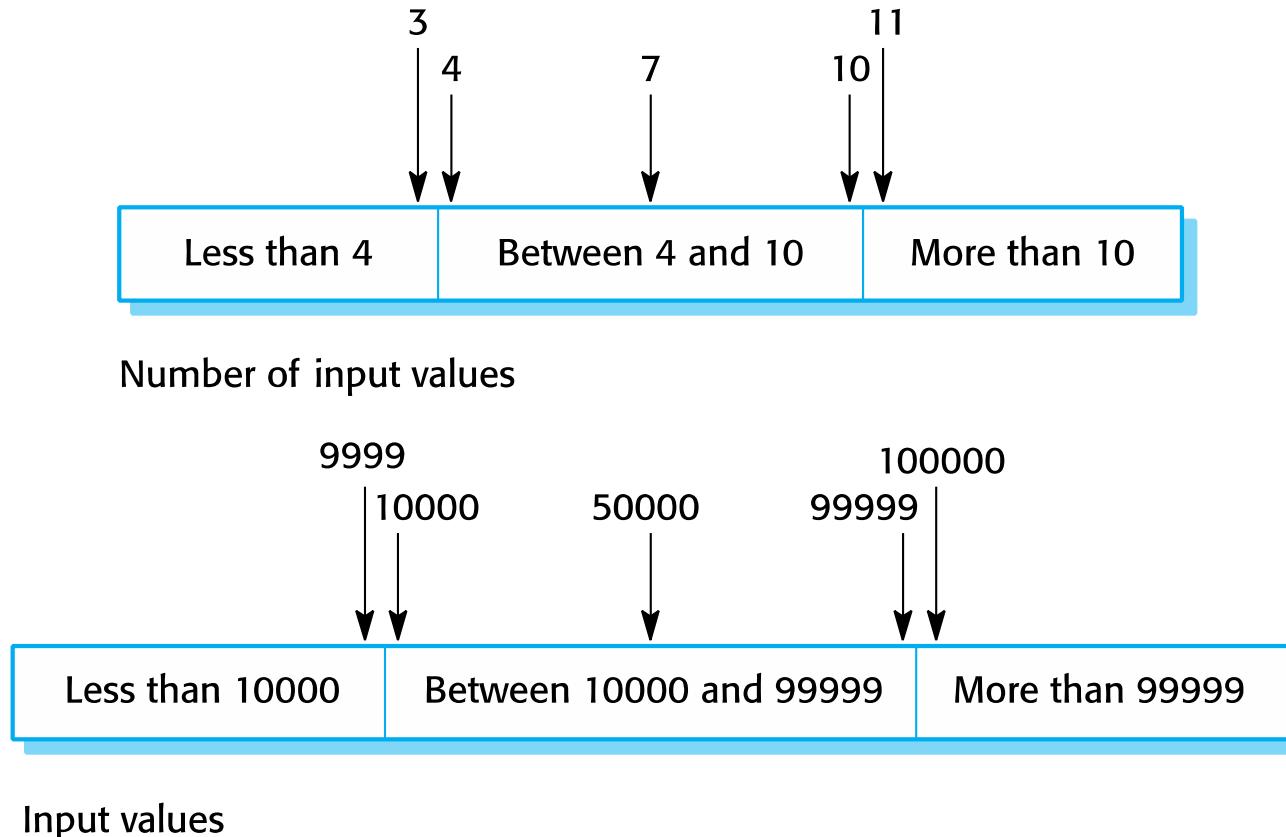
Equivalence partitioning



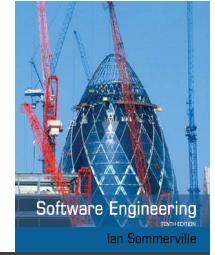
Software Engineering
Ian Sommerville



Equivalence partitions

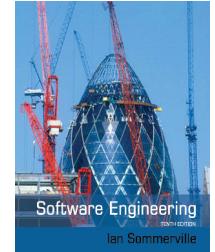


Testing guidelines (sequences)



- ✧ 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

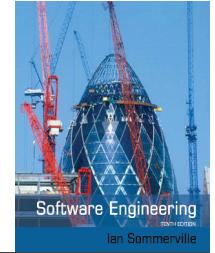


- ✧ Choose inputs that force the system to generate all error messages
- ✧ Design inputs that cause input buffers to overflow
- ✧ Repeat the same input or series of inputs numerous times
- ✧ Force invalid outputs to be generated
- ✧ Force computation results to be too large or too small.

Component testing



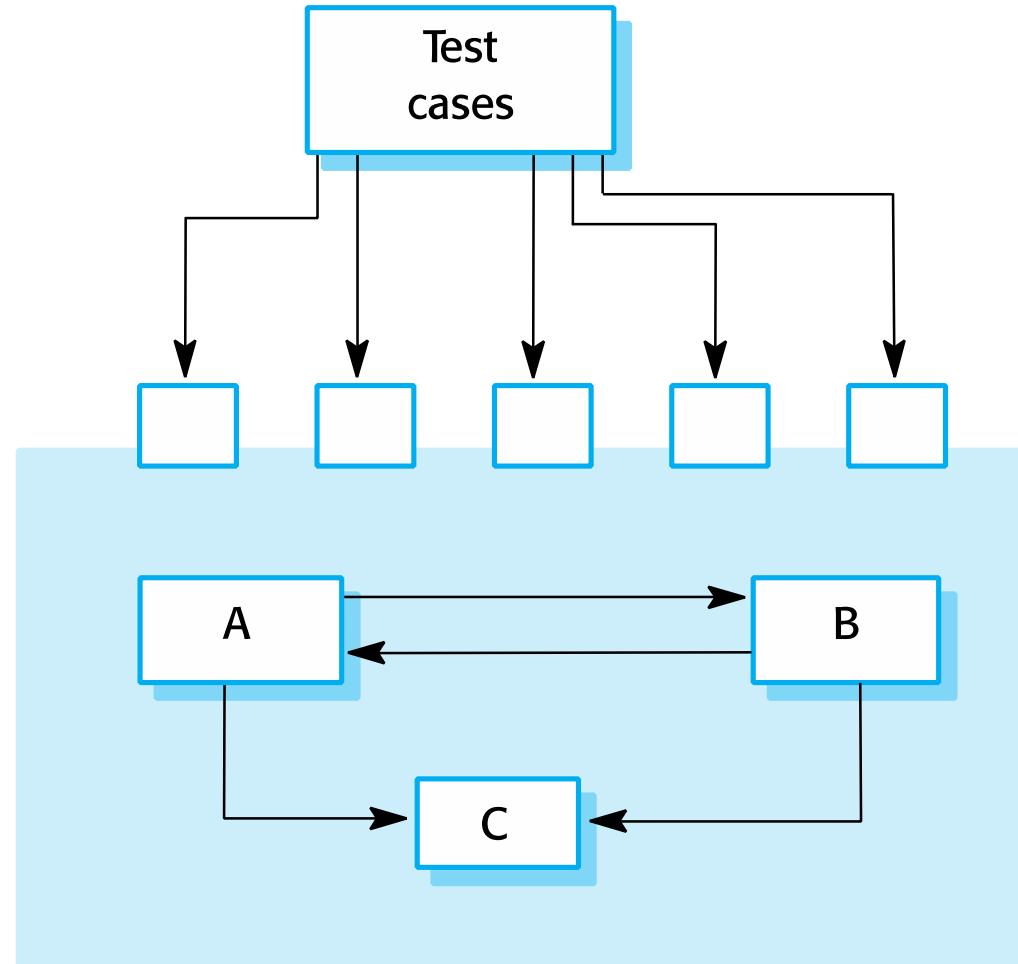
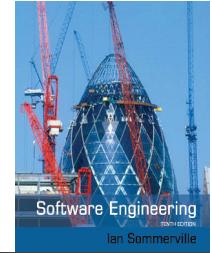
- ✧ Software components are often composite components that are made up of several interacting objects.
 - For example, in the weather station system, the reconfiguration component includes objects that deal with each aspect of the reconfiguration.
- ✧ You access the functionality of these objects through 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

- ✧ 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 testing





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.

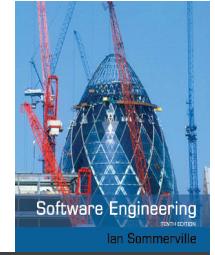
✧ Interface misunderstanding

- 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



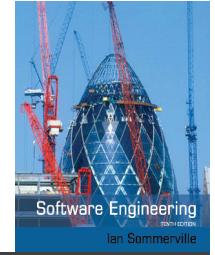
- ✧ System testing during development involves integrating components to create a version of the system and then testing the integrated system.
- ✧ The focus in system testing is testing the interactions between components.
- ✧ System testing checks that components are compatible, interact correctly 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

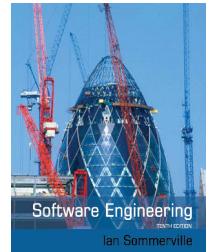


- ✧ During system testing, reusable components that have been separately developed and off-the-shelf systems may be integrated with newly developed components. The complete system is then tested.
- ✧ Components developed by different team members or sub-teams may be integrated at this stage. System testing 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

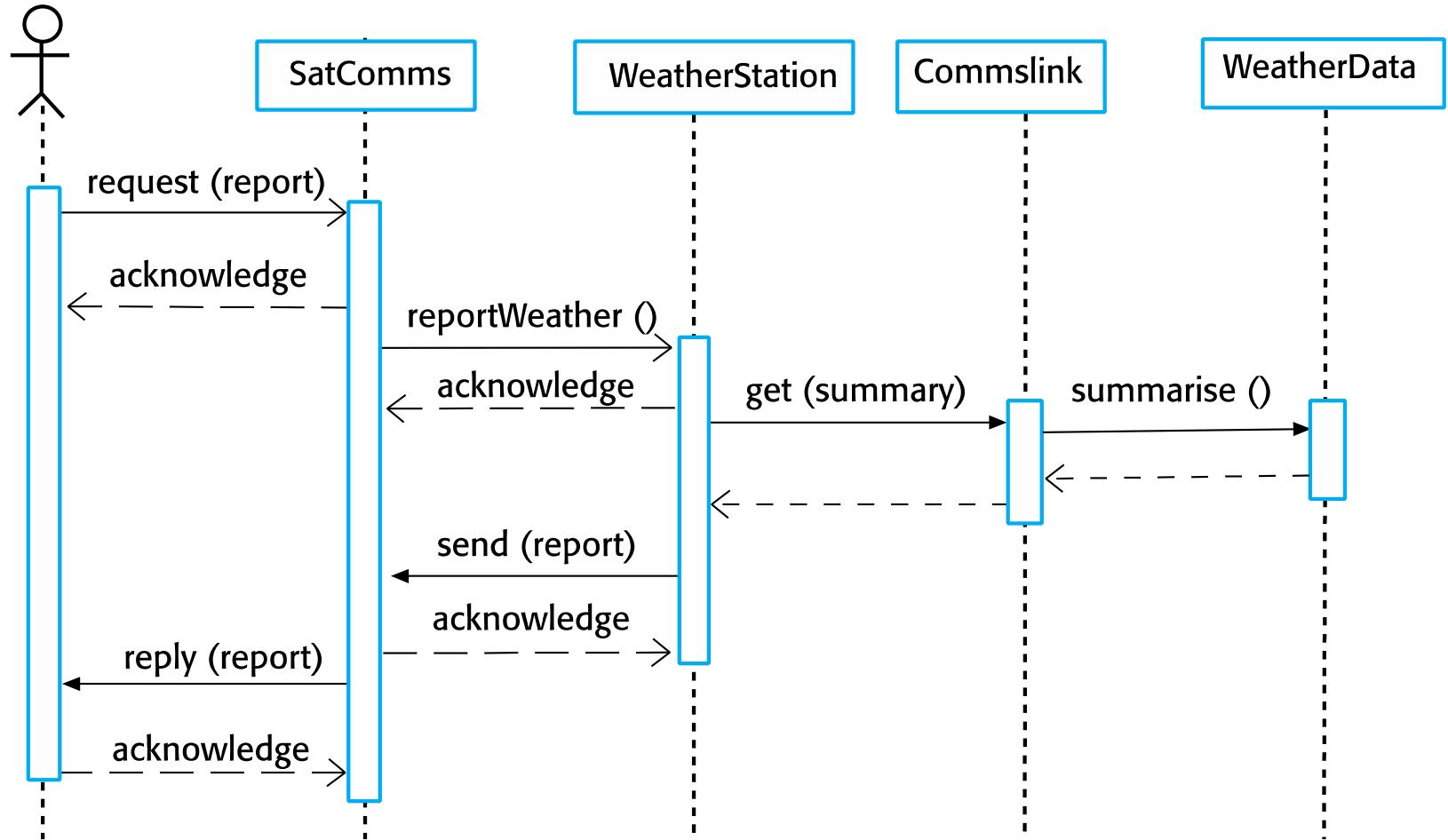


- ✧ The use-cases developed to identify system interactions can be used as a basis for system testing.
- ✧ Each use case usually involves several system components so testing the use case forces these interactions to occur.
- ✧ The sequence diagrams associated with the use case documents the components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



Collect weather data sequence chart

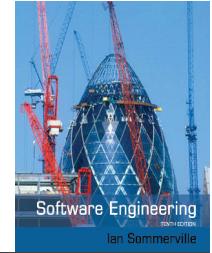
information system



Test cases derived from sequence diagram



- ✧ An input of a request for a report should have an associated acknowledgement. A report should ultimately be returned from the request.
 - You should create summarized data that can be used to check that the report is correctly organized.
- ✧ An input request for a report to WeatherStation results in a summarized report being generated.
 - Can be tested by creating raw data corresponding to the summary that you have prepared for the test of SatComms and checking that the WeatherStation object correctly produces this summary. This raw data is also used to test the WeatherData object.



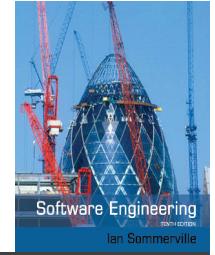
Testing policies

- ✧ 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.



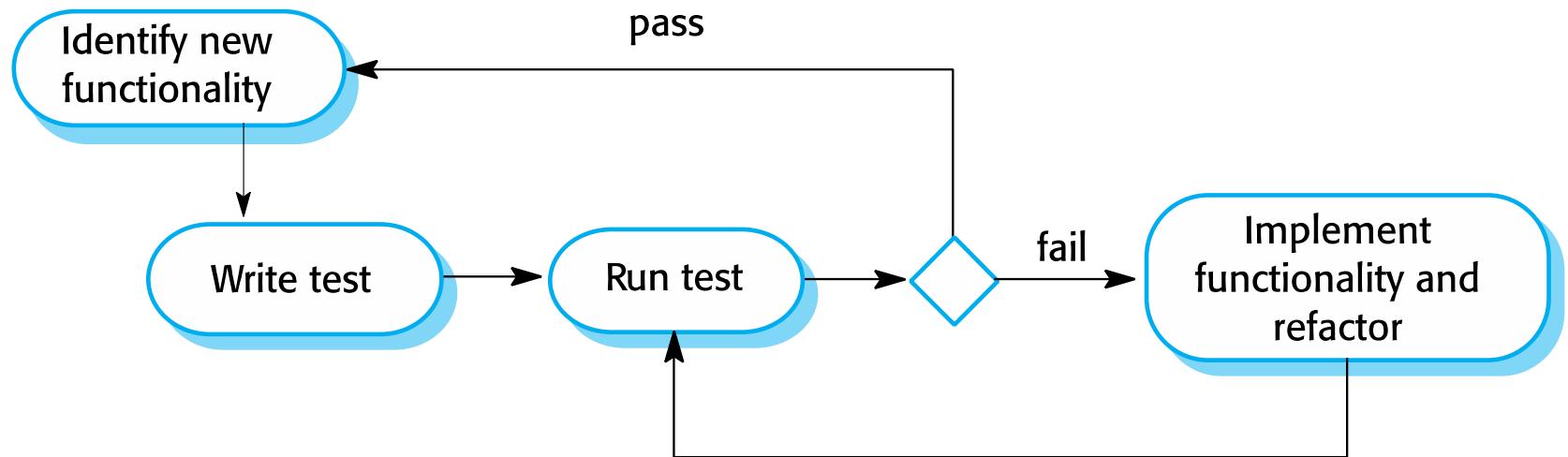
Test-driven development

Test-driven development

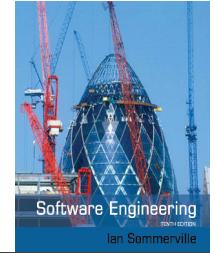


- ✧ Test-driven development (TDD) is an approach to program development in which you inter-leave testing and code development.
- ✧ Tests are written before code and ‘passing’ the tests is the critical driver of development.
- ✧ You develop code incrementally, along with a test for that increment. You 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.

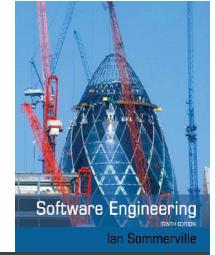
✧ Simplified debugging

- 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



- ✧ 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

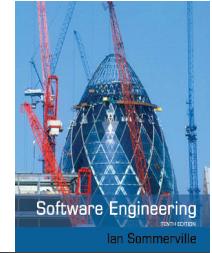


- ✧ Release testing is a form of system testing.
- ✧ Important differences:
 - A separate team that has not been involved in the system development, should be responsible for release testing.
 - System testing by the development team should focus on discovering bugs in the system (defect testing). The 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.
- ✧ Mencare system 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.

A usage scenario for the Mentcare system



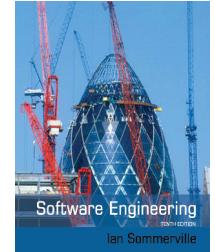
George is a nurse who specializes in mental healthcare. One of his responsibilities is to visit patients at home to check that their treatment is effective and that they are not suffering from medication side effects.

On a day for home visits, George logs into the Mentcare system and uses it to print his schedule of home visits for that day, along with summary information about the patients to be visited. He requests that the records for these patients be downloaded to his laptop. He is prompted for his key phrase to encrypt the records on the laptop.

One of the patients that he 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. George looks up Jim's record and is prompted for his key phrase to decrypt the record. He checks the drug prescribed and queries its side effects. Sleeplessness is a known side effect so he notes the problem in Jim's record and suggests that he visits the clinic to have his medication changed. Jim agrees so George enters a prompt to call him when he gets back to the clinic to make an appointment with a physician. George ends the consultation and the system re-encrypts Jim's record.

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

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.

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.

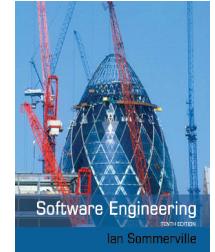


Software Engineering

Ian Sommerville

User testing

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

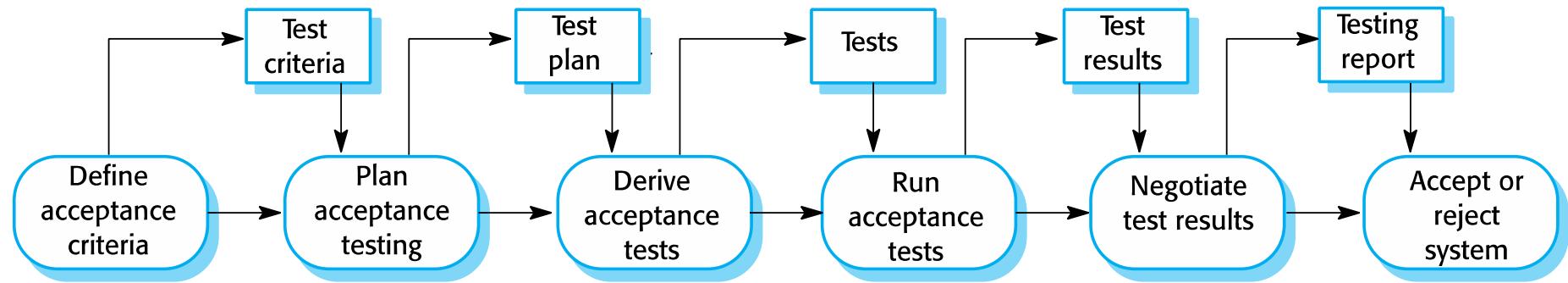
- Customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment. Primarily for custom systems.

The acceptance testing process

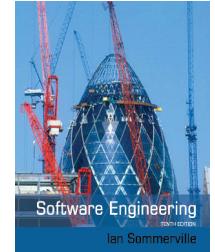


Software Engineering

Ian Sommerville

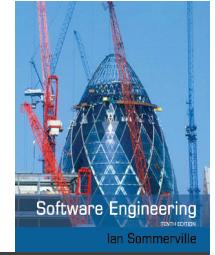


Stages in the acceptance testing process



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

Agile methods and acceptance testing



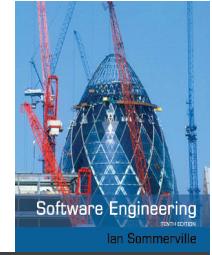
- ✧ In agile methods, the user/customer is part of the development team and is responsible for making decisions on the acceptability of the system.
- ✧ Tests are defined by the user/customer and are integrated with other tests in that they are run automatically when changes are made.
- ✧ There is no 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



- ✧ 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.

Key points

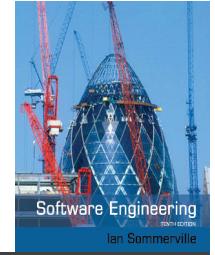


- ✧ When testing software, you should try to ‘break’ the software by using experience and guidelines to choose types of test case that have been effective in discovering defects in other systems.
- ✧ Wherever possible, you should write automated tests. The tests are embedded in a program that can be 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.



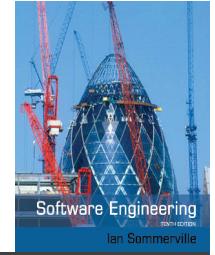
Chapter 9 – Software Evolution

Topics covered



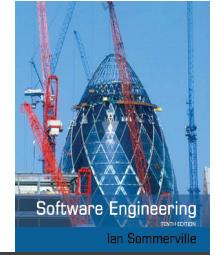
- ✧ Evolution processes
- ✧ Legacy systems
- ✧ Software maintenance

Software change



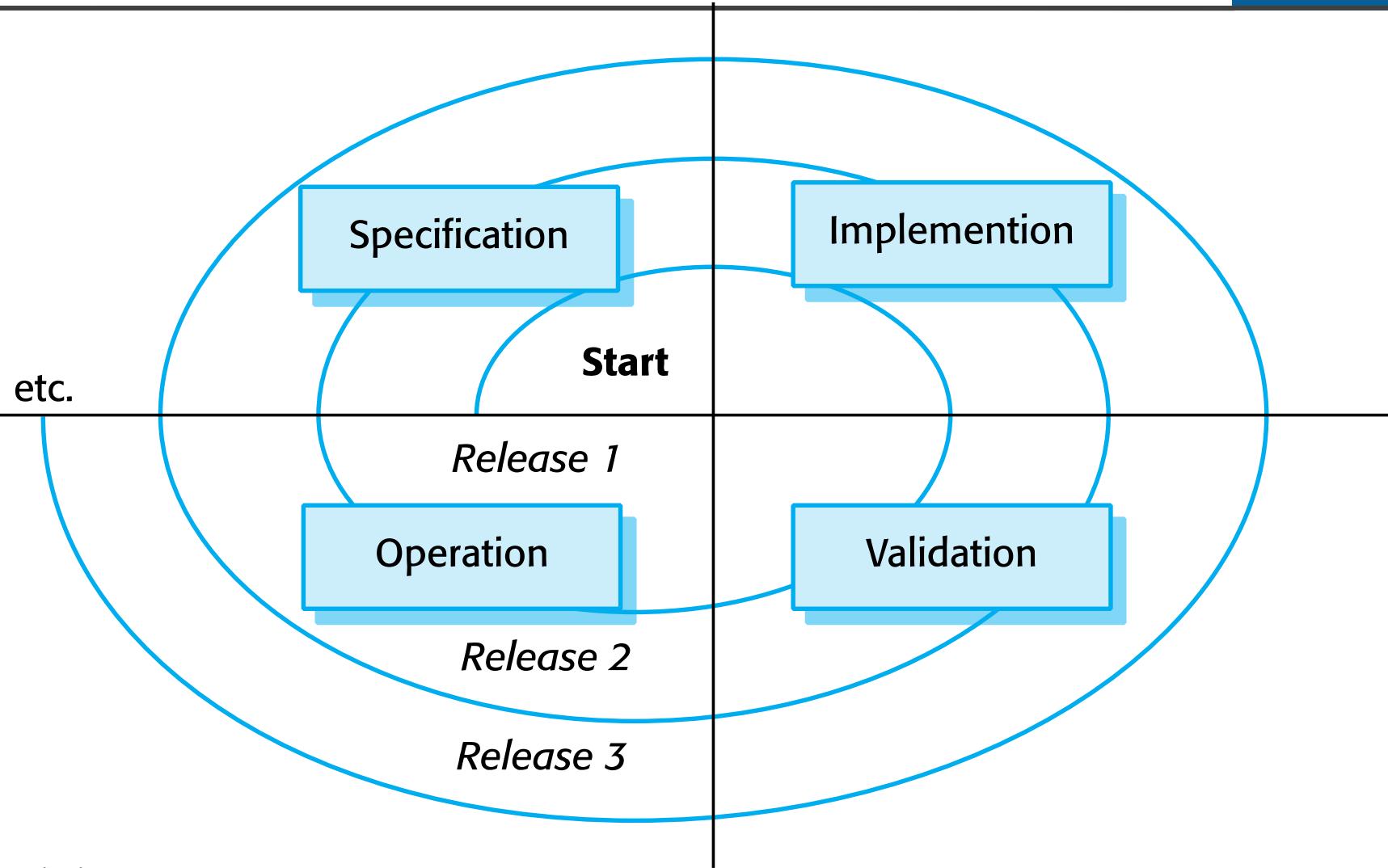
- ✧ Software change is inevitable
 - New requirements emerge when the software is used;
 - The business environment changes;
 - Errors must be repaired;
 - New computers and equipment is added to the system;
 - The performance or reliability of the system may have to be improved.
- ✧ A key problem for all organizations is implementing and managing change to their existing software systems.

Importance of evolution

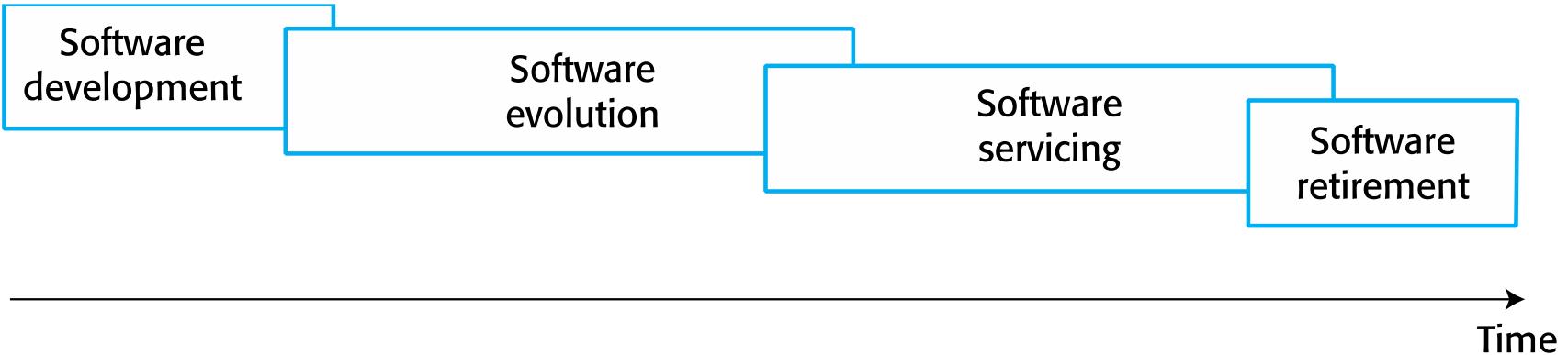


- ✧ Organisations have huge investments in their software systems - they are critical business assets.
- ✧ To maintain the value of these assets to the business, they must be changed and updated.
- ✧ The majority of the software budget in large companies is devoted to changing and evolving existing software rather than developing new software.

A spiral model of development and evolution



Evolution and servicing



Evolution and servicing



✧ Evolution

- The stage in a software system's life cycle where it is in operational use and is evolving as new requirements are proposed and implemented in the system.

✧ Servicing

- At this stage, the software remains useful but the only changes made are those required to keep it operational i.e. bug fixes and changes to reflect changes in the software's environment. No new functionality is added.

✧ Phase-out

- The software may still be used but no further changes are made to it.

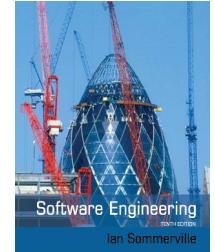


Software Engineering

Ian Sommerville

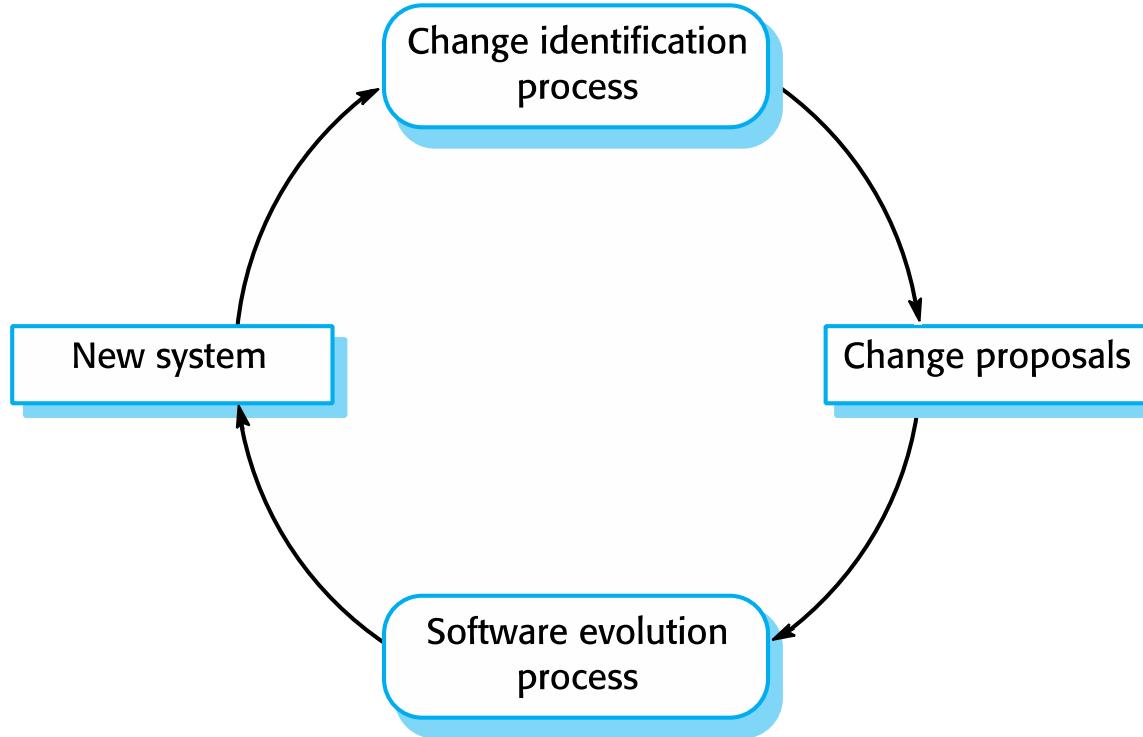
Evolution processes

Evolution processes

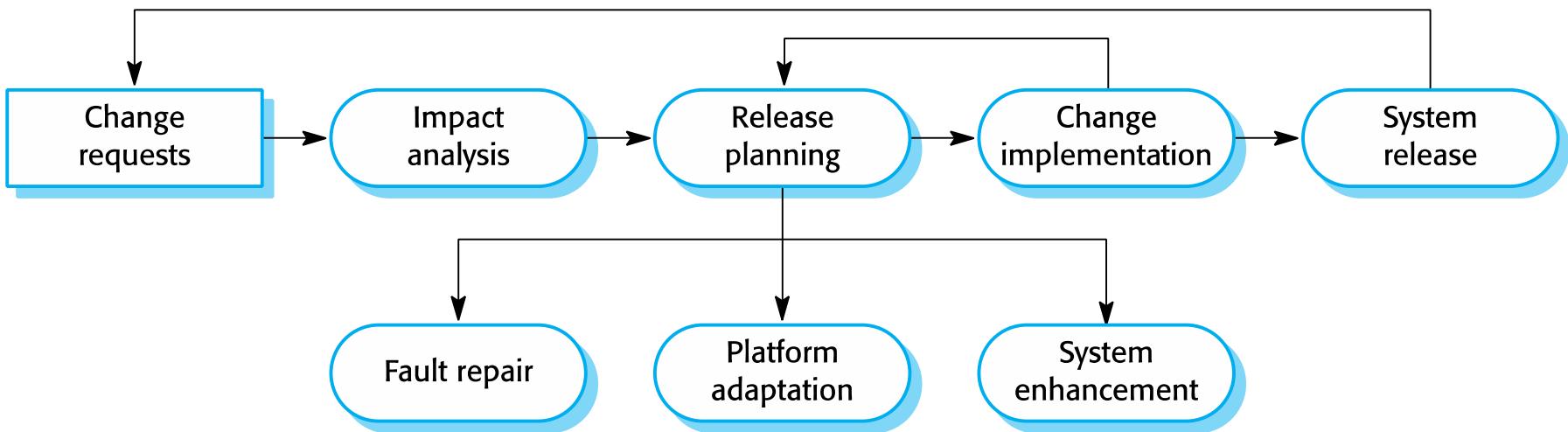
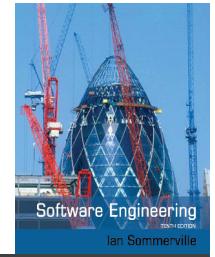


- ✧ Software evolution processes depend on
 - The type of software being maintained;
 - The development processes used;
 - The skills and experience of the people involved.
- ✧ Proposals for change are the driver for system evolution.
 - Should be linked with components that are affected by the change, thus allowing the cost and impact of the change to be estimated.
- ✧ Change identification and evolution continues throughout the system lifetime.

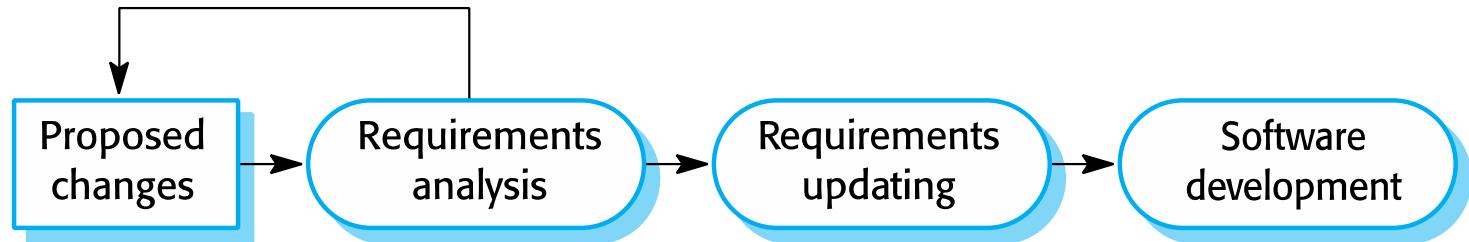
Change identification and evolution processes



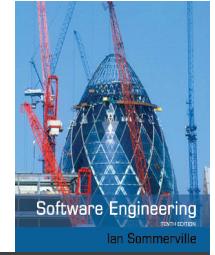
The software evolution process



Change implementation

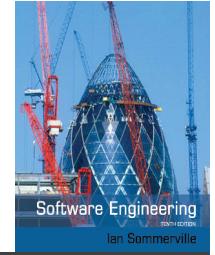


Change implementation



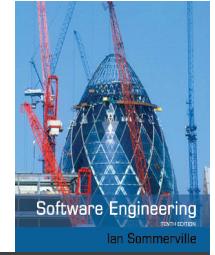
- ✧ Iteration of the development process where the revisions to the system are designed, implemented and tested.
- ✧ A critical difference is that the first stage of change implementation may involve program understanding, especially if the original system developers are not responsible for the change implementation.
- ✧ During the program understanding phase, you have to understand how the program is structured, how it delivers functionality and how the proposed change might affect the program.

Urgent change requests

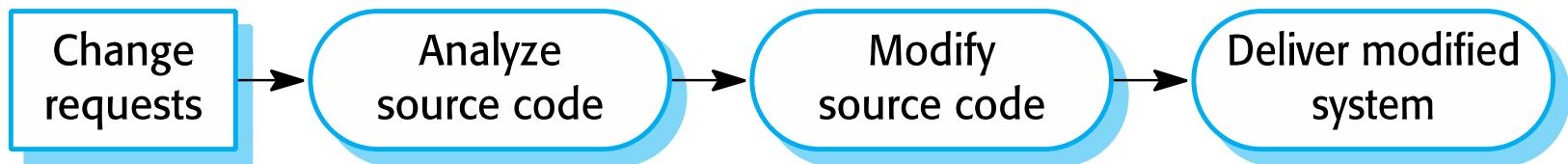


- ✧ Urgent changes may have to be implemented without going through all stages of the software engineering process
 - If a serious system fault has to be repaired to allow normal operation to continue;
 - If changes to the system's environment (e.g. an OS upgrade) have unexpected effects;
 - If there are business changes that require a very rapid response (e.g. the release of a competing product).

The emergency repair process



Software Engineering
Ian Sommerville



Agile methods and evolution



- ✧ Agile methods are based on incremental development so the transition from development to evolution is a seamless one.
 - Evolution is simply a continuation of the development process based on frequent system releases.
- ✧ Automated regression testing is particularly valuable when changes are made to a system.
- ✧ Changes may be expressed as additional user stories.

Handover problems



- ✧ Where the development team have used an agile approach but the evolution team is unfamiliar with agile methods and prefer a plan-based approach.
 - The evolution team may expect detailed documentation to support evolution and this is not produced in agile processes.
- ✧ Where a plan-based approach has been used for development but the evolution team prefer to use agile methods.
 - The evolution team may have to start from scratch developing automated tests and the code in the system may not have been refactored and simplified as is expected in agile development.



Software Engineering

Ian Sommerville

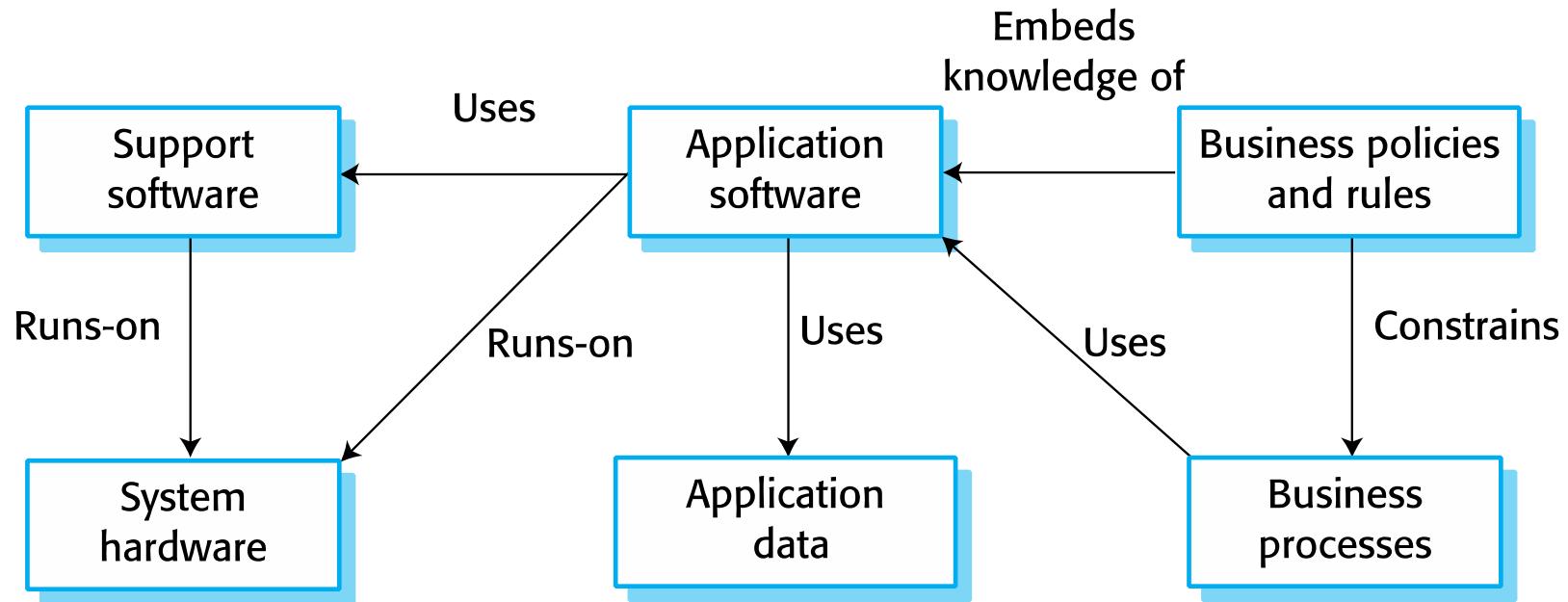
Legacy systems

Legacy systems

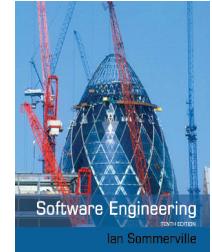


- ✧ Legacy systems are older systems that rely on languages and technology that are no longer used for new systems development.
- ✧ Legacy software may be dependent on older hardware, such as mainframe computers and may have associated legacy processes and procedures.
- ✧ Legacy systems are not just software systems but are broader socio-technical systems that include hardware, software, libraries and other supporting software and business processes.

The elements of a legacy system



Legacy system components



- ✧ *System hardware* Legacy systems may have been written for hardware that is no longer available.
- ✧ *Support software* The legacy system may rely on a range of support software, which may be obsolete or unsupported.
- ✧ *Application software* The application system that provides the business services is usually made up of a number of application programs.
- ✧ *Application data* These are data that are processed by the application system. They may be inconsistent, duplicated or held in different databases.

Legacy system components

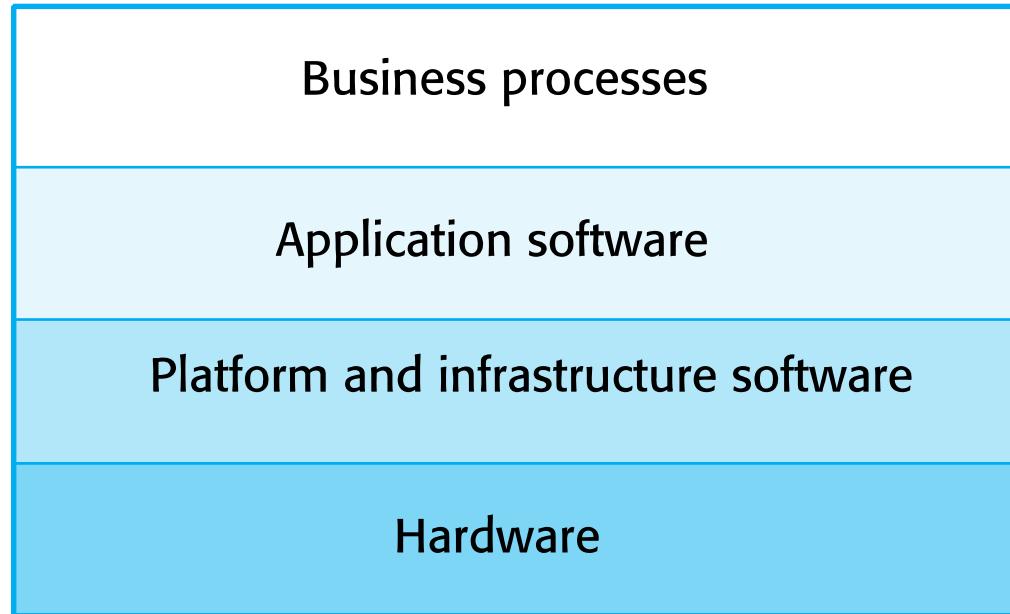


- ✧ *Business processes* These are processes that are used in the business to achieve some business objective.
- ✧ Business processes may be designed around a legacy system and constrained by the functionality that it provides.
- ✧ *Business policies and rules* These are definitions of how the business should be carried out and constraints on the business. Use of the legacy application system may be embedded in these policies and rules.

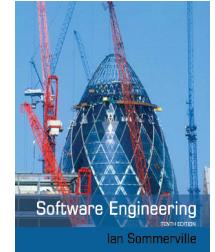
Legacy system layers



Socio-technical system



Legacy system replacement



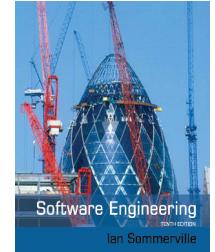
- ✧ Legacy system replacement is risky and expensive so businesses continue to use these systems
- ✧ System replacement is risky for a number of reasons
 - Lack of complete system specification
 - Tight integration of system and business processes
 - Undocumented business rules embedded in the legacy system
 - New software development may be late and/or over budget

Legacy system change



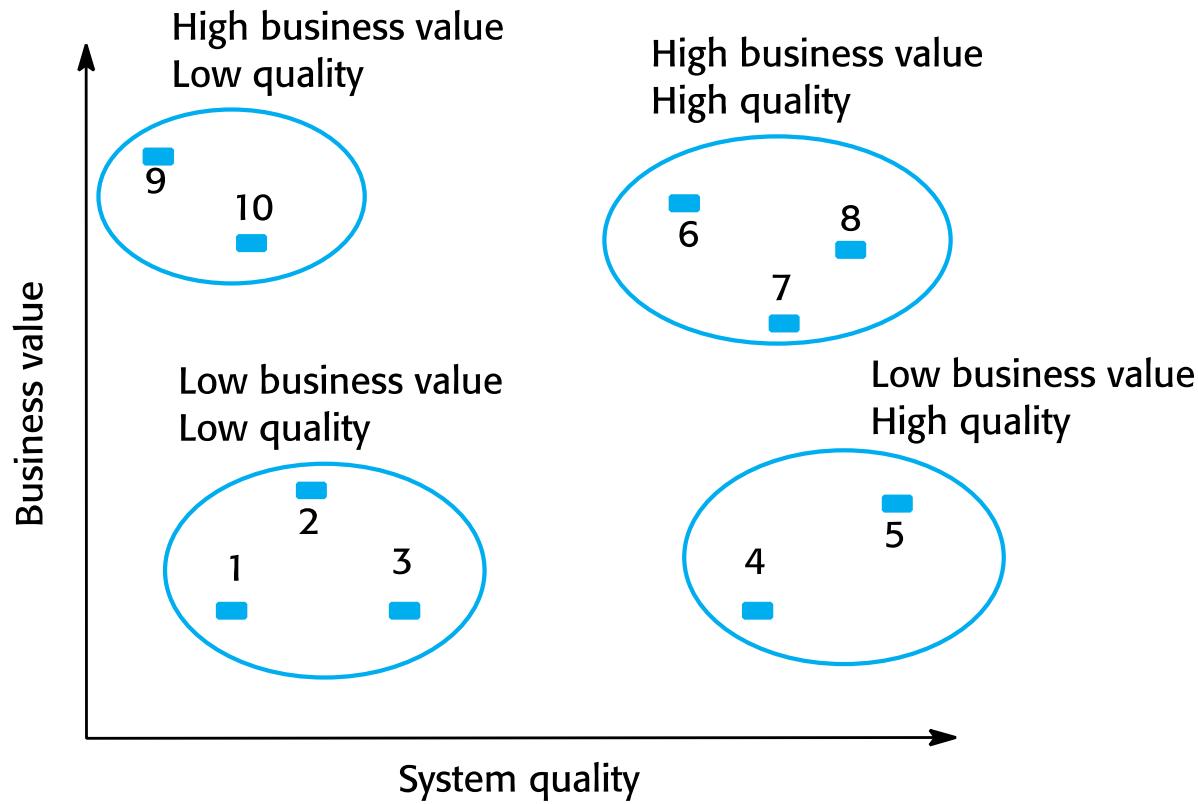
- ✧ Legacy systems are expensive to change for a number of reasons:
 - No consistent programming style
 - Use of obsolete programming languages with few people available with these language skills
 - Inadequate system documentation
 - System structure degradation
 - Program optimizations may make them hard to understand
 - Data errors, duplication and inconsistency

Legacy system management

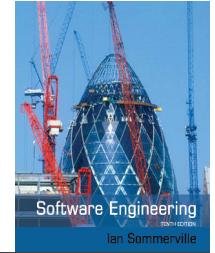


- ✧ Organisations that rely on legacy systems must choose a strategy for evolving these systems
 - Scrap the system completely and modify business processes so that it is no longer required;
 - Continue maintaining the system;
 - Transform the system by re-engineering to improve its maintainability;
 - Replace the system with a new system.
- ✧ The strategy chosen should depend on the system quality and its business value.

Figure 9.13 An example of a legacy system assessment

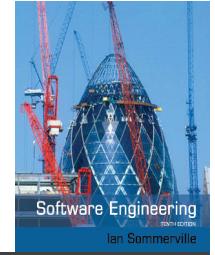


Legacy system categories



- ✧ Low quality, low business value
 - These systems should be scrapped.
- ✧ Low-quality, high-business value
 - These make an important business contribution but are expensive to maintain. Should be re-engineered or replaced if a suitable system is available.
- ✧ High-quality, low-business value
 - Replace with COTS, scrap completely or maintain.
- ✧ High-quality, high business value
 - Continue in operation using normal system maintenance.

Business value assessment



- ✧ Assessment should take different viewpoints into account
 - System end-users;
 - Business customers;
 - Line managers;
 - IT managers;
 - Senior managers.
- ✧ Interview different stakeholders and collate results.

Issues in business value assessment



✧ The use of the system

- If systems are only used occasionally or by a small number of people, they may have a low business value.

✧ The business processes that are supported

- A system may have a low business value if it forces the use of inefficient business processes.

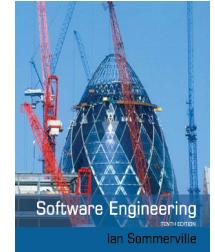
✧ System dependability

- If a system is not dependable and the problems directly affect business customers, the system has a low business value.

✧ The system outputs

- If the business depends on system outputs, then the system has a high business value.

System quality assessment



✧ Business process assessment

- How well does the business process support the current goals of the business?

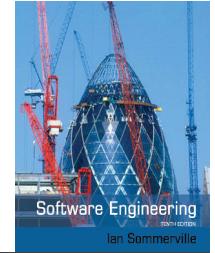
✧ Environment assessment

- How effective is the system's environment and how expensive is it to maintain?

✧ Application assessment

- What is the quality of the application software system?

Business process assessment



- ✧ Use a viewpoint-oriented approach and seek answers from system stakeholders
 - Is there a defined process model and is it followed?
 - Do different parts of the organisation use different processes for the same function?
 - How has the process been adapted?
 - What are the relationships with other business processes and are these necessary?
 - Is the process effectively supported by the legacy application software?
- ✧ Example - a travel ordering system may have a low business value because of the widespread use of web-based ordering.

Factors used in environment assessment



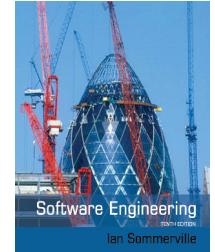
Factor	Questions
Supplier stability	Is the supplier still in existence? Is the supplier financially stable and likely to continue in existence? If the supplier is no longer in business, does someone else maintain the systems?
Failure rate	Does the hardware have a high rate of reported failures? Does the support software crash and force system restarts?
Age	How old is the hardware and software? The older the hardware and support software, the more obsolete it will be. It may still function correctly but there could be significant economic and business benefits to moving to a more modern system.
Performance	Is the performance of the system adequate? Do performance problems have a significant effect on system users?

Factors used in environment assessment



Factor	Questions
Support requirements	What local support is required by the hardware and software? If there are high costs associated with this support, it may be worth considering system replacement.
Maintenance costs	What are the costs of hardware maintenance and support software licences? Older hardware may have higher maintenance costs than modern systems. Support software may have high annual licensing costs.
Interoperability	Are there problems interfacing the system to other systems? Can compilers, for example, be used with current versions of the operating system? Is hardware emulation required?

Factors used in application assessment



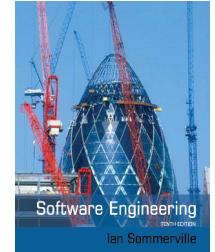
Factor	Questions
Understandability	How difficult is it to understand the source code of the current system? How complex are the control structures that are used? Do variables have meaningful names that reflect their function?
Documentation	What system documentation is available? Is the documentation complete, consistent, and current?
Data	Is there an explicit data model for the system? To what extent is data duplicated across files? Is the data used by the system up to date and consistent?
Performance	Is the performance of the application adequate? Do performance problems have a significant effect on system users?

Factors used in application assessment



Factor	Questions
Programming language	Are modern compilers available for the programming language used to develop the system? Is the programming language still used for new system development?
Configuration management	Are all versions of all parts of the system managed by a configuration management system? Is there an explicit description of the versions of components that are used in the current system?
Test data	Does test data for the system exist? Is there a record of regression tests carried out when new features have been added to the system?
Personnel skills	Are there people available who have the skills to maintain the application? Are there people available who have experience with the system?

System measurement

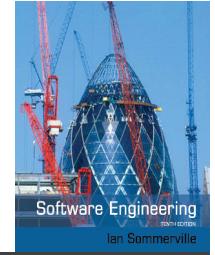


- ✧ You may collect quantitative data to make an assessment of the quality of the application system
 - The number of system change requests; The higher this accumulated value, the lower the quality of the system.
 - The number of different user interfaces used by the system; The more interfaces, the more likely it is that there will be inconsistencies and redundancies in these interfaces.
 - The volume of data used by the system. As the volume of data (number of files, size of database, etc.) processed by the system increases, so too do the inconsistencies and errors in that data.
 - Cleaning up old data is a very expensive and time-consuming process



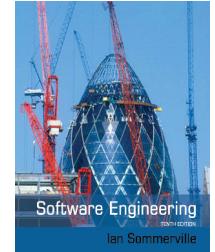
Software maintenance

Software maintenance



- ✧ Modifying a program after it has been put into use.
- ✧ The term is mostly used for changing custom software. Generic software products are said to evolve to create new versions.
- ✧ Maintenance does not normally involve major changes to the system's architecture.
- ✧ Changes are implemented by modifying existing components and adding new components to the system.

Types of maintenance



✧ Fault repairs

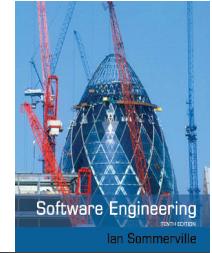
- Changing a system to fix bugs/vulnerabilities and correct deficiencies in the way meets its requirements.

✧ Environmental adaptation

- Maintenance to adapt software to a different operating environment
- Changing a system so that it operates in a different environment (computer, OS, etc.) from its initial implementation.

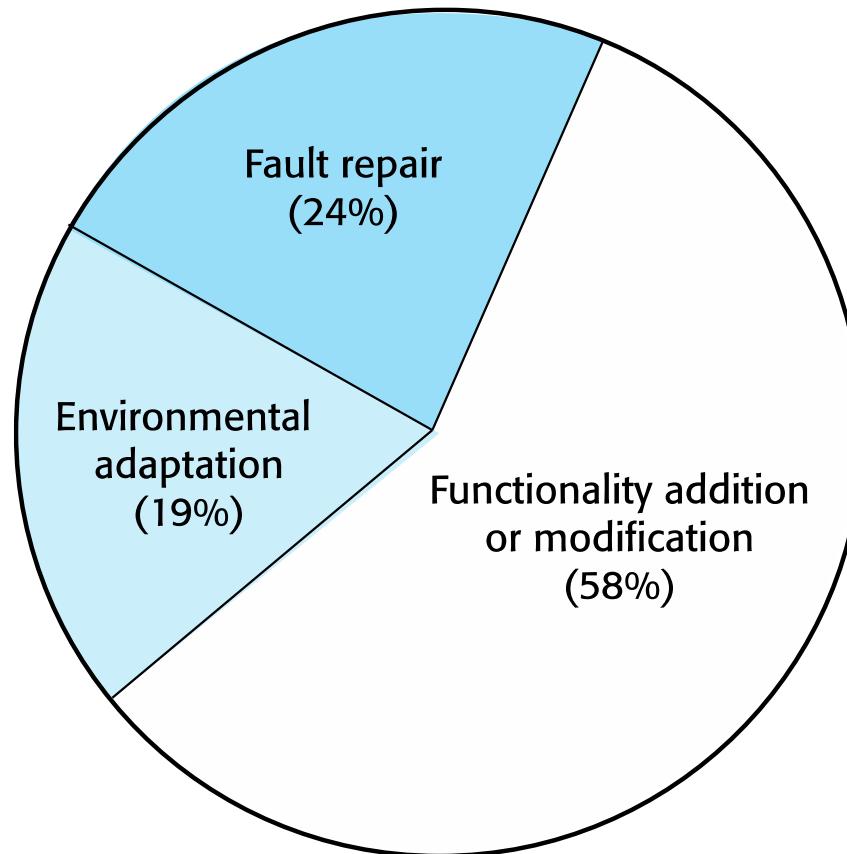
✧ Functionality addition and modification

- Modifying the system to satisfy new requirements.



Maintenance effort distribution

Software Engineering
Ian Sommerville

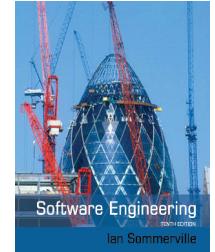


Maintenance costs



- ✧ Usually greater than development costs (2* to 100* depending on the application).
- ✧ Affected by both technical and non-technical factors.
- ✧ Increases as software is maintained.
Maintenance corrupts the software structure so makes further maintenance more difficult.
- ✧ Ageing software can have high support costs (e.g. old languages, compilers etc.).

Maintenance costs



- ✧ It is usually more expensive to add new features to a system during maintenance than it is to add the same features during development
 - A new team has to understand the programs being maintained
 - Separating maintenance and development means there is no incentive for the development team to write maintainable software
 - Program maintenance work is unpopular
 - Maintenance staff are often inexperienced and have limited domain knowledge.
 - As programs age, their structure degrades and they become harder to change

Maintenance prediction



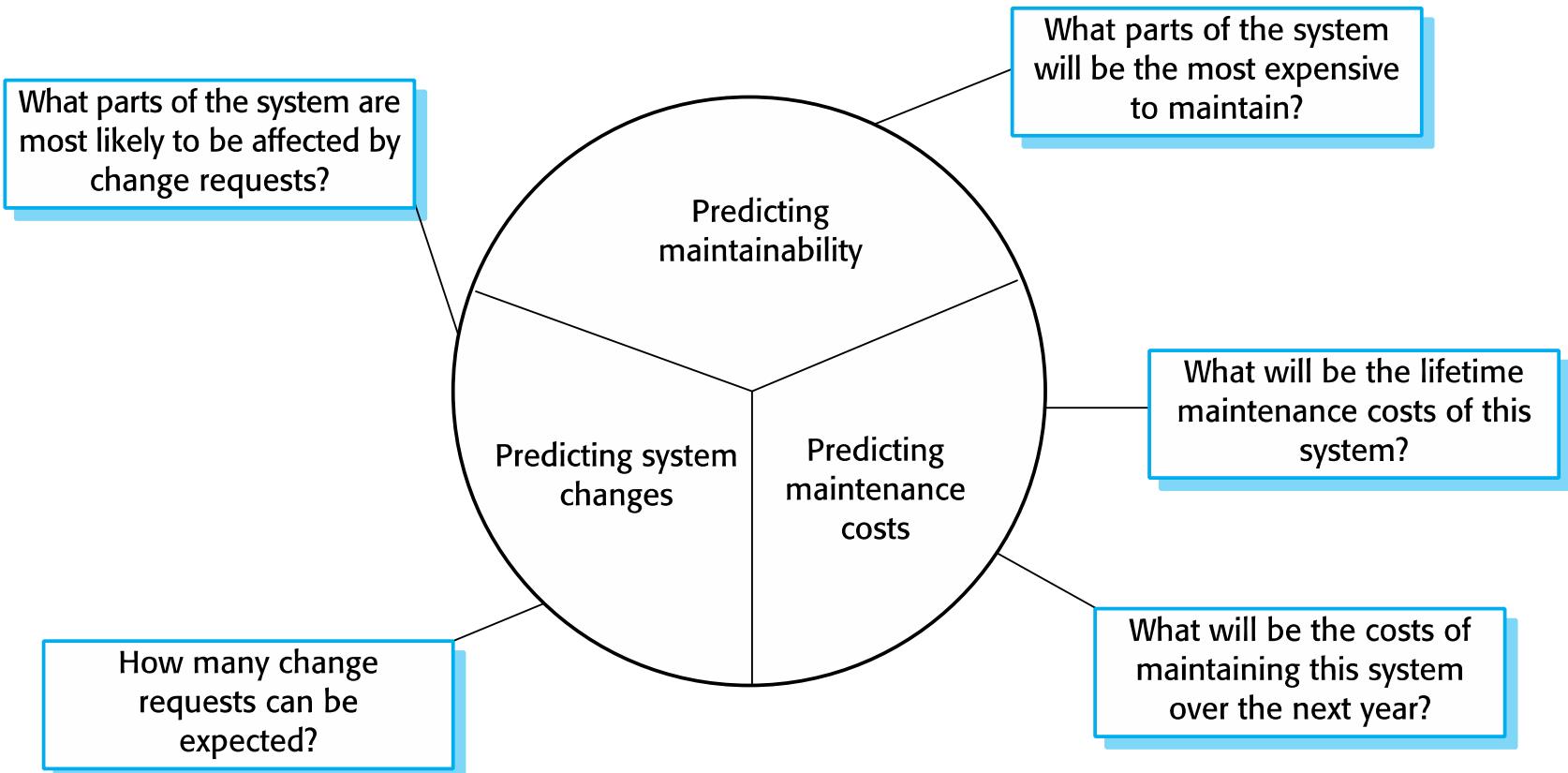
- ✧ Maintenance prediction is concerned with assessing which parts of the system may cause problems and have high maintenance costs
 - Change acceptance depends on the maintainability of the components affected by the change;
 - Implementing changes degrades the system and reduces its maintainability;
 - Maintenance costs depend on the number of changes and costs of change depend on maintainability.

Maintenance prediction

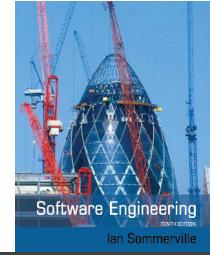


Software Engineering

Ian Sommerville



Change prediction



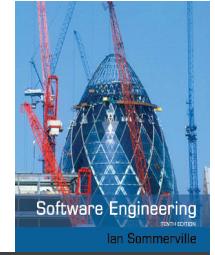
- ✧ Predicting the number of changes requires an understanding of the relationships between a system and its environment.
- ✧ Tightly coupled systems require changes whenever the environment is changed.
- ✧ Factors influencing this relationship are
 - Number and complexity of system interfaces;
 - Number of inherently volatile system requirements;
 - The business processes where the system is used.

Complexity metrics



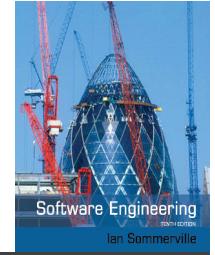
- ✧ Predictions of maintainability can be made by assessing the complexity of system components.
- ✧ Studies have shown that most maintenance effort is spent on a relatively small number of system components.
- ✧ Complexity depends on
 - Complexity of control structures;
 - Complexity of data structures;
 - Object, method (procedure) and module size.

Process metrics



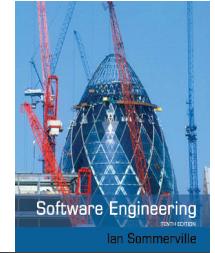
- ✧ Process metrics may be used to assess maintainability
 - Number of requests for corrective maintenance;
 - Average time required for impact analysis;
 - Average time taken to implement a change request;
 - Number of outstanding change requests.
- ✧ If any or all of these is increasing, this may indicate a decline in maintainability.

Software reengineering



- ✧ Restructuring or rewriting part or all of a legacy system without changing its functionality.
- ✧ Applicable where some but not all sub-systems of a larger system require frequent maintenance.
- ✧ Reengineering involves adding effort to make them easier to maintain. The system may be restructured and re-documented.

Advantages of reengineering



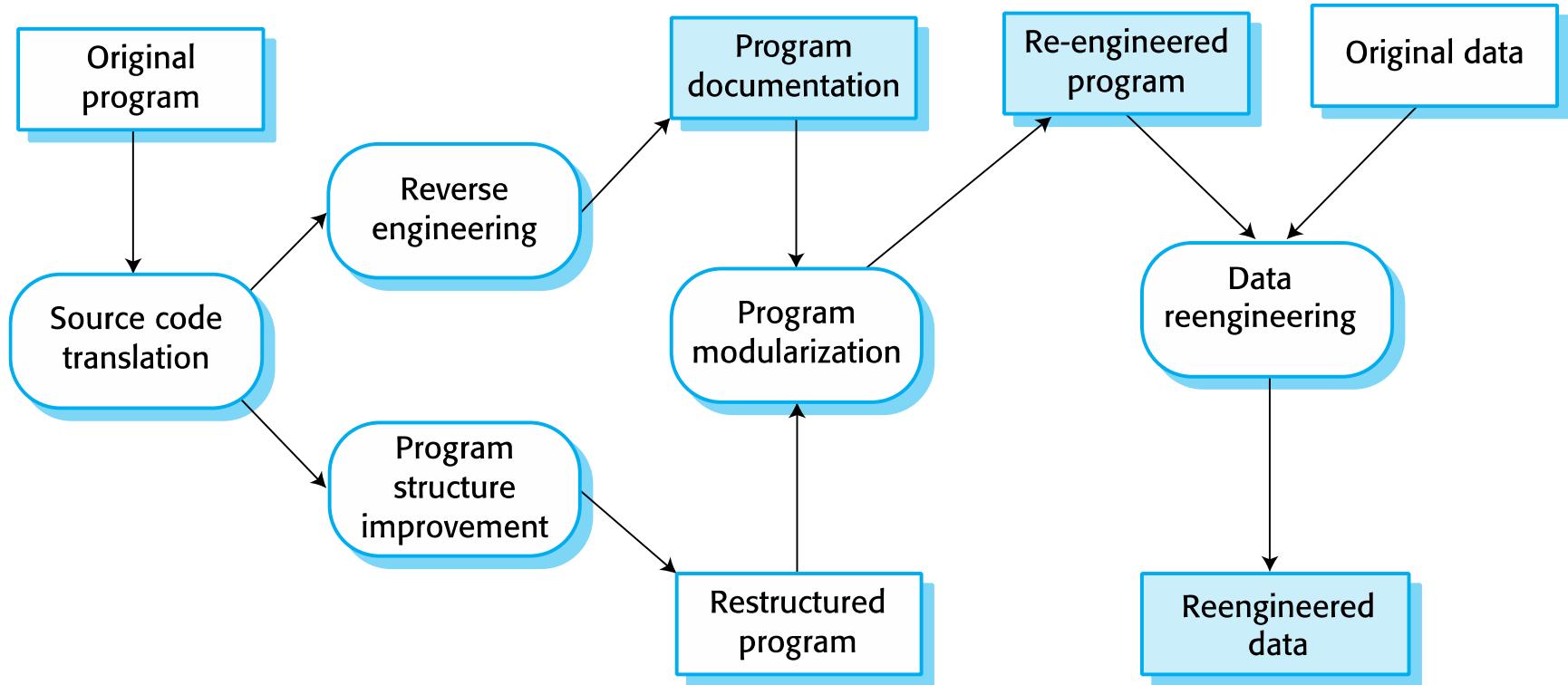
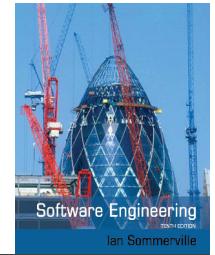
✧ Reduced risk

- There is a high risk in new software development. There may be development problems, staffing problems and specification problems.

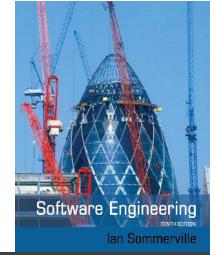
✧ Reduced cost

- The cost of re-engineering is often significantly less than the costs of developing new software.

The reengineering process

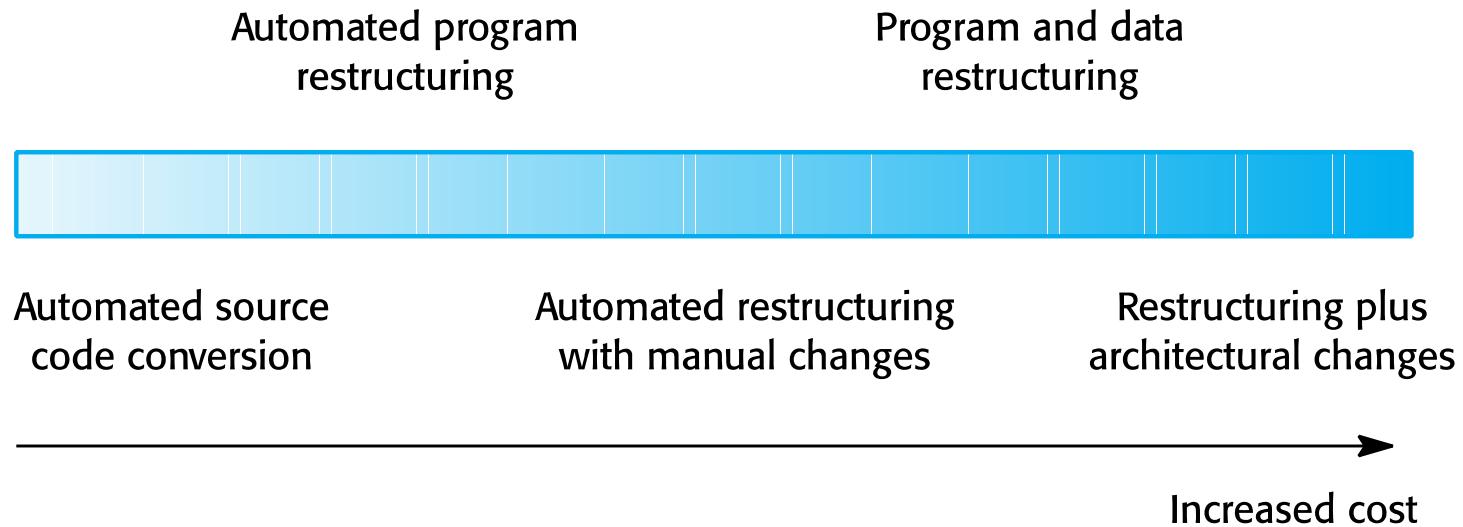


Reengineering process activities

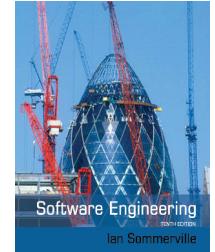


- ✧ Source code translation
 - Convert code to a new language.
- ✧ Reverse engineering
 - Analyse the program to understand it;
- ✧ Program structure improvement
 - Restructure automatically for understandability;
- ✧ Program modularisation
 - Reorganise the program structure;
- ✧ Data reengineering
 - Clean-up and restructure system data.

Reengineering approaches

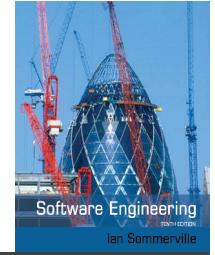


Reengineering cost factors



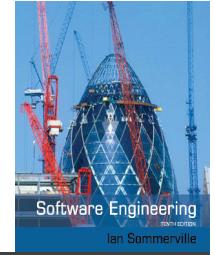
- ✧ The quality of the software to be reengineered.
- ✧ The tool support available for reengineering.
- ✧ The extent of the data conversion which is required.
- ✧ The availability of expert staff for reengineering.
 - This can be a problem with old systems based on technology that is no longer widely used.

Refactoring



- ✧ Refactoring is the process of making improvements to a program to slow down degradation through change.
- ✧ You can think of refactoring as ‘preventative maintenance’ that reduces the problems of future change.
- ✧ Refactoring involves modifying a program to improve its structure, reduce its complexity or make it easier to understand.
- ✧ When you refactor a program, you should not add functionality but rather concentrate on program improvement.

Refactoring and reengineering



- ✧ Re-engineering takes place after a system has been maintained for some time and maintenance costs are increasing. You use automated tools to process and re-engineer a legacy system to create a new system that is more maintainable.
- ✧ Refactoring is a continuous process of improvement throughout the development and evolution process. It is intended to avoid the structure and code degradation that increases the costs and difficulties of maintaining a system.



'Bad smells' in program code

✧ Duplicate code

- The same or very similar code may be included at different places in a program. This can be removed and implemented as a single method or function that is called as required.

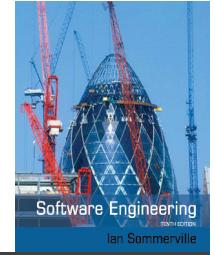
✧ Long methods

- If a method is too long, it should be redesigned as a number of shorter methods.

✧ Switch (case) statements

- These often involve duplication, where the switch depends on the type of a value. The switch statements may be scattered around a program. In object-oriented languages, you can often use polymorphism to achieve the same thing.

‘Bad smells’ in program code



✧ Data clumping

- Data clumps occur when the same group of data items (fields in classes, parameters in methods) re-occur in several places in a program. These can often be replaced with an object that encapsulates all of the data.

✧ Speculative generality

- This occurs when developers include generality in a program in case it is required in the future. This can often simply be removed.

Key points



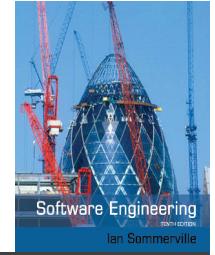
- ✧ Software development and evolution can be thought of as an integrated, iterative process that can be represented using a spiral model.
- ✧ For custom systems, the costs of software maintenance usually exceed the software development costs.
- ✧ The process of software evolution is driven by requests for changes and includes change impact analysis, release planning and change implementation.
- ✧ Legacy systems are older software systems, developed using obsolete software and hardware technologies, that remain useful for a business.

Key points



- ✧ It is often cheaper and less risky to maintain a legacy system than to develop a replacement system using modern technology.
- ✧ The business value of a legacy system and the quality of the application should be assessed to help decide if a system should be replaced, transformed or maintained.
- ✧ There are 3 types of software maintenance, namely bug fixing, modifying software to work in a new environment, and implementing new or changed requirements.

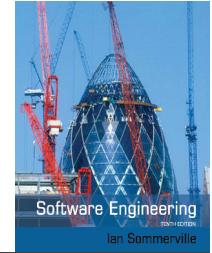
Key points



- ✧ Software re-engineering is concerned with re-structuring and re-documenting software to make it easier to understand and change.
- ✧ Refactoring, making program changes that preserve functionality, is a form of preventative maintenance.



Chapter 10 – Dependable systems



Topics covered

- ✧ Dependability properties
- ✧ Sociotechnical systems
- ✧ Redundancy and diversity
- ✧ Dependable processes
- ✧ Formal methods and dependability

System dependability

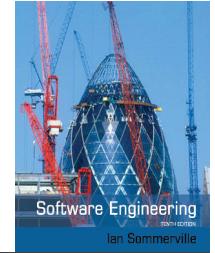


- ✧ For many computer-based systems, the most important system property is the dependability of the system.
- ✧ The dependability of a system reflects the user's degree of trust in that system. It reflects the extent of the user's confidence that it will operate as users expect and that it will not 'fail' in normal use.
- ✧ Dependability covers the related systems attributes of reliability, availability and security. These are all inter-dependent.

Importance of dependability



- ✧ System failures may have widespread effects with large numbers of people affected by the failure.
- ✧ Systems that are not dependable and are unreliable, unsafe or insecure may be rejected by their users.
- ✧ The costs of system failure may be very high if the failure leads to economic losses or physical damage.
- ✧ Undependable systems may cause information loss with a high consequent recovery cost.



Causes of failure

Software Engineering
Ian Sommerville

✧ Hardware failure

- Hardware fails because of design and manufacturing errors or because components have reached the end of their natural life.

✧ Software failure

- Software fails due to errors in its specification, design or implementation.

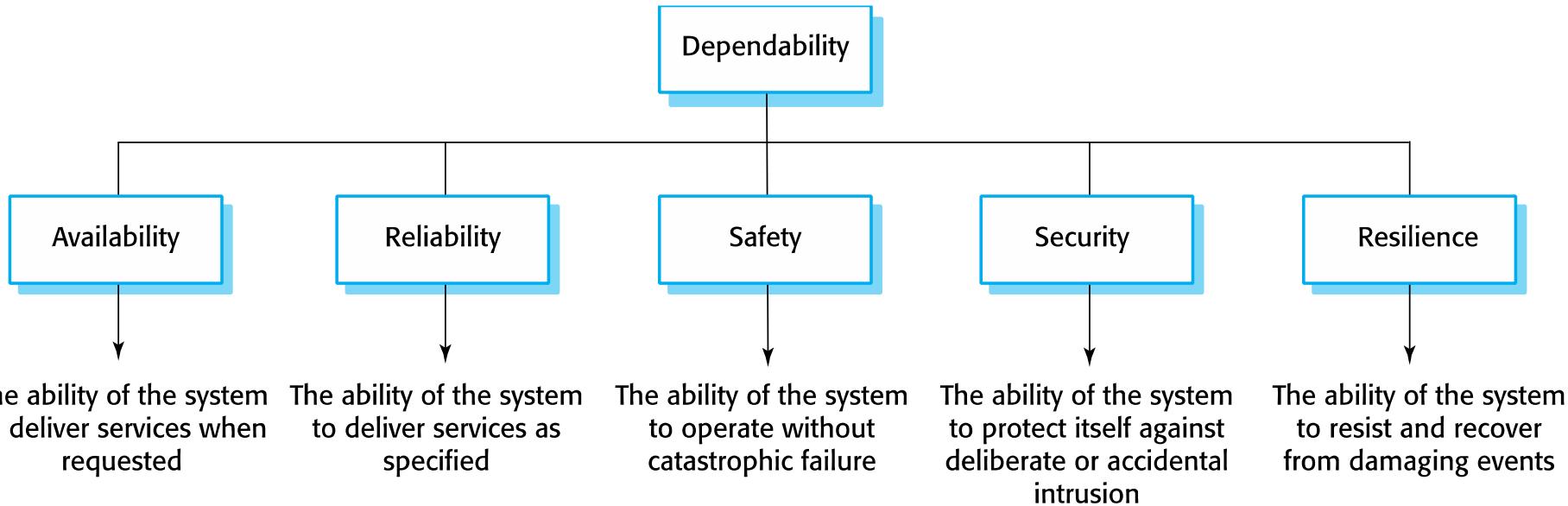
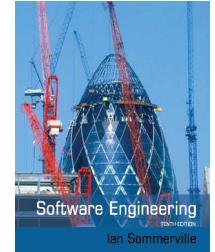
✧ Operational failure

- Human operators make mistakes. Now perhaps the largest single cause of system failures in socio-technical systems.

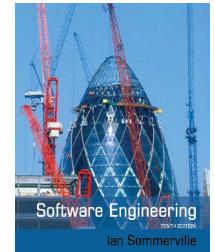


Dependability properties

The principal dependability properties



Principal properties



✧ Availability

- The probability that the system will be up and running and able to deliver useful services to users.

✧ Reliability

- The probability that the system will correctly deliver services as expected by users.

✧ Safety

- A judgment of how likely it is that the system will cause damage to people or its environment.

Principal properties



✧ Security

- A judgment of how likely it is that the system can resist accidental or deliberate intrusions.

✧ Resilience

- A judgment of how well a system can maintain the continuity of its critical services in the presence of disruptive events such as equipment failure and cyberattacks.

Other dependability properties



✧ Repairability

- Reflects the extent to which the system can be repaired in the event of a failure

✧ Maintainability

- Reflects the extent to which the system can be adapted to new requirements;

✧ Error tolerance

- Reflects the extent to which user input errors can be avoided and tolerated.

Dependability attribute dependencies



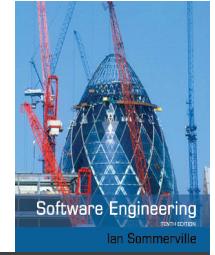
- ✧ Safe system operation depends on the system being available and operating reliably.
- ✧ A system may be unreliable because its data has been corrupted by an external attack.
- ✧ Denial of service attacks on a system are intended to make it unavailable.
- ✧ If a system is infected with a virus, you cannot be confident in its reliability or safety.

Dependability achievement



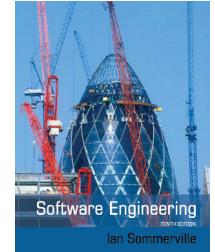
- ✧ Avoid the introduction of accidental errors when developing the system.
- ✧ Design V & V processes that are effective in discovering residual errors in the system.
- ✧ Design systems to be fault tolerant so that they can continue in operation when faults occur
- ✧ Design protection mechanisms that guard against external attacks.

Dependability achievement



- ✧ Configure the system correctly for its operating environment.
- ✧ Include system capabilities to recognise and resist cyberattacks.
- ✧ Include recovery mechanisms to help restore normal system service after a failure.

Dependability costs

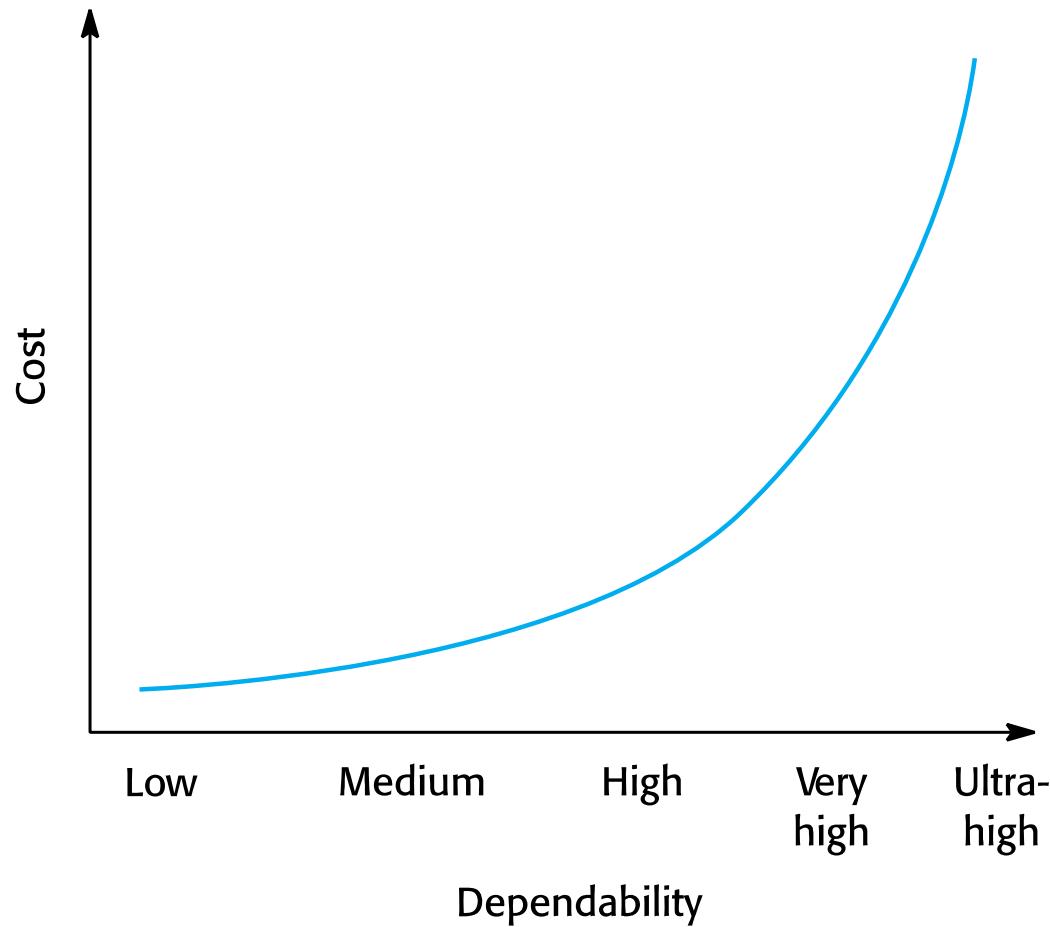


- ✧ Dependability costs tend to increase exponentially as increasing levels of dependability are required.
- ✧ There are two reasons for this
 - The use of more expensive development techniques and hardware that are required to achieve the higher levels of dependability.
 - The increased testing and system validation that is required to convince the system client and regulators that the required levels of dependability have been achieved.

Cost/dependability curve



Software Engineering
Ian Sommerville



Dependability economics

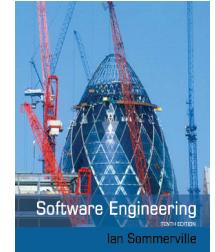


- ✧ Because of very high costs of dependability achievement, it may be more cost effective to accept untrustworthy systems and pay for failure costs
- ✧ However, this depends on social and political factors. A reputation for products that can't be trusted may lose future business
- ✧ Depends on system type - for business systems in particular, modest levels of dependability may be adequate



Sociotechnical systems

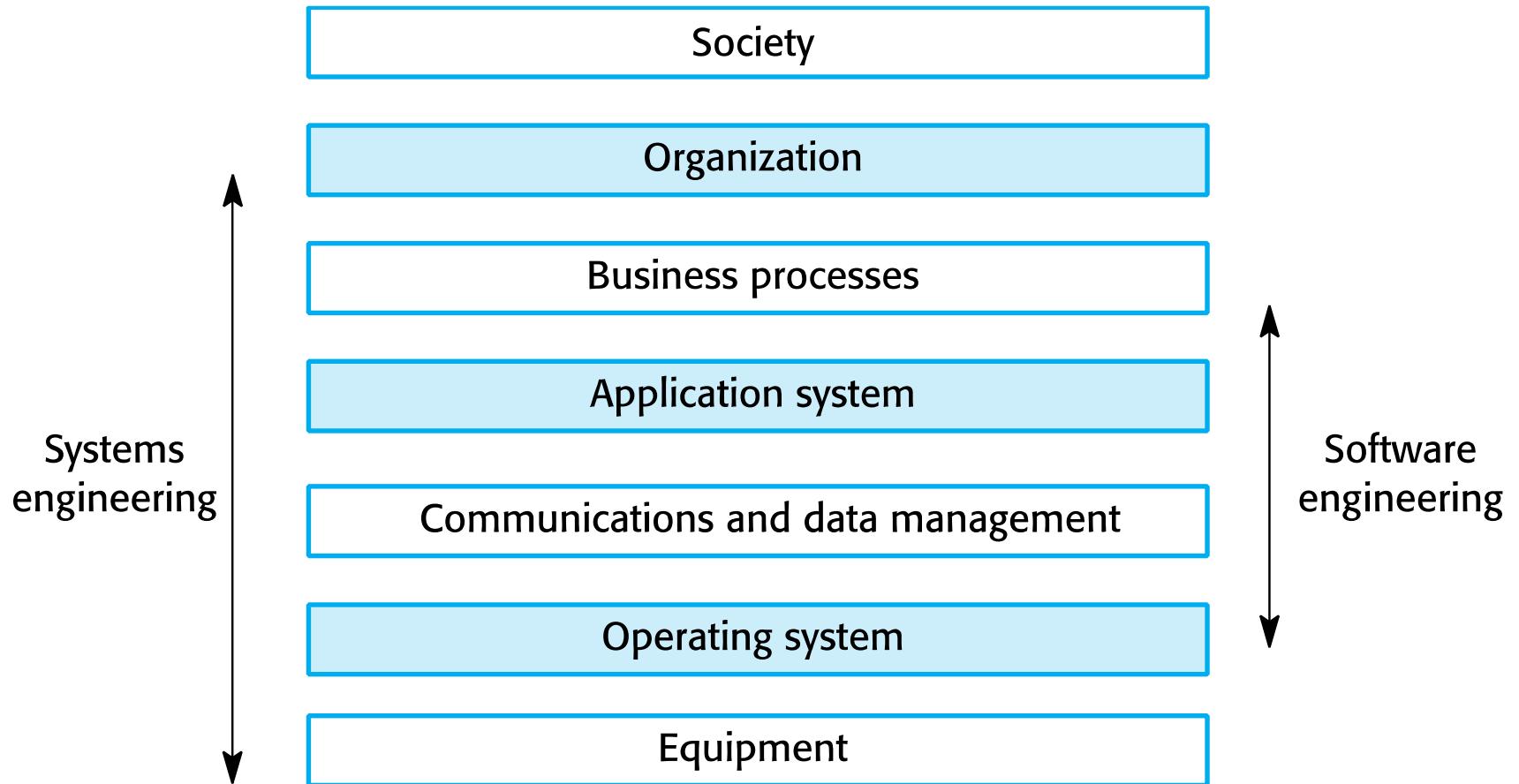
Systems and software



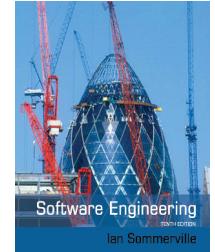
- ✧ Software engineering is not an isolated activity but is part of a broader systems engineering process.
- ✧ Software systems are therefore not isolated systems but are essential components of broader systems that have a human, social or organizational purpose.
- ✧ Example
 - The wilderness weather system is part of broader weather recording and forecasting systems
 - These include hardware and software, forecasting processes, system users, the organizations that depend on weather forecasts, etc.



The sociotechnical systems stack



Layers in the STS stack



✧ Equipment

- Hardware devices, some of which may be computers. Most devices will include an embedded system of some kind.

✧ Operating system

- Provides a set of common facilities for higher levels in the system.

✧ Communications and data management

- Middleware that provides access to remote systems and databases.

✧ Application systems

- Specific functionality to meet some organization requirements.



Layers in the STS stack

✧ Business processes

- A set of processes involving people and computer systems that support the activities of the business.

✧ Organizations

- Higher level strategic business activities that affect the operation of the system.

✧ Society

- Laws, regulation and culture that affect the operation of the system.

Holistic system design



Software Engineering
Ian Sommerville

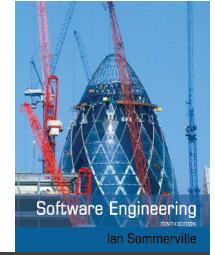
- ✧ There are interactions and dependencies between the layers in a system and changes at one level ripple through the other levels
 - Example: Change in regulations (society) leads to changes in business processes and application software.
- ✧ For dependability, a systems perspective is essential
 - Contain software failures within the enclosing layers of the STS stack.
 - Understand how faults and failures in adjacent layers may affect the software in a system.

Regulation and compliance



- ✧ The general model of economic organization that is now almost universal in the world is that privately owned companies offer goods and services and make a profit on these.
- ✧ To ensure the safety of their citizens, most governments regulate (limit the freedom of) privately owned companies so that they must follow certain standards to ensure that their products are safe and secure.

Regulated systems



- ✧ Many critical systems are regulated systems, which means that their use must be approved by an external regulator before the systems go into service.
 - Nuclear systems
 - Air traffic control systems
 - Medical devices
- ✧ A safety and dependability case has to be approved by the regulator. Therefore, critical systems development has to create the evidence to convince a regulator that the system is dependable, safe and secure.

Safety regulation



- ✧ Regulation and compliance (following the rules) applies to the sociotechnical system as a whole and not simply the software element of that system.
- ✧ Safety-related systems may have to be certified as safe by the regulator.
- ✧ To achieve certification, companies that are developing safety-critical systems have to produce an extensive safety case that shows that rules and regulations have been followed.
- ✧ It can be as expensive develop the documentation for certification as it is to develop the system itself.

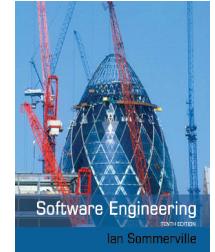


Software Engineering

Ian Sommerville

Redundancy and diversity

Redundancy and diversity



✧ Redundancy

- Keep more than a single version of critical components so that if one fails then a backup is available.

✧ Diversity

- Provide the same functionality in different ways in different components so that they will not fail in the same way.

✧ Redundant and diverse components should be independent so that they will not suffer from ‘common-mode’ failures

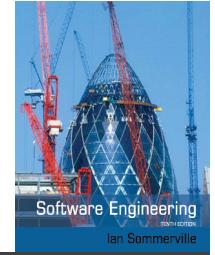
- For example, components implemented in different programming languages means that a compiler fault will not affect all of them.

Diversity and redundancy examples



- ✧ **Redundancy.** Where availability is critical (e.g. in e-commerce systems), companies normally keep backup servers and switch to these automatically if failure occurs.
- ✧ **Diversity.** To provide resilience against external attacks, different servers may be implemented using different operating systems (e.g. Windows and Linux)

Process diversity and redundancy



- ✧ Process activities, such as validation, should not depend on a single approach, such as testing, to validate the system.
- ✧ Redundant and diverse process activities are important especially for verification and validation.
- ✧ Multiple, different process activities that complement each other and allow for cross-checking help to avoid process errors, which may lead to errors in the software.

Problems with redundancy and diversity

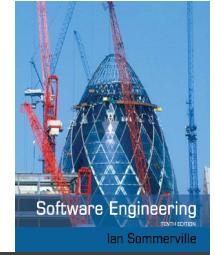


- ✧ Adding diversity and redundancy to a system increases the system complexity.
- ✧ This can increase the chances of error because of unanticipated interactions and dependencies between the redundant system components.
- ✧ Some engineers therefore advocate simplicity and extensive V & V as a more effective route to software dependability.
- ✧ Airbus FCS architecture is redundant/diverse; Boeing 777 FCS architecture has no software diversity



Dependable processes

Dependable processes



- ✧ To ensure a minimal number of software faults, it is important to have a well-defined, repeatable software process.
- ✧ A well-defined repeatable process is one that does not depend entirely on individual skills; rather can be enacted by different people.
- ✧ Regulators use information about the process to check if good software engineering practice has been used.
- ✧ For fault detection, it is clear that the process activities should include significant effort devoted to verification and validation.

Dependable process characteristics



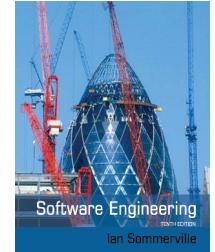
✧ Explicitly defined

- A process that has a defined process model that is used to drive the software production process. Data must be collected during the process that proves that the development team has followed the process as defined in the process model.

✧ Repeatable

- A process that does not rely on individual interpretation and judgment. The process can be repeated across projects and with different team members, irrespective of who is involved in the development.

Attributes of dependable processes



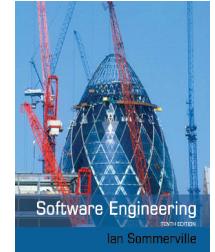
Process characteristic	Description
Auditable	The process should be understandable by people apart from process participants, who can check that process standards are being followed and make suggestions for process improvement.
Diverse	The process should include redundant and diverse verification and validation activities.
Documentable	The process should have a defined process model that sets out the activities in the process and the documentation that is to be produced during these activities.
Robust	The process should be able to recover from failures of individual process activities.
Standardized	A comprehensive set of software development standards covering software production and documentation should be available.

Dependable process activities



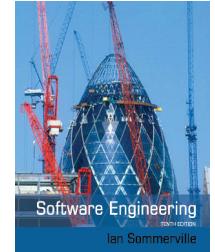
- ✧ Requirements reviews to check that the requirements are, as far as possible, complete and consistent.
- ✧ Requirements management to ensure that changes to the requirements are controlled and that the impact of proposed requirements changes is understood.
- ✧ Formal specification, where a mathematical model of the software is created and analyzed.
- ✧ System modeling, where the software design is explicitly documented as a set of graphical models, and the links between the requirements and these models are documented.

Dependable process activities



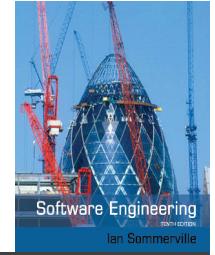
- ✧ Design and program inspections, where the different descriptions of the system are inspected and checked by different people.
- ✧ Static analysis, where automated checks are carried out on the source code of the program.
- ✧ Test planning and management, where a comprehensive set of system tests is designed.
 - The testing process has to be carefully managed to demonstrate that these tests provide coverage of the system requirements and have been correctly applied in the testing process.

Dependable processes and agility



- ✧ Dependable software often requires certification so both process and product documentation has to be produced.
- ✧ Up-front requirements analysis is also essential to discover requirements and requirements conflicts that may compromise the safety and security of the system.
- ✧ These conflict with the general approach in agile development of co-development of the requirements and the system and minimizing documentation.

Dependable processes and agility

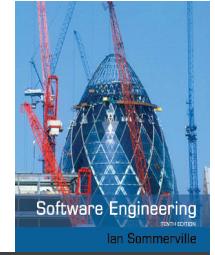


- ✧ An agile process may be defined that incorporates techniques such as iterative development, test-first development and user involvement in the development team.
- ✧ So long as the team follows that process and documents their actions, agile methods can be used.
- ✧ However, additional documentation and planning is essential so 'pure agile' is impractical for dependable systems engineering.

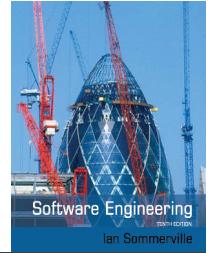


Formal methods and dependability

Formal specification



- ✧ Formal methods are approaches to software development that are based on mathematical representation and analysis of software.
- ✧ Formal methods include
 - Formal specification;
 - Specification analysis and proof;
 - Transformational development;
 - Program verification.
- ✧ Formal methods significantly reduce some types of programming errors and can be cost-effective for dependable systems engineering.



Formal approaches

Software Engineering
Ian Sommerville

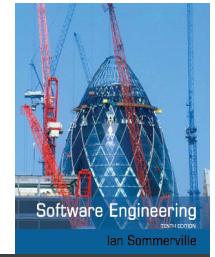
✧ Verification-based approaches

- Different representations of a software system such as a specification and a program implementing that specification are proved to be equivalent.
- This demonstrates the absence of implementation errors.

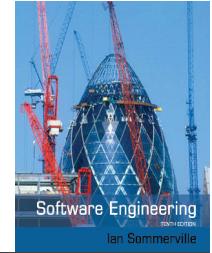
✧ Refinement-based approaches

- A representation of a system is systematically transformed into another, lower-level representation e.g. a specification is transformed automatically into an implementation.
- This means that, if the transformation is correct, the representations are equivalent.

Use of formal methods



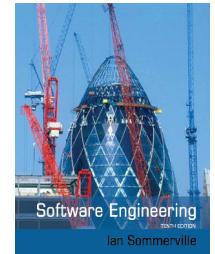
- ✧ The principal benefits of formal methods are in reducing the number of faults in systems.
- ✧ Consequently, their main area of applicability is in dependable systems engineering. There have been several successful projects where formal methods have been used in this area.
- ✧ In this area, the use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



Classes of error

- ✧ Specification and design errors and omissions.
 - Developing and analysing a formal model of the software may reveal errors and omissions in the software requirements. If the model is generated automatically or systematically from source code, analysis using model checking can find undesirable states that may occur such as deadlock in a concurrent system.
- ✧ Inconsistencies between a specification and a program.
 - If a refinement method is used, mistakes made by developers that make the software inconsistent with the specification are avoided. Program proving discovers inconsistencies between a program and its specification.

Benefits of formal specification

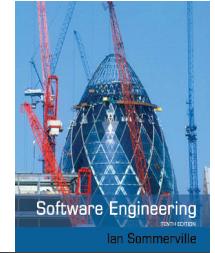


- ✧ Developing a formal specification requires the system requirements to be analyzed in detail. This helps to detect problems, inconsistencies and incompleteness in the requirements.
- ✧ As the specification is expressed in a formal language, it can be automatically analyzed to discover inconsistencies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

Acceptance of formal methods



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



Key points

- ✧ System dependability is important because failure of critical systems can lead to economic losses, information loss, physical damage or threats to human life.
- ✧ The dependability of a computer system is a system property that reflects the user's degree of trust in the system. The most important dimensions of dependability are availability, reliability, safety, security and resilience.
- ✧ Sociotechnical systems include computer hardware, software and people, and are situated within an organization. They are designed to support organizational or business goals and objectives.

Key points

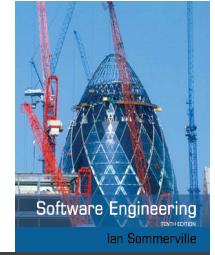


- ✧ The use of a dependable, repeatable process is essential if faults in a system are to be minimized. The process should include verification and validation activities at all stages, from requirements definition through to system implementation.
- ✧ The use of redundancy and diversity in hardware, software processes and software systems is essential to the development of dependable systems.
- ✧ Formal methods, where a formal model of a system is used as a basis for development help reduce the number of specification and implementation errors in a system.



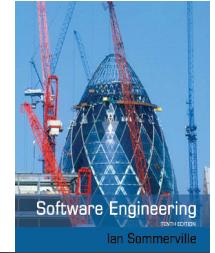
Chapter 11 – Reliability Engineering

Topics covered



- ✧ Availability and reliability
- ✧ Reliability requirements
- ✧ Fault-tolerant architectures
- ✧ Programming for reliability
- ✧ Reliability measurement

Software reliability

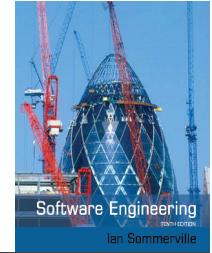


- ✧ In general, software customers expect all software to be dependable. However, for non-critical applications, they may be willing to accept some system failures.
- ✧ Some applications (critical systems) have very high reliability requirements and special software engineering techniques may be used to achieve this.
 - Medical systems
 - Telecommunications and power systems
 - Aerospace systems

Faults, errors and failures



Term	Description
Human error or mistake	Human behavior that results in the introduction of faults into a system. For example, in the wilderness weather system, a programmer might decide that the way to compute the time for the next transmission is to add 1 hour to the current time. This works except when the transmission time is between 23.00 and midnight (midnight is 00.00 in the 24-hour clock).
System fault	A characteristic of a software system that can lead to a system error. The fault is the inclusion of the code to add 1 hour to the time of the last transmission, without a check if the time is greater than or equal to 23.00.
System error	An erroneous system state that can lead to system behavior that is unexpected by system users. The value of transmission time is set incorrectly (to 24.XX rather than 00.XX) when the faulty code is executed.
System failure	An event that occurs at some point in time when the system does not deliver a service as expected by its users. No weather data is transmitted because the time is invalid.



Faults and failures

- ✧ Failures are usually a result of system errors that are derived from faults in the system
- ✧ However, faults do not necessarily result in system errors
 - The erroneous system state resulting from the fault may be transient and ‘corrected’ before an error arises.
 - The faulty code may never be executed.
- ✧ Errors do not necessarily lead to system failures
 - The error can be corrected by built-in error detection and recovery
 - The failure can be protected against by built-in protection facilities. These may, for example, protect system resources from system errors

Fault management



✧ Fault avoidance

- The system is developed in such a way that human error is avoided and thus system faults are minimised.
- The development process is organised so that faults in the system are detected and repaired before delivery to the customer.

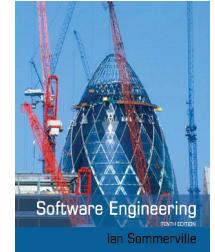
✧ Fault detection

- Verification and validation techniques are used to discover and remove faults in a system before it is deployed.

✧ Fault tolerance

- The system is designed so that faults in the delivered software do not result in system failure.

Reliability achievement



Software Engineering
Ian Sommerville

✧ Fault avoidance

- Development techniques are used that either minimise the possibility of mistakes or trap mistakes before they result in the introduction of system faults.

✧ Fault detection and removal

- Verification and validation techniques are used that increase the probability of detecting and correcting errors before the system goes into service are used.

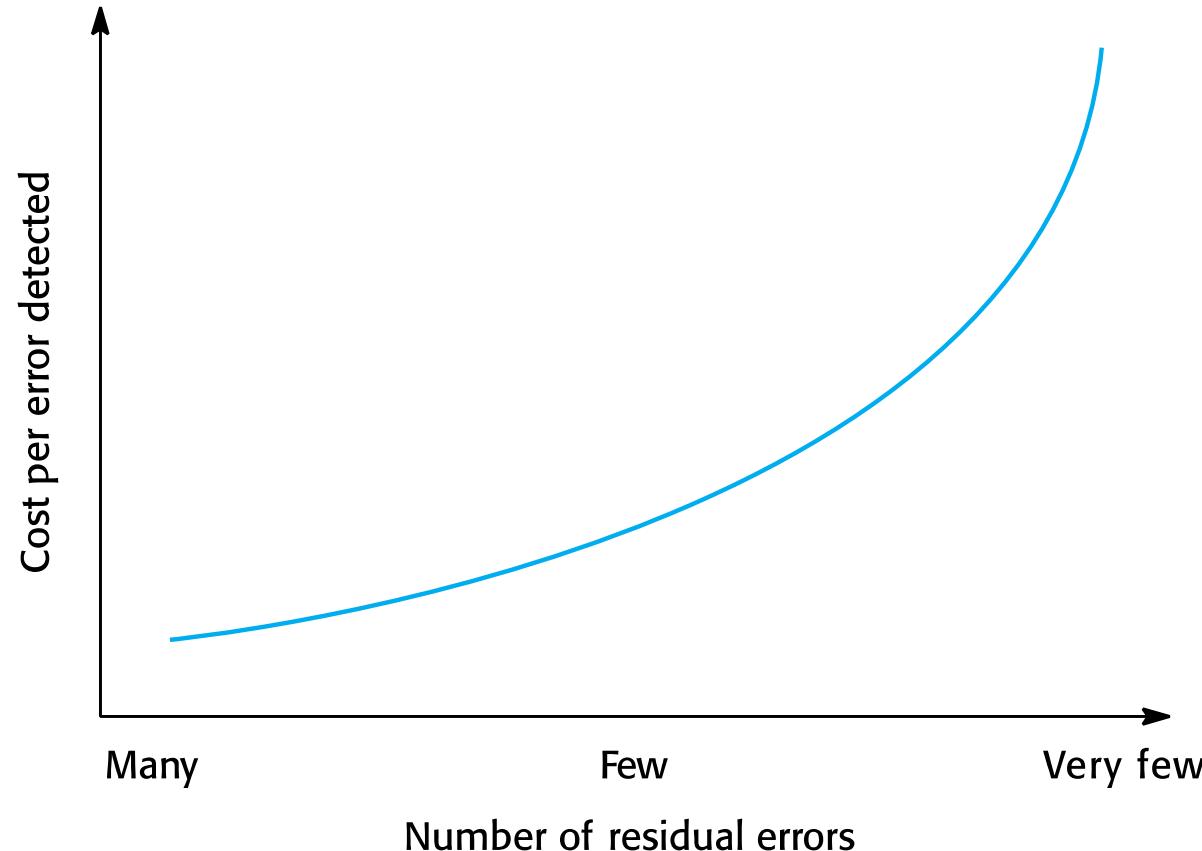
✧ Fault tolerance

- Run-time techniques are used to ensure that system faults do not result in system errors and/or that system errors do not lead to system failures.

The increasing costs of residual fault removal



Software Engineering
Ian Sommerville





Availability and reliability



Availability and reliability

✧ Reliability

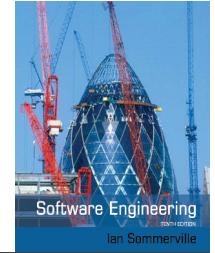
- The probability of failure-free system operation over a specified time in a given environment for a given purpose

✧ Availability

- The probability that a system, at a point in time, will be operational and able to deliver the requested services

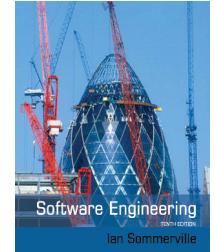
✧ Both of these attributes can be expressed quantitatively
e.g. availability of 0.999 means that the system is up and running for 99.9% of the time.

Reliability and specifications



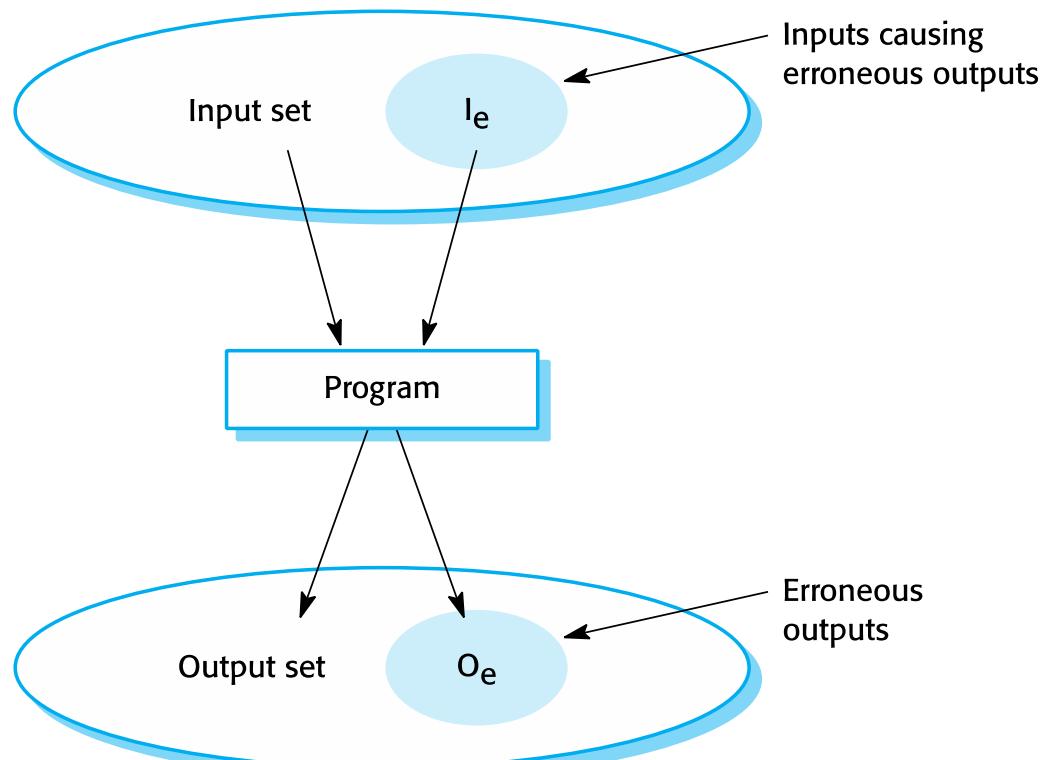
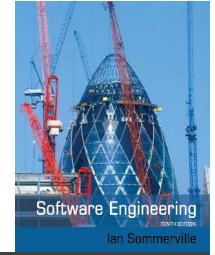
- ✧ Reliability can only be defined formally with respect to a system specification i.e. a failure is a deviation from a specification.
- ✧ However, many specifications are incomplete or incorrect – hence, a system that conforms to its specification may ‘fail’ from the perspective of system users.
- ✧ Furthermore, users don’t read specifications so don’t know how the system is supposed to behave.
- ✧ Therefore perceived reliability is more important in practice.

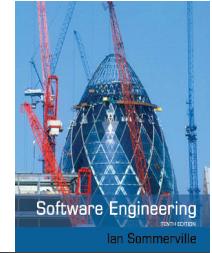
Perceptions of reliability



- ✧ The formal definition of reliability does not always reflect the user's perception of a system's reliability
 - The assumptions that are made about the environment where a system will be used may be incorrect
 - Usage of a system in an office environment is likely to be quite different from usage of the same system in a university environment
 - The consequences of system failures affects the perception of reliability
 - Unreliable windscreen wipers in a car may be irrelevant in a dry climate
 - Failures that have serious consequences (such as an engine breakdown in a car) are given greater weight by users than failures that are inconvenient

A system as an input/output mapping

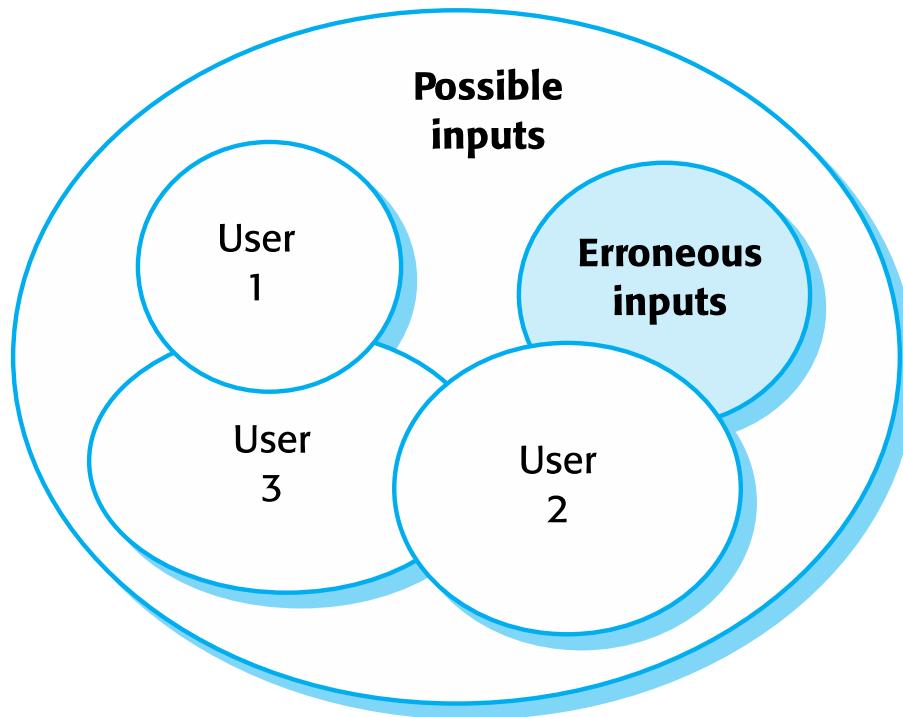




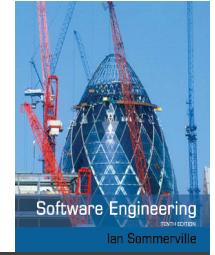
Availability perception

- ✧ Availability is usually expressed as a percentage of the time that the system is available to deliver services e.g. 99.95%.
- ✧ However, this does not take into account two factors:
 - The number of users affected by the service outage. Loss of service in the middle of the night is less important for many systems than loss of service during peak usage periods.
 - The length of the outage. The longer the outage, the more the disruption. Several short outages are less likely to be disruptive than 1 long outage. Long repair times are a particular problem.

Software usage patterns



Reliability in use



- ✧ Removing X% of the faults in a system will not necessarily improve the reliability by X%.
- ✧ Program defects may be in rarely executed sections of the code so may never be encountered by users. Removing these does not affect the perceived reliability.
- ✧ Users adapt their behaviour to avoid system features that may fail for them.
- ✧ A program with known faults may therefore still be perceived as reliable by its users.



Software Engineering

Ian Sommerville

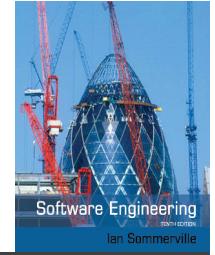
Reliability requirements

System reliability requirements



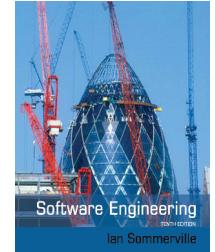
- ✧ Functional reliability requirements define system and software functions that avoid, detect or tolerate faults in the software and so ensure that these faults do not lead to system failure.
- ✧ Software reliability requirements may also be included to cope with hardware failure or operator error.
- ✧ Reliability is a measurable system attribute so non-functional reliability requirements may be specified quantitatively. These define the number of failures that are acceptable during normal use of the system or the time in which the system must be available.

Reliability metrics



- ✧ Reliability metrics are units of measurement of system reliability.
- ✧ System reliability is measured by counting the number of operational failures and, where appropriate, relating these to the demands made on the system and the time that the system has been operational.
- ✧ A long-term measurement programme is required to assess the reliability of critical systems.
- ✧ Metrics
 - Probability of failure on demand
 - Rate of occurrence of failures/Mean time to failure
 - Availability

Probability of failure on demand (POFOD)



- ✧ This is the probability that the system will fail when a service request is made. Useful when demands for service are intermittent and relatively infrequent.
- ✧ Appropriate for protection systems where services are demanded occasionally and where there are serious consequence if the service is not delivered.
- ✧ Relevant for many safety-critical systems with exception management components
 - Emergency shutdown system in a chemical plant.

Rate of fault occurrence (ROCOF)

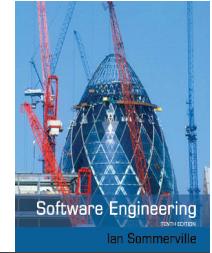


- ✧ Reflects the rate of occurrence of failure in the system.
- ✧ ROCOF of 0.002 means 2 failures are likely in each 1000 operational time units e.g. 2 failures per 1000 hours of operation.
- ✧ Relevant for systems where the system has to process a large number of similar requests in a short time
 - Credit card processing system, airline booking system.
- ✧ Reciprocal of ROCOF is Mean time to Failure (MTTF)
 - Relevant for systems with long transactions i.e. where system processing takes a long time (e.g. CAD systems). MTTF should be longer than expected transaction length.

Availability



- ✧ Measure of the fraction of the time that the system is available for use.
- ✧ Takes repair and restart time into account
- ✧ Availability of 0.998 means software is available for 998 out of 1000 time units.
- ✧ Relevant for non-stop, continuously running systems
 - telephone switching systems, railway signalling systems.



Software Engineering

Ian Sommerville

Availability specification

Availability	Explanation
0.9	The system is available for 90% of the time. This means that, in a 24-hour period (1,440 minutes), the system will be unavailable for 144 minutes.
0.99	In a 24-hour period, the system is unavailable for 14.4 minutes.
0.999	The system is unavailable for 84 seconds in a 24-hour period.
0.9999	The system is unavailable for 8.4 seconds in a 24-hour period. Roughly, one minute per week.

Non-functional reliability requirements



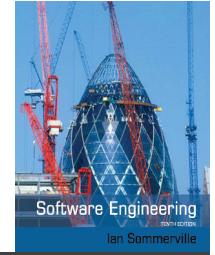
- ✧ Non-functional reliability requirements are specifications of the required reliability and availability of a system using one of the reliability metrics (POFOD, ROCOF or AVAIL).
- ✧ Quantitative reliability and availability specification has been used for many years in safety-critical systems but is uncommon for business critical systems.
- ✧ However, as more and more companies demand 24/7 service from their systems, it makes sense for them to be precise about their reliability and availability expectations.

Benefits of reliability specification



- ✧ The process of deciding the required level of the reliability helps to clarify what stakeholders really need.
- ✧ It provides a basis for assessing when to stop testing a system. You stop when the system has reached its required reliability level.
- ✧ It is a means of assessing different design strategies intended to improve the reliability of a system.
- ✧ If a regulator has to approve a system (e.g. all systems that are critical to flight safety on an aircraft are regulated), then evidence that a required reliability target has been met is important for system certification.

Specifying reliability requirements



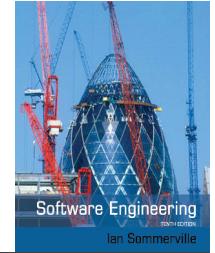
- ✧ Specify the availability and reliability requirements for different types of failure. There should be a lower probability of high-cost failures than failures that don't have serious consequences.
- ✧ Specify the availability and reliability requirements for different types of system service. Critical system services should have the highest reliability but you may be willing to tolerate more failures in less critical services.
- ✧ Think about whether a high level of reliability is really required. Other mechanisms can be used to provide reliable system service.

ATM reliability specification



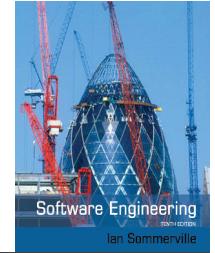
- ✧ Key concerns
 - To ensure that their ATMs carry out customer services as requested and that they properly record customer transactions in the account database.
 - To ensure that these ATM systems are available for use when required.
- ✧ Database transaction mechanisms may be used to correct transaction problems so a low-level of ATM reliability is all that is required
- ✧ Availability, in this case, is more important than reliability

ATM availability specification



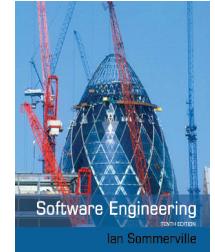
- ✧ System services
 - The customer account database service;
 - The individual services provided by an ATM such as ‘withdraw cash’, ‘provide account information’, etc.
- ✧ The database service is critical as failure of this service means that all of the ATMs in the network are out of action.
- ✧ You should specify this to have a high level of availability.
 - Database availability should be around 0.9999, between 7 am and 11pm.
 - This corresponds to a downtime of less than 1 minute per week.

ATM availability specification



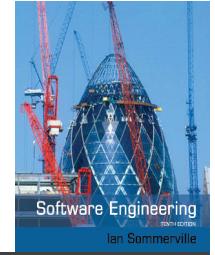
- ✧ For an individual ATM, the key reliability issues depends on mechanical reliability and the fact that it can run out of cash.
- ✧ A lower level of software availability for the ATM software is acceptable.
- ✧ The overall availability of the ATM software might therefore be specified as 0.999, which means that a machine might be unavailable for between 1 and 2 minutes each day.

Insulin pump reliability specification



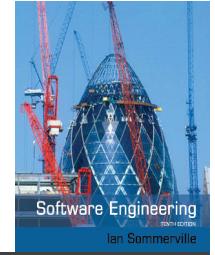
- ✧ Probability of failure (POFOD) is the most appropriate metric.
- ✧ Transient failures that can be repaired by user actions such as recalibration of the machine. A relatively low value of POFOD is acceptable (say 0.002) – one failure may occur in every 500 demands.
- ✧ Permanent failures require the software to be re-installed by the manufacturer. This should occur no more than once per year. POFOD for this situation should be less than 0.00002.

Functional reliability requirements



- ✧ Checking requirements that identify checks to ensure that incorrect data is detected before it leads to a failure.
- ✧ Recovery requirements that are geared to help the system recover after a failure has occurred.
- ✧ Redundancy requirements that specify redundant features of the system to be included.
- ✧ Process requirements for reliability which specify the development process to be used may also be included.

Examples of functional reliability requirements



RR1: A pre-defined range for all operator inputs shall be defined and the system shall check that all operator inputs fall within this pre-defined range. (Checking)

RR2: Copies of the patient database shall be maintained on two separate servers that are not housed in the same building. (Recovery, redundancy)

RR3: N-version programming shall be used to implement the braking control system. (Redundancy)

RR4: The system must be implemented in a safe subset of Ada and checked using static analysis. (Process)



Software Engineering

Ian Sommerville

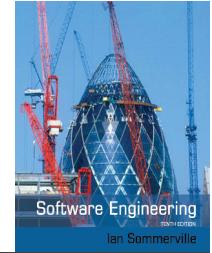
Fault-tolerant architectures

Fault tolerance



Software Engineering
Ian Sommerville

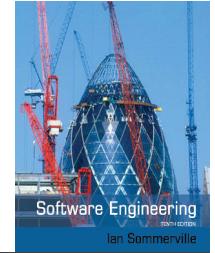
- ✧ In critical situations, software systems must be fault tolerant.
- ✧ Fault tolerance is required where there are high availability requirements or where system failure costs are very high.
- ✧ Fault tolerance means that the system can continue in operation in spite of software failure.
- ✧ Even if the system has been proved to conform to its specification, it must also be fault tolerant as there may be specification errors or the validation may be incorrect.



Fault-tolerant system architectures

- ✧ Fault-tolerant systems architectures are used in situations where fault tolerance is essential. These architectures are generally all based on redundancy and diversity.
- ✧ Examples of situations where dependable architectures are used:
 - Flight control systems, where system failure could threaten the safety of passengers
 - Reactor systems where failure of a control system could lead to a chemical or nuclear emergency
 - Telecommunication systems, where there is a need for 24/7 availability.

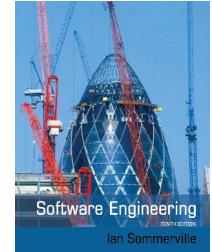
Protection systems



Software Engineering
Ian Sommerville

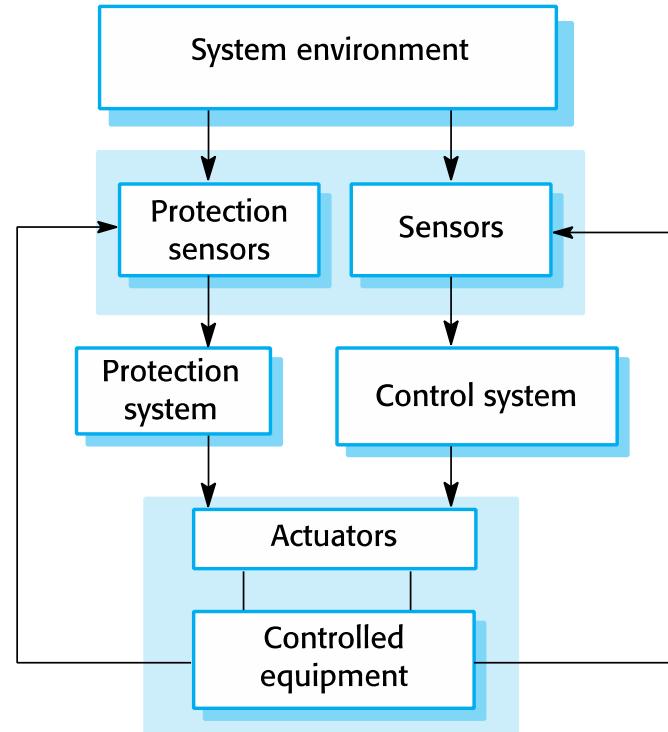
- ✧ A specialized system that is associated with some other control system, which can take emergency action if a failure occurs.
 - System to stop a train if it passes a red light
 - System to shut down a reactor if temperature/pressure are too high
- ✧ Protection systems independently monitor the controlled system and the environment.
- ✧ If a problem is detected, it issues commands to take emergency action to shut down the system and avoid a catastrophe.

Protection system architecture



Software Engineering

Ian Sommerville

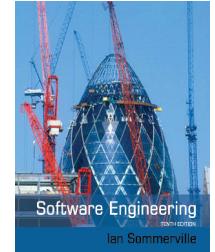


Protection system functionality



- ✧ Protection systems are redundant because they include monitoring and control capabilities that replicate those in the control software.
- ✧ Protection systems should be diverse and use different technology from the control software.
- ✧ They are simpler than the control system so more effort can be expended in validation and dependability assurance.
- ✧ Aim is to ensure that there is a low probability of failure on demand for the protection system.

Self-monitoring architectures

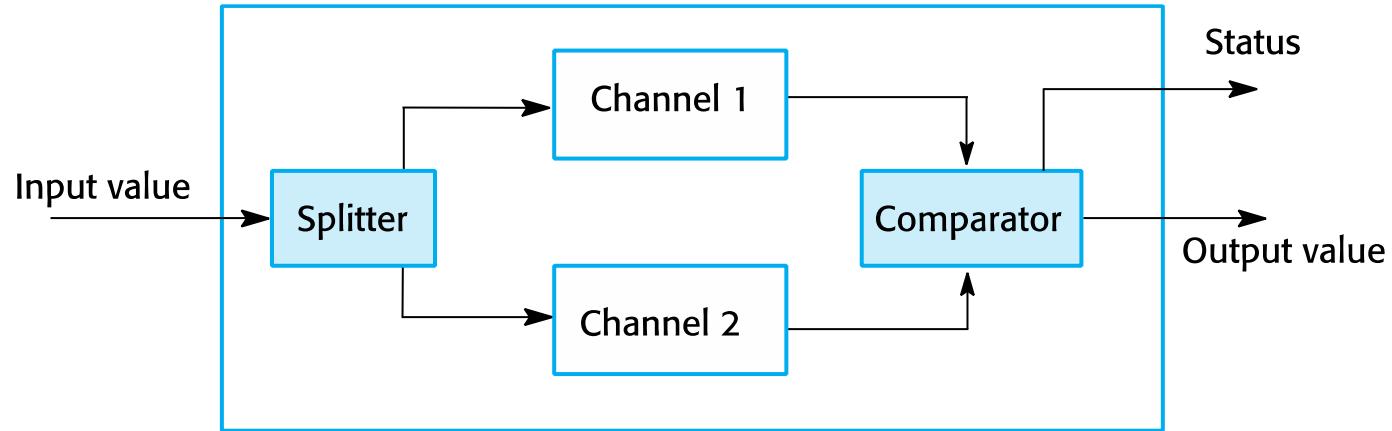


- ✧ Multi-channel architectures where the system monitors its own operations and takes action if inconsistencies are detected.
- ✧ The same computation is carried out on each channel and the results are compared. If the results are identical and are produced at the same time, then it is assumed that the system is operating correctly.
- ✧ If the results are different, then a failure is assumed and a failure exception is raised.

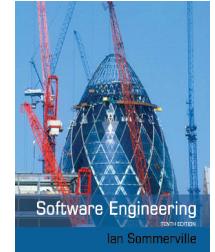
Self-monitoring architecture



Software Engineering
Ian Sommerville



Self-monitoring systems



- ✧ Hardware in each channel has to be diverse so that common mode hardware failure will not lead to each channel producing the same results.
- ✧ Software in each channel must also be diverse, otherwise the same software error would affect each channel.
- ✧ If high-availability is required, you may use several self-checking systems in parallel.
 - This is the approach used in the Airbus family of aircraft for their flight control systems.

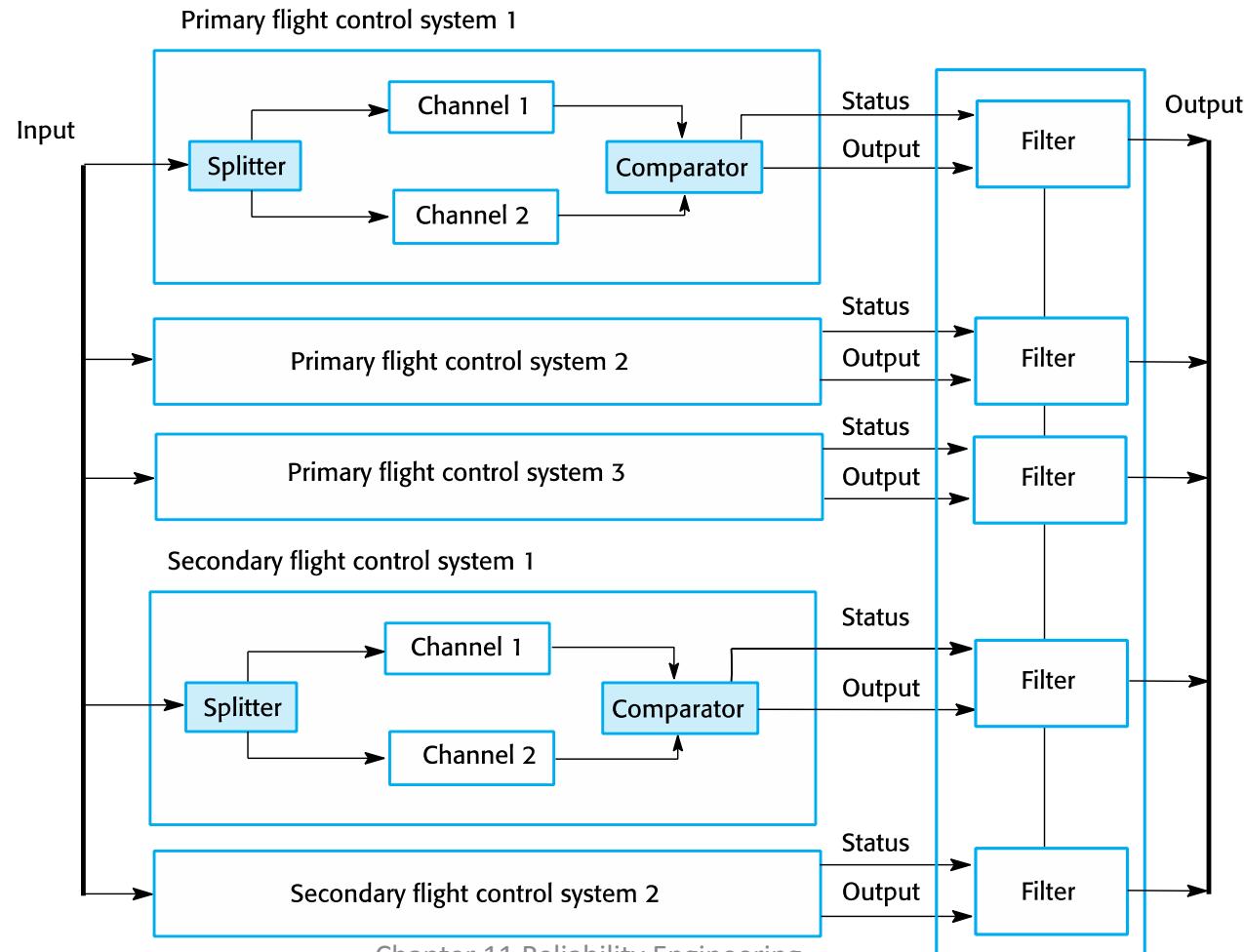
Airbus flight control system architecture



Software Engineering

Ian Sommerville

Input value



Airbus architecture discussion



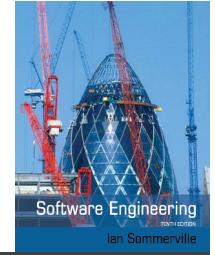
- ✧ The Airbus FCS has 5 separate computers, any one of which can run the control software.
- ✧ Extensive use has been made of diversity
 - Primary systems use a different processor from the secondary systems.
 - Primary and secondary systems use chipsets from different manufacturers.
 - Software in secondary systems is less complex than in primary system – provides only critical functionality.
 - Software in each channel is developed in different programming languages by different teams.
 - Different programming languages used in primary and secondary systems.

N-version programming



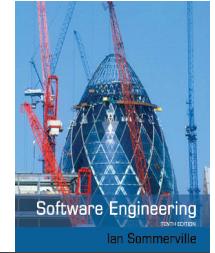
- ✧ Multiple versions of a software system carry out computations at the same time. There should be an odd number of computers involved, typically 3.
- ✧ The results are compared using a voting system and the majority result is taken to be the correct result.
- ✧ Approach derived from the notion of triple-modular redundancy, as used in hardware systems.

Hardware fault tolerance



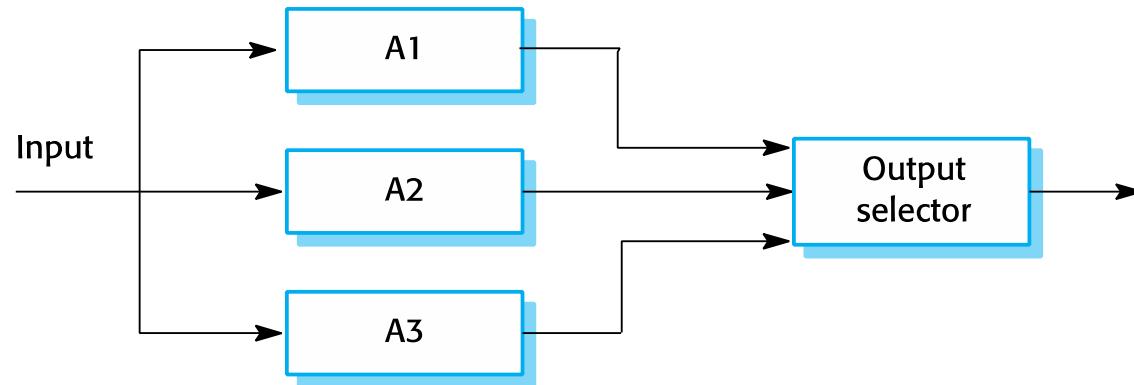
- ✧ Depends on triple-modular redundancy (TMR).
- ✧ There are three replicated identical components that receive the same input and whose outputs are compared.
- ✧ If one output is different, it is ignored and component failure is assumed.
- ✧ Based on most faults resulting from component failures rather than design faults and a low probability of simultaneous component failure.

Triple modular redundancy

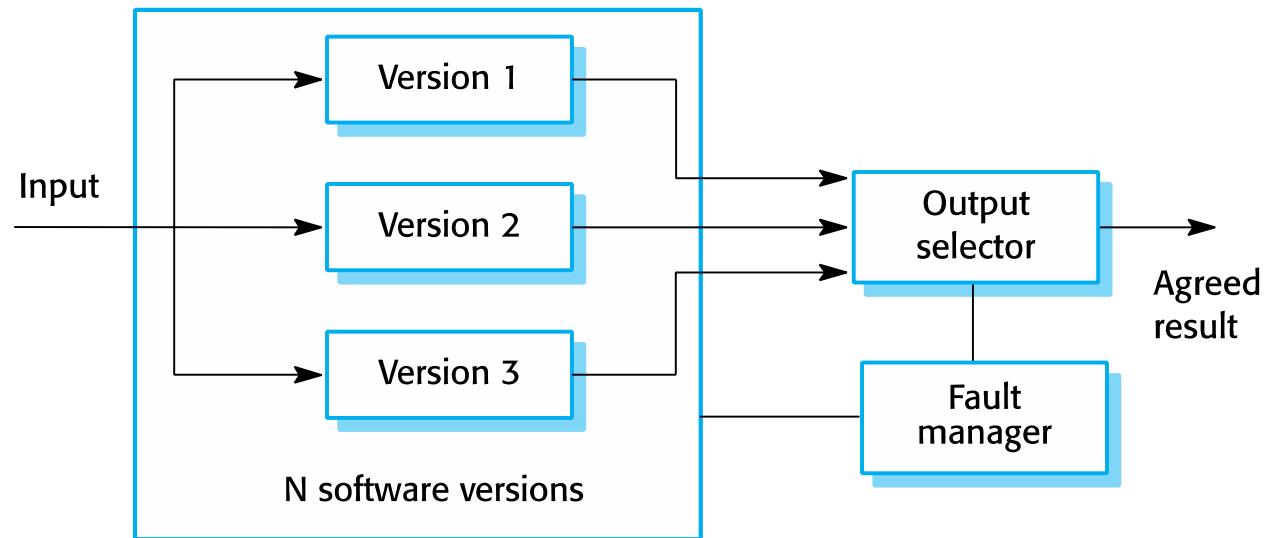


Software Engineering

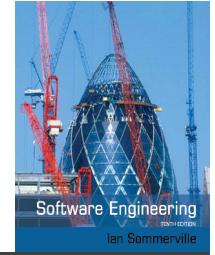
Ian Sommerville



N-version programming



N-version programming



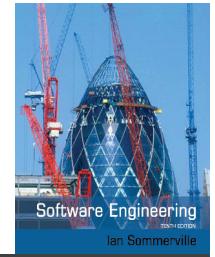
- ✧ The different system versions are designed and implemented by different teams. It is assumed that there is a low probability that they will make the same mistakes. The algorithms used should but may not be different.
- ✧ There is some empirical evidence that teams commonly misinterpret specifications in the same way and chose the same algorithms in their systems.

Software diversity



- ✧ Approaches to software fault tolerance depend on software diversity where it is assumed that different implementations of the same software specification will fail in different ways.
- ✧ It is assumed that implementations are (a) independent and (b) do not include common errors.
- ✧ Strategies to achieve diversity
 - Different programming languages
 - Different design methods and tools
 - Explicit specification of different algorithms

Problems with design diversity



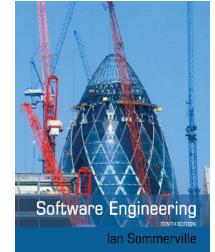
- ✧ Teams are not culturally diverse so they tend to tackle problems in the same way.
- ✧ Characteristic errors
 - Different teams make the same mistakes. Some parts of an implementation are more difficult than others so all teams tend to make mistakes in the same place;
 - Specification errors;
 - If there is an error in the specification then this is reflected in all implementations;
 - This can be addressed to some extent by using multiple specification representations.

Specification dependency



- ✧ Both approaches to software redundancy are susceptible to specification errors. If the specification is incorrect, the system could fail
- ✧ This is also a problem with hardware but software specifications are usually more complex than hardware specifications and harder to validate.
- ✧ This has been addressed in some cases by developing separate software specifications from the same user specification.

Improvements in practice



- ✧ In principle, if diversity and independence can be achieved, multi-version programming leads to very significant improvements in reliability and availability.
- ✧ In practice, observed improvements are much less significant but the approach seems leads to reliability improvements of between 5 and 9 times.
- ✧ The key question is whether or not such improvements are worth the considerable extra development costs for multi-version programming.



Programming for reliability

Dependable programming



- ✧ Good programming practices can be adopted that help reduce the incidence of program faults.
- ✧ These programming practices support
 - Fault avoidance
 - Fault detection
 - Fault tolerance

Good practice guidelines for dependable programming



Dependable programming guidelines

- 1. Limit the visibility of information in a program**
- 2. Check all inputs for validity**
- 3. Provide a handler for all exceptions**
- 4. Minimize the use of error-prone constructs**
- 5. Provide restart capabilities**
- 6. Check array bounds**
- 7. Include timeouts when calling external components**
- 8. Name all constants that represent real-world values**



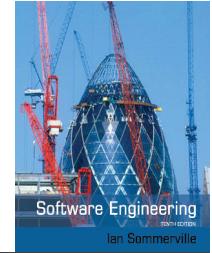
(1) Limit the visibility of information in a program

- ✧ Program components should only be allowed access to data that they need for their implementation.
- ✧ This means that accidental corruption of parts of the program state by these components is impossible.
- ✧ You can control visibility by using abstract data types where the data representation is private and you only allow access to the data through predefined operations such as get () and put ().



(2) Check all inputs for validity

- ✧ All programs take inputs from their environment and make assumptions about these inputs.
- ✧ However, program specifications rarely define what to do if an input is not consistent with these assumptions.
- ✧ Consequently, many programs behave unpredictably when presented with unusual inputs and, sometimes, these are threats to the security of the system.
- ✧ Consequently, you should always check inputs before processing against the assumptions made about these inputs.



Validity checks

Software Engineering
Ian Sommerville

Version 9

✧ Range checks

- Check that the input falls within a known range.

✧ Size checks

- Check that the input does not exceed some maximum size e.g. 40 characters for a name.

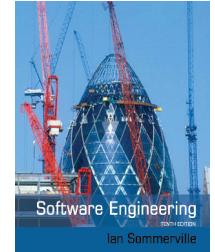
✧ Representation checks

- Check that the input does not include characters that should not be part of its representation e.g. names do not include numerals.

✧ Reasonableness checks

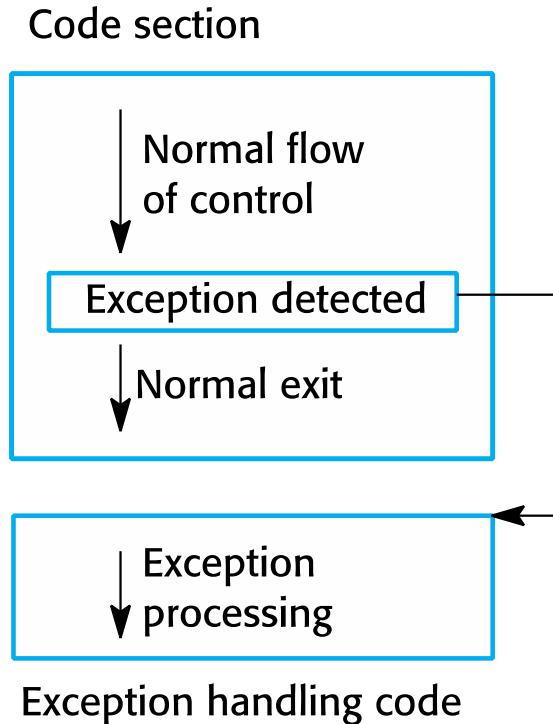
- Use information about the input to check if it is reasonable rather than an extreme value.

(3) Provide a handler for all exceptions

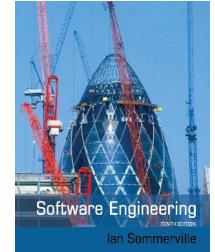


- ✧ A program exception is an error or some unexpected event such as a power failure.
- ✧ Exception handling constructs allow for such events to be handled without the need for continual status checking to detect exceptions.
- ✧ Using normal control constructs to detect exceptions needs many additional statements to be added to the program. This adds a significant overhead and is potentially error-prone.

Exception handling



Exception handling



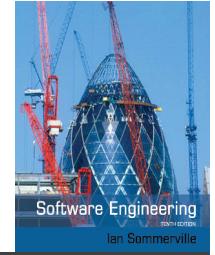
- ✧ Three possible exception handling strategies
 - Signal to a calling component that an exception has occurred and provide information about the type of exception.
 - Carry out some alternative processing to the processing where the exception occurred. This is only possible where the exception handler has enough information to recover from the problem that has arisen.
 - Pass control to a run-time support system to handle the exception.
- ✧ Exception handling is a mechanism to provide some fault tolerance

(4) Minimize the use of error-prone constructs



- ✧ Program faults are usually a consequence of human error because programmers lose track of the relationships between the different parts of the system
- ✧ This is exacerbated by error-prone constructs in programming languages that are inherently complex or that don't check for mistakes when they could do so.
- ✧ Therefore, when programming, you should try to avoid or at least minimize the use of these error-prone constructs.

Error-prone constructs



- ✧ Unconditional branch (goto) statements
- ✧ Floating-point numbers
 - Inherently imprecise. The imprecision may lead to invalid comparisons.
- ✧ Pointers
 - Pointers referring to the wrong memory areas can corrupt data. Aliasing can make programs difficult to understand and change.
- ✧ Dynamic memory allocation
 - Run-time allocation can cause memory overflow.

Error-prone constructs



✧ Parallelism

- Can result in subtle timing errors because of unforeseen interaction between parallel processes.

✧ Recursion

- Errors in recursion can cause memory overflow as the program stack fills up.

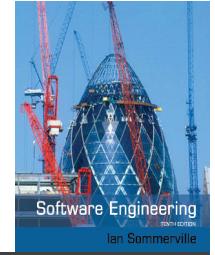
✧ Interrupts

- Interrupts can cause a critical operation to be terminated and make a program difficult to understand.

✧ Inheritance

- Code is not localised. This can result in unexpected behaviour when changes are made and problems of understanding the code.

Error-prone constructs



✧ Aliasing

- Using more than 1 name to refer to the same state variable.

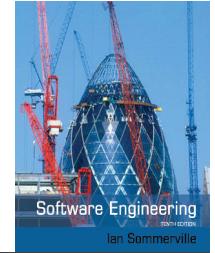
✧ Unbounded arrays

- Buffer overflow failures can occur if no bound checking on arrays.

✧ Default input processing

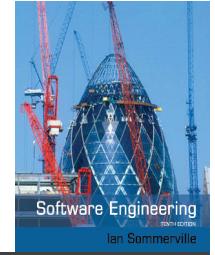
- An input action that occurs irrespective of the input.
- This can cause problems if the default action is to transfer control elsewhere in the program. In incorrect or deliberately malicious input can then trigger a program failure.

(5) Provide restart capabilities



- ✧ For systems that involve long transactions or user interactions, you should always provide a restart capability that allows the system to restart after failure without users having to redo everything that they have done.
- ✧ Restart depends on the type of system
 - Keep copies of forms so that users don't have to fill them in again if there is a problem
 - Save state periodically and restart from the saved state

(6) Check array bounds



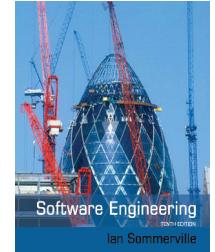
- ✧ In some programming languages, such as C, it is possible to address a memory location outside of the range allowed for in an array declaration.
- ✧ This leads to the well-known ‘bounded buffer’ vulnerability where attackers write executable code into memory by deliberately writing beyond the top element in an array.
- ✧ If your language does not include bound checking, you should therefore always check that an array access is within the bounds of the array.

(7) Include timeouts when calling external components



- ✧ In a distributed system, failure of a remote computer can be 'silent' so that programs expecting a service from that computer may never receive that service or any indication that there has been a failure.
- ✧ To avoid this, you should always include timeouts on all calls to external components.
- ✧ After a defined time period has elapsed without a response, your system should then assume failure and take whatever actions are required to recover from this.

(8) Name all constants that represent real-world values



- ✧ Always give constants that reflect real-world values (such as tax rates) names rather than using their numeric values and always refer to them by name
- ✧ You are less likely to make mistakes and type the wrong value when you are using a name rather than a value.
- ✧ This means that when these ‘constants’ change (for sure, they are not really constant), then you only have to make the change in one place in your program.

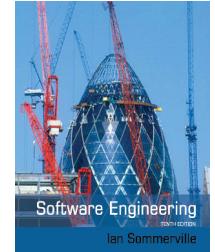


Software Engineering

Ian Sommerville

Reliability measurement

Reliability measurement



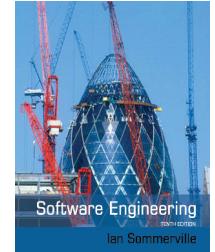
- ✧ To assess the reliability of a system, you have to collect data about its operation. The data required may include:
 - The number of system failures given a number of requests for system services. This is used to measure the POFOD. This applies irrespective of the time over which the demands are made.
 - The time or the number of transactions between system failures plus the total elapsed time or total number of transactions. This is used to measure ROCOF and MTTF.
 - The repair or restart time after a system failure that leads to loss of service. This is used in the measurement of availability. Availability does not just depend on the time between failures but also on the time required to get the system back into operation.

Reliability testing



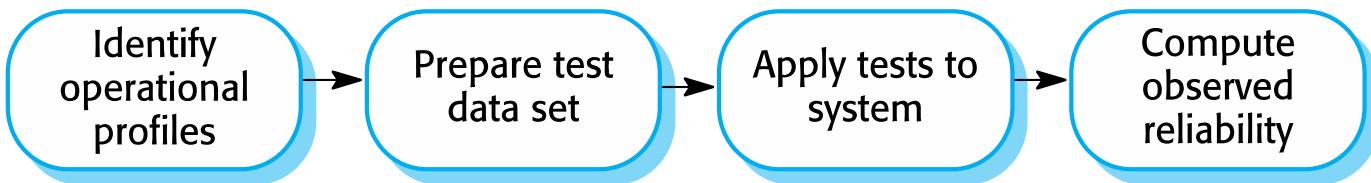
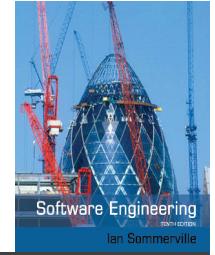
- ✧ Reliability testing (Statistical testing) involves running the program to assess whether or not it has reached the required level of reliability.
- ✧ This cannot normally be included as part of a normal defect testing process because data for defect testing is (usually) atypical of actual usage data.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

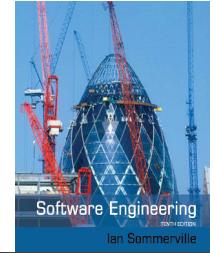
Statistical testing



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

Reliability measurement





Reliability measurement problems

Software Engineering
Ian Sommerville

11th edition

✧ Operational profile uncertainty

- The operational profile may not be an accurate reflection of the real use of the system.

✧ High costs of test data generation

- Costs can be very high if the test data for the system cannot be generated automatically.

✧ Statistical uncertainty

- You need a statistically significant number of failures to compute the reliability but highly reliable systems will rarely fail.

✧ Recognizing failure

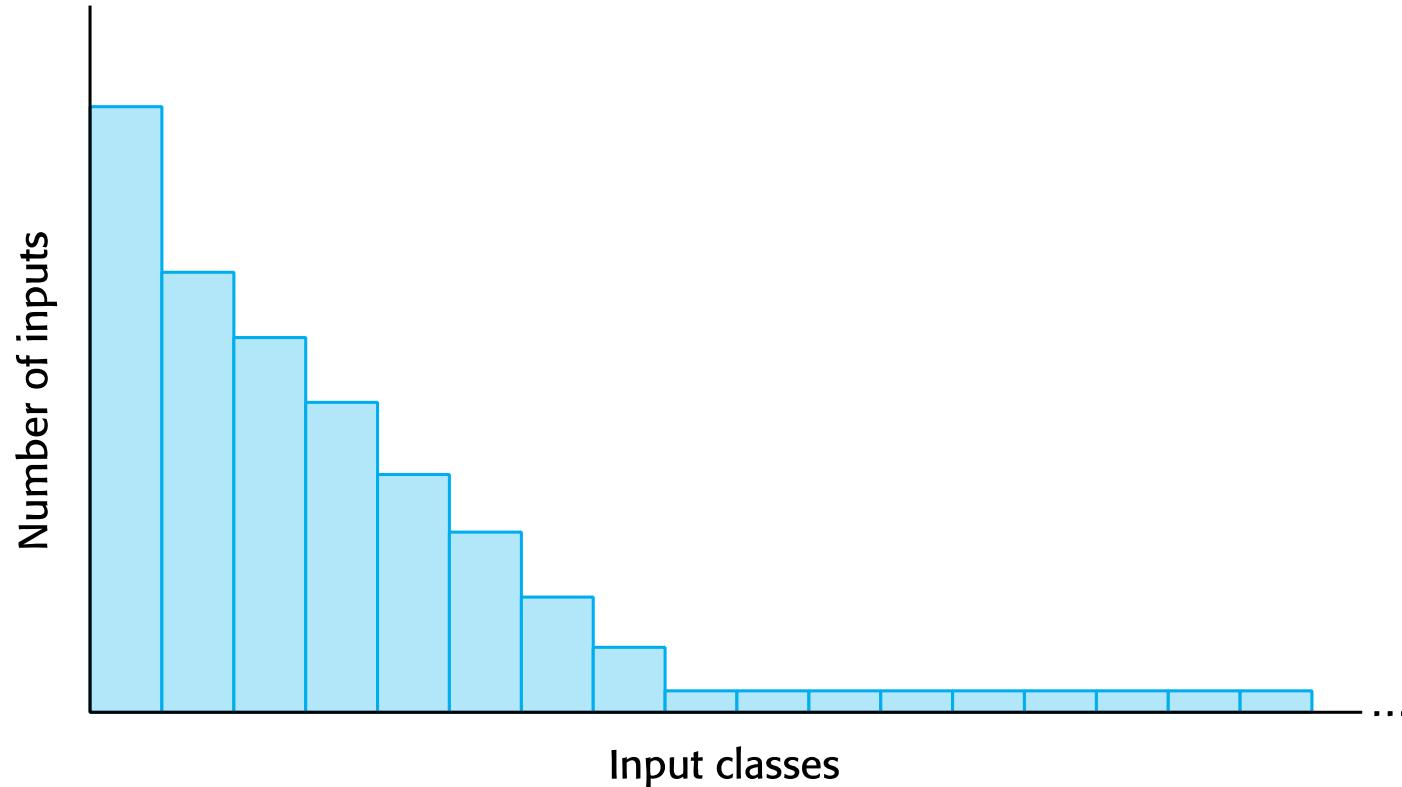
- It is not always obvious when a failure has occurred as there may be conflicting interpretations of a specification.

Operational profiles

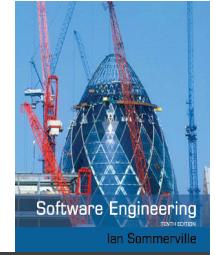


- ✧ An operational profile is a set of test data whose frequency matches the actual frequency of these inputs from 'normal' usage of the system. A close match with actual usage is necessary otherwise the measured reliability will not be reflected in the actual usage of the system.
- ✧ It can be generated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

An operational profile



Operational profile generation



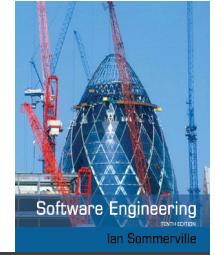
- ✧ Should be generated automatically whenever possible.
- ✧ Automatic profile generation is difficult for interactive systems.
- ✧ May be straightforward for ‘normal’ inputs but it is difficult to predict ‘unlikely’ inputs and to create test data for them.
- ✧ Pattern of usage of new systems is unknown.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

Key points



- ✧ Software reliability can be achieved by avoiding the introduction of faults, by detecting and removing faults before system deployment and by including fault tolerance facilities that allow the system to remain operational after a fault has caused a system failure.
- ✧ Reliability requirements can be defined quantitatively in the system requirements specification.
- ✧ Reliability metrics include probability of failure on demand (POFOD), rate of occurrence of failure (ROCOF) and availability (AVAIL).

Key points



- ✧ Functional reliability requirements are requirements for system functionality, such as checking and redundancy requirements, which help the system meet its non-functional reliability requirements.
- ✧ Dependable system architectures are system architectures that are designed for fault tolerance.
- ✧ There are a number of architectural styles that support fault tolerance including protection systems, self-monitoring architectures and N-version programming.



Key points

- ✧ Software diversity is difficult to achieve because it is practically impossible to ensure that each version of the software is truly independent.
- ✧ Dependable programming relies on including redundancy in a program as checks on the validity of inputs and the values of program variables.
- ✧ Statistical testing is used to estimate software reliability. It relies on testing the system with test data that matches an operational profile, which reflects the distribution of inputs to the software when it is in use.

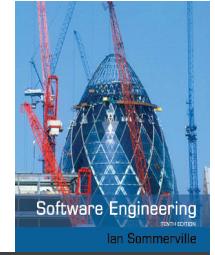


Software Engineering

Ian Sommerville

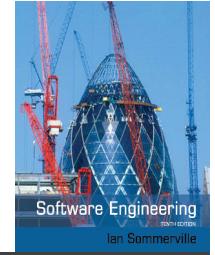
Chapter 12 – Safety Engineering

Topics covered



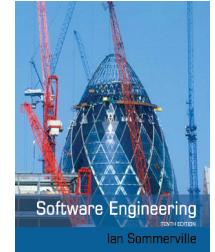
- ✧ Safety-critical systems
- ✧ Safety requirements
- ✧ Safety engineering processes
- ✧ Safety cases

Safety



- ✧ Safety is a property of a system that reflects the system's ability to operate, normally or abnormally, without danger of causing human injury or death and without damage to the system's environment.
- ✧ It is important to consider software safety as most devices whose failure is critical now incorporate software-based control systems.

Software in safety-critical systems



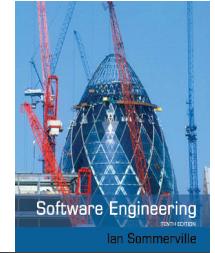
- ✧ The system may be software-controlled so that the decisions made by the software and subsequent actions are safety-critical. Therefore, the software behaviour is directly related to the overall safety of the system.
- ✧ Software is extensively used for checking and monitoring other safety-critical components in a system. For example, all aircraft engine components are monitored by software looking for early indications of component failure. This software is safety-critical because, if it fails, other components may fail and cause an accident.

Safety and reliability



Software Engineering
Ian Sommerville

- ✧ Safety and reliability are related but distinct
 - In general, reliability and availability are necessary but not sufficient conditions for system safety
- ✧ Reliability is concerned with conformance to a given specification and delivery of service
- ✧ Safety is concerned with ensuring system cannot cause damage irrespective of whether or not it conforms to its specification.
 - System reliability is essential for safety but is not enough
 - Reliable systems can be unsafe



Unsafe reliable systems

- ✧ There may be dormant faults in a system that are undetected for many years and only rarely arise.
- ✧ Specification errors
 - If the system specification is incorrect then the system can behave as specified but still cause an accident.
- ✧ Hardware failures generating spurious inputs
 - Hard to anticipate in the specification.
- ✧ Context-sensitive commands i.e. issuing the right command at the wrong time
 - Often the result of operator error.



Software Engineering

Ian Sommerville

Safety-critical systems

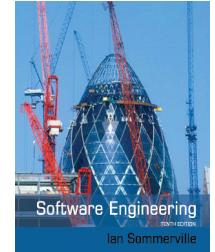
Safety critical systems



Software Engineering
Ian Sommerville

- ✧ Systems where it is essential that system operation is always safe i.e. the system should never cause damage to people or the system's environment
- ✧ Examples
 - Control and monitoring systems in aircraft
 - Process control systems in chemical manufacture
 - Automobile control systems such as braking and engine management systems

Safety criticality



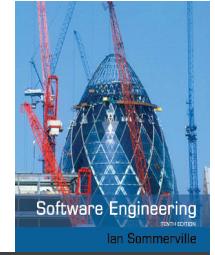
✧ Primary safety-critical systems

- Embedded software systems whose failure can cause the associated hardware to fail and directly threaten people. Example is the insulin pump control system.

✧ Secondary safety-critical systems

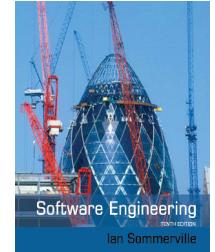
- Systems whose failure results in faults in other (socio-technical) systems, which can then have safety consequences.
 - For example, the Mentcare system is safety-critical as failure may lead to inappropriate treatment being prescribed.
 - Infrastructure control systems are also secondary safety-critical systems.

Hazards



- ✧ Situations or events that can lead to an accident
 - Stuck valve in reactor control system
 - Incorrect computation by software in navigation system
 - Failure to detect possible allergy in medication prescribing system
- ✧ Hazards do not inevitably result in accidents – accident prevention actions can be taken.

Safety achievement



Software Engineering
Ian Sommerville

✧ Hazard avoidance

- The system is designed so that some classes of hazard simply cannot arise.

✧ Hazard detection and removal

- The system is designed so that hazards are detected and removed before they result in an accident.

✧ Damage limitation

- The system includes protection features that minimise the damage that may result from an accident.

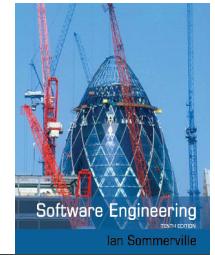
Safety terminology



Software Engineering

Ian Sommerville

Term	Definition
Accident (or mishap)	An unplanned event or sequence of events which results in human death or injury, damage to property, or to the environment. An overdose of insulin is an example of an accident.
Hazard	A condition with the potential for causing or contributing to an accident. A failure of the sensor that measures blood glucose is an example of a hazard.
Damage	A measure of the loss resulting from a mishap. Damage can range from many people being killed as a result of an accident to minor injury or property damage. Damage resulting from an overdose of insulin could be serious injury or the death of the user of the insulin pump.
Hazard severity	An assessment of the worst possible damage that could result from a particular hazard. Hazard severity can range from catastrophic, where many people are killed, to minor, where only minor damage results. When an individual death is a possibility, a reasonable assessment of hazard severity is 'very high'.
Hazard probability	The probability of the events occurring which create a hazard. Probability values tend to be arbitrary but range from 'probable' (say 1/100 chance of a hazard occurring) to 'implausible' (no conceivable situations are likely in which the hazard could occur). The probability of a sensor failure in the insulin pump that results in an overdose is probably low.
Risk	This is a measure of the probability that the system will cause an accident. The risk is assessed by considering the hazard probability, the hazard severity, and the probability that the hazard will lead to an accident. The risk of an insulin overdose is probably medium to low.



Normal accidents

- ✧ Accidents in complex systems rarely have a single cause as these systems are designed to be resilient to a single point of failure
 - Designing systems so that a single point of failure does not cause an accident is a fundamental principle of safe systems design.
- ✧ Almost all accidents are a result of combinations of malfunctions rather than single failures.
- ✧ It is probably the case that anticipating all problem combinations, especially, in software controlled systems is impossible so achieving complete safety is impossible. Accidents are inevitable.

Software safety benefits



- ✧ Although software failures can be safety-critical, the use of software control systems contributes to increased system safety
 - Software monitoring and control allows a wider range of conditions to be monitored and controlled than is possible using electro-mechanical safety systems.
 - Software control allows safety strategies to be adopted that reduce the amount of time people spend in hazardous environments.
 - Software can detect and correct safety-critical operator errors.

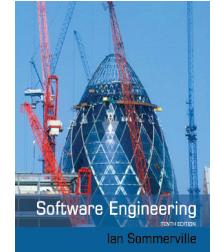


Software Engineering

Ian Sommerville

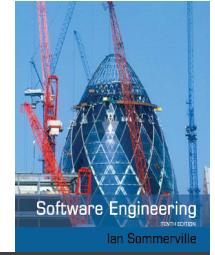
Safety requirements

Safety specification



- ✧ The goal of safety requirements engineering is to identify protection requirements that ensure that system failures do not cause injury or death or environmental damage.
- ✧ Safety requirements may be 'shall not' requirements i.e. they define situations and events that should never occur.
- ✧ Functional safety requirements define:
 - Checking and recovery features that should be included in a system
 - Features that provide protection against system failures and external attacks

Hazard-driven analysis

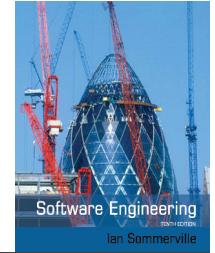


Software Engineering

Ian Sommerville

- ✧ Hazard identification
- ✧ Hazard assessment
- ✧ Hazard analysis
- ✧ Safety requirements specification

Hazard identification

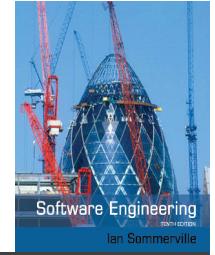


Software Engineering

Ian Sommerville

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

Insulin pump risks



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

Hazard assessment



Software Engineering
Ian Sommerville

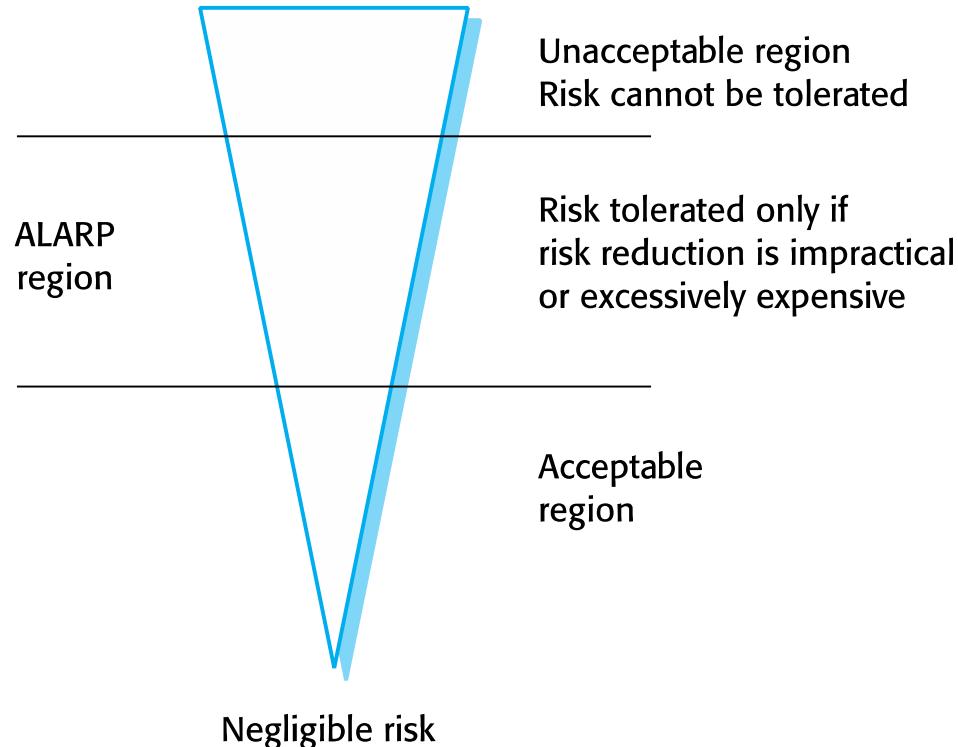
- ✧ The process is concerned with understanding the likelihood that a risk will arise and the potential consequences if an accident or incident should occur.
- ✧ Risks may be categorised as:
 - **Intolerable.** Must never arise or result in an accident
 - **As low as reasonably practical(ALARP).** Must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

The risk triangle

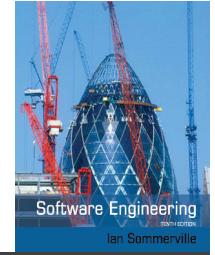


Software Engineering

Ian Sommerville

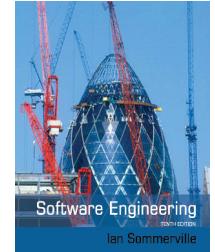


Social acceptability of risk



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

Hazard assessment



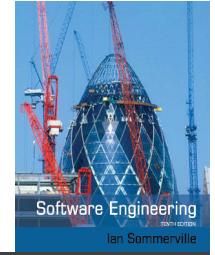
- ✧ Estimate the risk probability and the risk severity.
- ✧ It is not normally possible to do this precisely so relative values are used such as 'unlikely', 'rare', 'very high', etc.
- ✧ The aim must be to exclude risks that are likely to arise or that have high severity.

Risk classification for the insulin pump



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

Hazard analysis



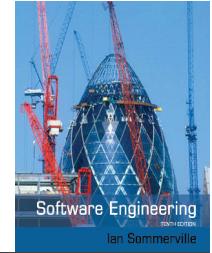
Software Engineering
Ian Sommerville

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

Fault-tree analysis



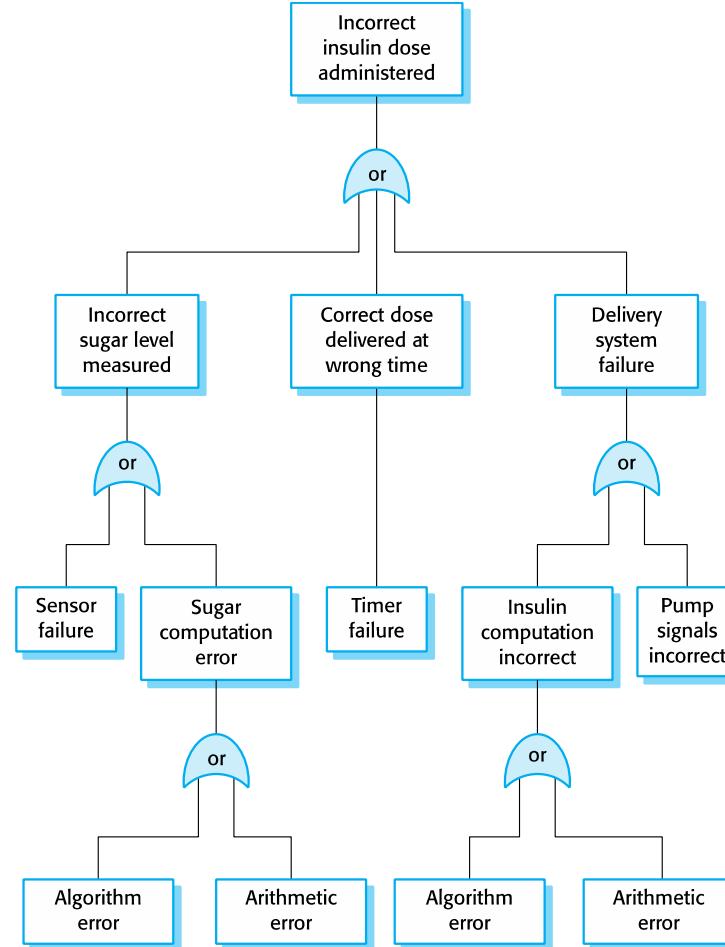
- ✧ A deductive top-down technique.
- ✧ Put the risk or hazard at the root of the tree and identify the system states that could lead to that hazard.
- ✧ Where appropriate, link these with 'and' or 'or' conditions.
- ✧ A goal should be to minimise the number of single causes of system failure.



An example of a software fault tree

Software Engineering

Ian Sommerville

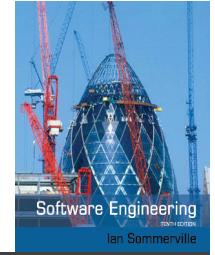


Fault tree analysis



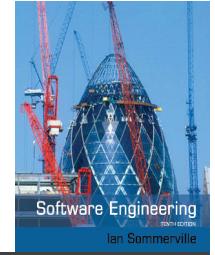
- ✧ Three possible conditions that can lead to delivery of incorrect dose of insulin
 - Incorrect measurement of blood sugar level
 - Failure of delivery system
 - Dose delivered at wrong time
- ✧ By analysis of the fault tree, root causes of these hazards related to software are:
 - Algorithm error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

Risk reduction



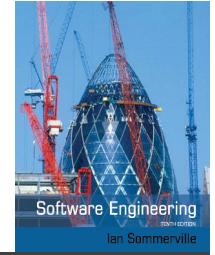
- ✧ The aim of this process is to identify dependability requirements that specify how the risks should be managed and ensure that accidents/incidents do not arise.
- ✧ Risk reduction strategies
 - Hazard avoidance;
 - Hazard detection and removal;
 - Damage limitation.

Strategy use



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

Insulin pump - software risks



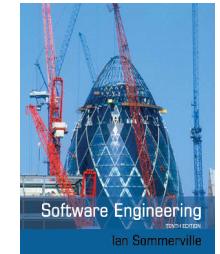
✧ Arithmetic error

- A computation causes the value of a variable to overflow or underflow;
- Maybe include an exception handler for each type of arithmetic error.

✧ Algorithmic error

- Compare dose to be delivered with previous dose or safe maximum doses. Reduce dose if too high.

Examples of safety requirements



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

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

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

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

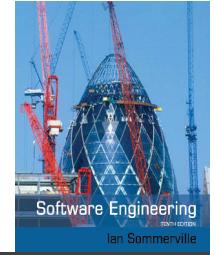
SR5: The audible alarm shall be sounded when any hardware or software anomaly is discovered and a diagnostic message, as defined in Table 4, shall be displayed.

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



Safety engineering processes

Safety engineering processes



Software Engineering
Ian Sommerville

- ✧ Safety engineering processes are based on reliability engineering processes
 - Plan-based approach with reviews and checks at each stage in the process
 - General goal of fault avoidance and fault detection
 - Must also include safety reviews and explicit identification and tracking of hazards

Regulation



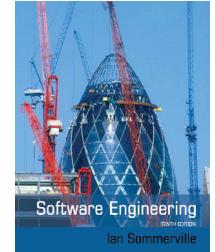
- ✧ Regulators may require evidence that safety engineering processes have been used in system development
- ✧ For example:
 - The specification of the system that has been developed and records of the checks made on that specification.
 - Evidence of the verification and validation processes that have been carried out and the results of the system verification and validation.
 - Evidence that the organizations developing the system have defined and dependable software processes that include safety assurance reviews. There must also be records that show that these processes have been properly enacted.

Agile methods and safety



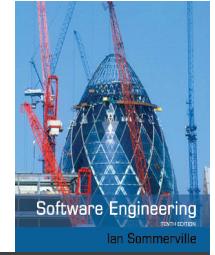
- ✧ Agile methods are not usually used for safety-critical systems engineering
 - Extensive process and product documentation is needed for system regulation. Contradicts the focus in agile methods on the software itself.
 - A detailed safety analysis of a complete system specification is important. Contradicts the interleaved development of a system specification and program.
- ✧ Some agile techniques such as test-driven development may be used

Safety assurance processes



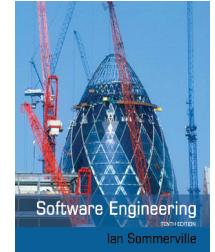
- ✧ Process assurance involves defining a dependable process and ensuring that this process is followed during the system development.
- ✧ Process assurance focuses on:
 - Do we have the right processes? Are the processes appropriate for the level of dependability required. Should include requirements management, change management, reviews and inspections, etc.
 - Are we doing the processes right? Have these processes been followed by the development team.
- ✧ Process assurance generates documentation
 - Agile processes therefore are rarely used for critical systems.

Processes for safety assurance



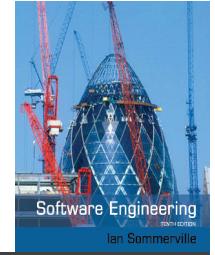
- ✧ Process assurance is important for safety-critical systems development:
 - Accidents are rare events so testing may not find all problems;
 - Safety requirements are sometimes 'shall not' requirements so cannot be demonstrated through testing.
- ✧ Safety assurance activities may be included in the software process that record the analyses that have been carried out and the people responsible for these.
 - Personal responsibility is important as system failures may lead to subsequent legal actions.

Safety related process activities



- ✧ Creation of a hazard logging and monitoring system.
- ✧ Appointment of project safety engineers who have explicit responsibility for system safety.
- ✧ Extensive use of safety reviews.
- ✧ Creation of a safety certification system where the safety of critical components is formally certified.
- ✧ Detailed configuration management (see Chapter 25).

Hazard analysis



Software Engineering

Ian Sommerville

- ✧ Hazard analysis involves identifying hazards and their root causes.
- ✧ There should be clear traceability from identified hazards through their analysis to the actions taken during the process to ensure that these hazards have been covered.
- ✧ A hazard log may be used to track hazards throughout the process.

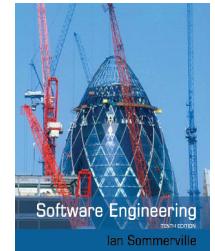
A simplified hazard log entry



Software Engineering

Ian Sommerville

Hazard Log		Page 4: Printed 20.02.2012			
<i>System:</i> Insulin Pump System		<i>File:</i> InsulinPump/Safety/HazardLog			
<i>Safety Engineer:</i> James Brown		<i>Log version:</i> 1/3			
<i>Identified Hazard</i>	Insulin overdose delivered to patient				
<i>Identified by</i>	Jane Williams				
<i>Criticality class</i>	1				
<i>Identified risk</i>	High				
<i>Fault tree identified</i>	YES	<i>Date</i>	24.01.07		<i>Location</i>
<i>Fault tree creators</i>	Jane Williams and Bill Smith				
<i>Fault tree checked</i>	YES	<i>Date</i>	28.01.07		<i>Checker</i>
					James Brown

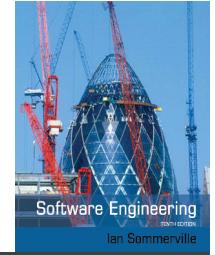


Hazard log (2)

System safety design requirements

1. The system shall include self-testing software that will test the sensor system, the clock, and the insulin delivery system.
2. The self-checking software shall be executed once per minute.
3. In the event of the self-checking software discovering a fault in any of the system components, an audible warning shall be issued and the pump display shall indicate the name of the component where the fault has been discovered. The delivery of insulin shall be suspended.
4. The system shall incorporate an override system that allows the system user to modify the computed dose of insulin that is to be delivered by the system.
5. The amount of override shall be no greater than a pre-set value (maxOverride), which is set when the system is configured by medical staff.

Safety reviews

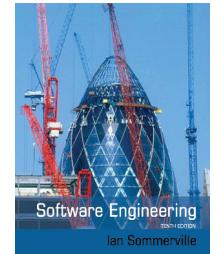


Software Engineering

Ian Sommerville

- ✧ Driven by the hazard register.
- ✧ For each identified hazard, the review team should assess the system and judge whether or not the system can cope with that hazard in a safe way.

Formal verification



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

Arguments for formal methods



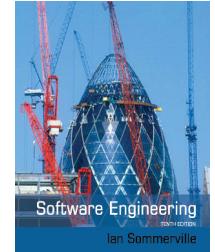
- ✧ Producing a mathematical specification requires a detailed analysis of the requirements and this is likely to uncover errors.
- ✧ Concurrent systems can be analysed to discover race conditions that might lead to deadlock. Testing for such problems is very difficult.
- ✧ They can detect implementation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

Arguments against formal methods



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

Formal methods cannot guarantee safety



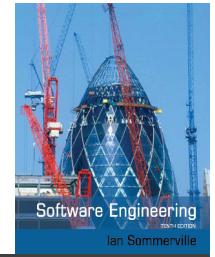
- ✧ The specification may not reflect the real requirements of system users. Users rarely understand formal notations so they cannot directly read the formal specification to find errors and omissions.
- ✧ The proof may contain errors. Program proofs are large and complex, so, like large and complex programs, they usually contain errors.
- ✧ The proof may make incorrect assumptions about the way that the system is used. If the system is not used as anticipated, then the system's behavior lies outside the scope of the proof.

Model checking



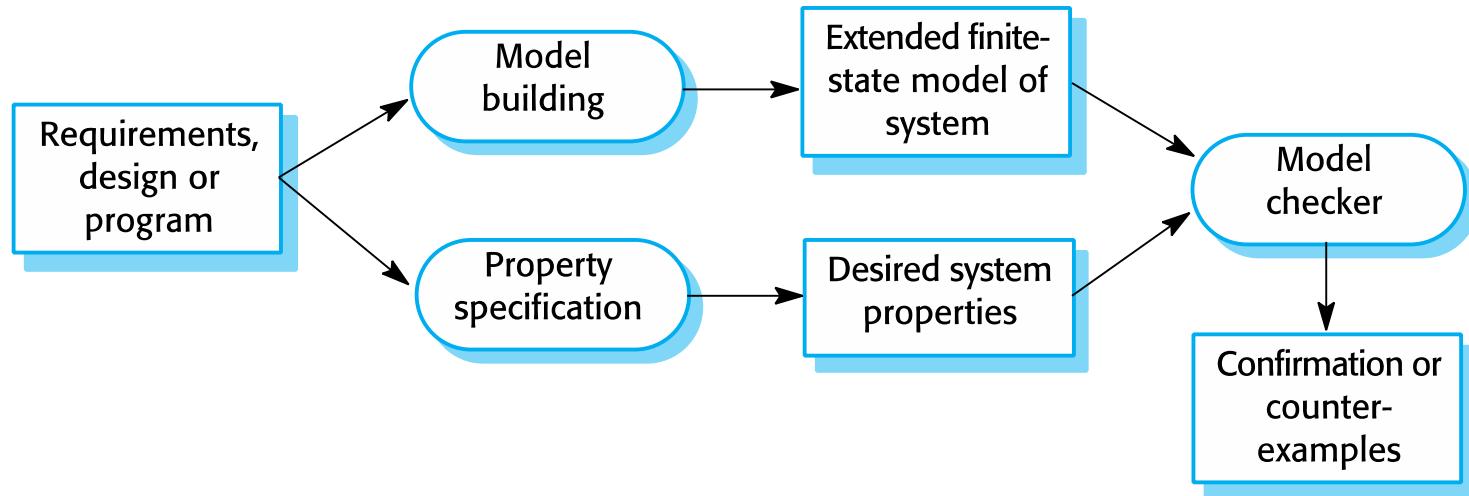
- ✧ Involves creating an extended finite state model of a system and, using a specialized system (a model checker), checking that model for errors.
- ✧ The model checker explores all possible paths through the model and checks that a user-specified property is valid for each path.
- ✧ Model checking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

Model checking

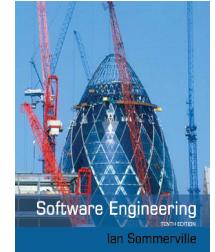


Software Engineering

Ian Sommerville



Static program analysis



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

Automated static analysis checks



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



Levels of static analysis

Software Engineering
Ian Sommerville

12.1

✧ Characteristic error checking

- The static analyzer can check for patterns in the code that are characteristic of errors made by programmers using a particular language.

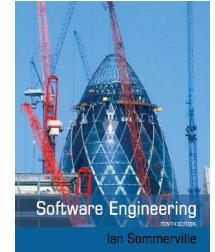
✧ User-defined error checking

- Users of a programming language define error patterns, thus extending the types of error that can be detected. This allows specific rules that apply to a program to be checked.

✧ Assertion checking

- Developers include formal assertions in their program and relationships that must hold. The static analyzer symbolically executes the code and highlights potential problems.

Use of static analysis

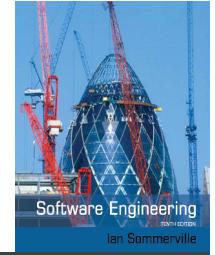


- ✧ Particularly valuable when a language such as C is used which has weak typing and hence many errors are undetected by the compiler.
- ✧ Particularly valuable for security checking – the static analyzer can discover areas of vulnerability such as buffer overflows or unchecked inputs.
- ✧ Static analysis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



Safety cases

Safety and dependability cases



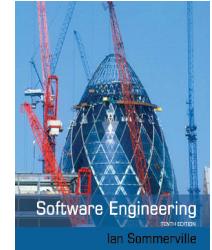
- ✧ Safety and dependability cases are structured documents that set out detailed arguments and evidence that a required level of safety or dependability has been achieved.
- ✧ They are normally required by regulators before a system can be certified for operational use. The regulator's responsibility is to check that a system is as safe or dependable as is practical.
- ✧ Regulators and developers work together and negotiate what needs to be included in a system safety/dependability case.

The system safety case

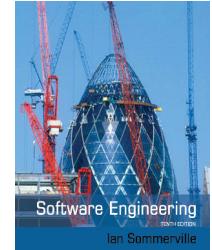


- ✧ A safety case is:
 - A documented body of evidence that provides a convincing and valid argument that a system is adequately safe for a given application in a given environment.
- ✧ Arguments in a safety case can be based on formal proof, design rationale, safety proofs, etc. Process factors may also be included.
- ✧ A software safety case is usually part of a wider system safety case that takes hardware and operational issues into account.

The contents of a software safety case



Chapter	Description
System description	An overview of the system and a description of its critical components.
Safety requirements	The safety requirements abstracted from the system requirements specification. Details of other relevant system requirements may also be included.
Hazard and risk analysis	Documents describing the hazards and risks that have been identified and the measures taken to reduce risk. Hazard analyses and hazard logs.
Design analysis	A set of structured arguments (see Section 15.5.1) that justify why the design is safe.
Verification and validation	A description of the V & V procedures used and, where appropriate, the test plans for the system. Summaries of the test results showing defects that have been detected and corrected. If formal methods have been used, a formal system specification and any analyses of that specification. Records of static analyses of the source code.



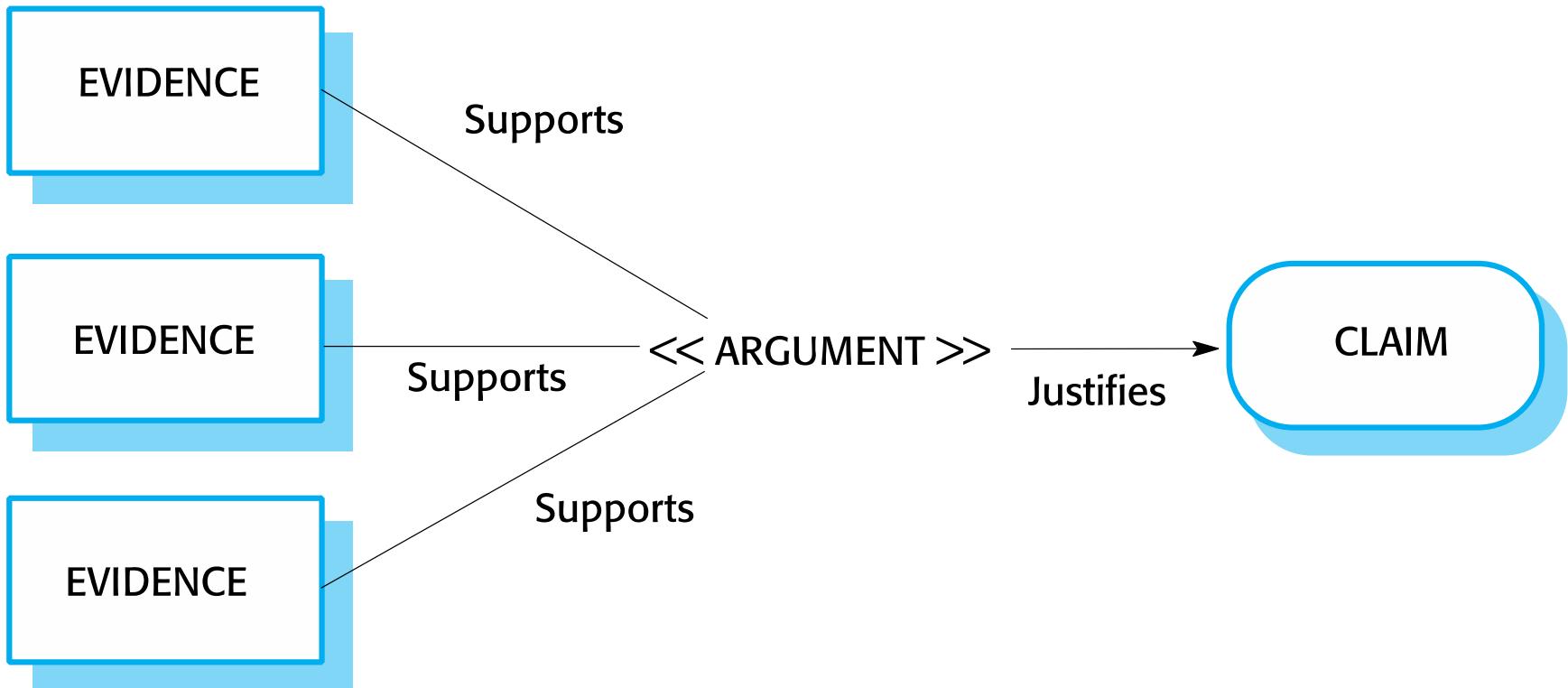
Chapter	Description
Review reports	Records of all design and safety reviews.
Team competences	Evidence of the competence of all of the team involved in safety-related systems development and validation.
Process QA	Records of the quality assurance processes (see Chapter 24) carried out during system development.
Change management processes	Records of all changes proposed, actions taken and, where appropriate, justification of the safety of these changes. Information about configuration management procedures and configuration management logs.
Associated safety cases	References to other safety cases that may impact the safety case.

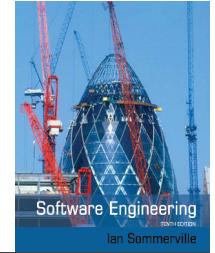
Structured arguments



- ✧ Safety cases should be based around structured arguments that present evidence to justify the assertions made in these arguments.
- ✧ The argument justifies why a claim about system safety and security is justified by the available evidence.

Structured arguments





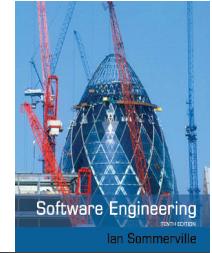
Insulin pump safety argument

- ✧ Arguments are based on claims and evidence.
- ✧ Insulin pump safety:
 - Claim: The maximum single dose of insulin to be delivered (CurrentDose) will not exceed MaxDose.
 - Evidence: Safety argument for insulin pump (discussed later)
 - Evidence: Test data for insulin pump. The value of currentDose was correctly computed in 400 tests
 - Evidence: Static analysis report for insulin pump software revealed no anomalies that affected the value of CurrentDose
 - Argument: The evidence presented demonstrates that the maximum dose of insulin that can be computed = MaxDose.

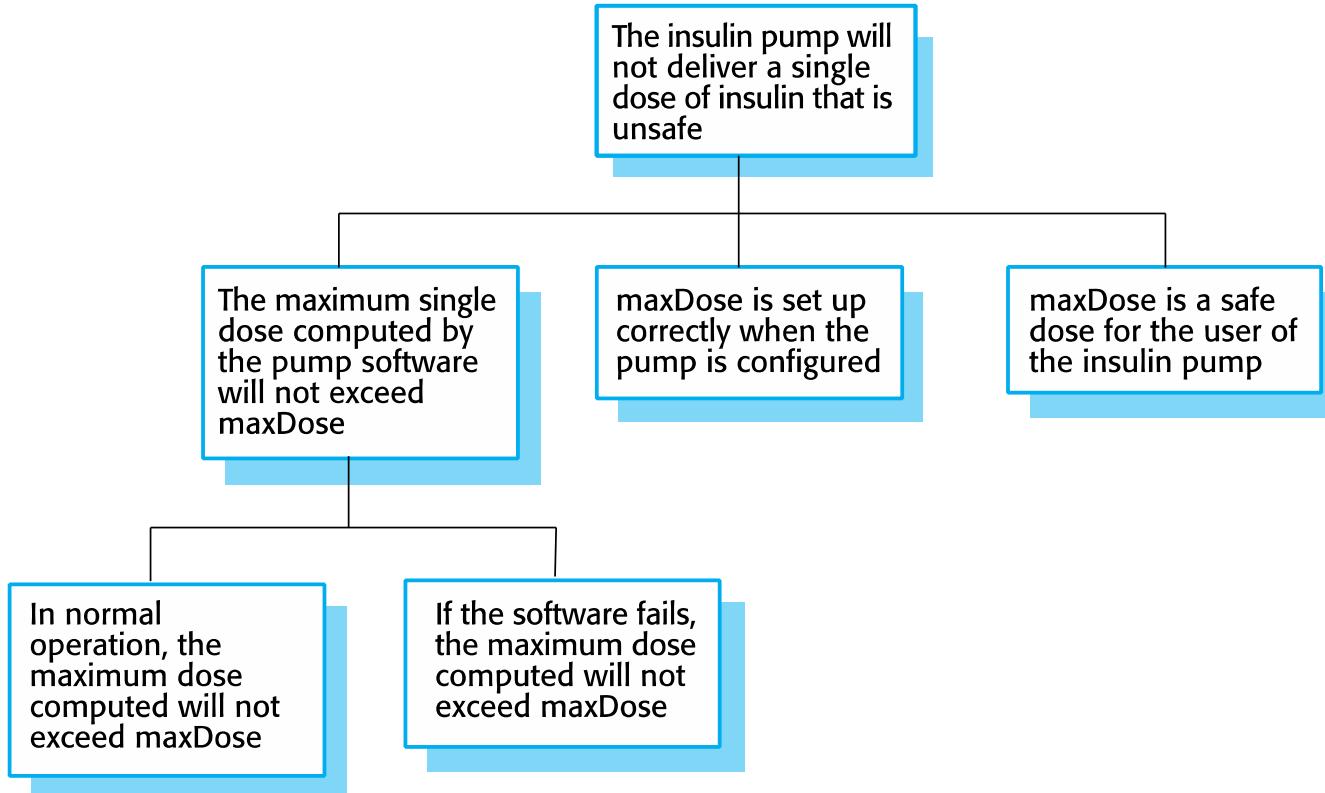
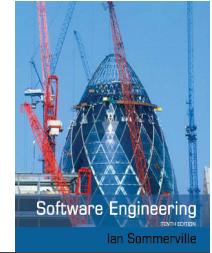
Structured safety arguments



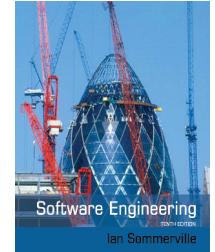
- ✧ Structured arguments that demonstrate that a system meets its safety obligations.
- ✧ It is not necessary to demonstrate that the program works as intended; the aim is simply to demonstrate safety.
- ✧ Generally based on a claim hierarchy.
 - You start at the leaves of the hierarchy and demonstrate safety. This implies the higher-level claims are true.



A safety claim hierarchy for the insulin pump

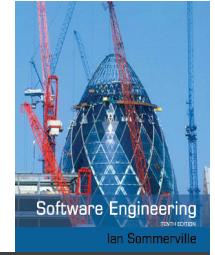


Software safety arguments



- ✧ Safety arguments are intended to show that the system cannot reach an unsafe state.
- ✧ These are weaker than correctness arguments which must show that the system code conforms to its specification.
- ✧ They are generally based on proof by contradiction
 - Assume that an unsafe state can be reached;
 - Show that this is contradicted by the program code.
- ✧ A graphical model of the safety argument may be developed.

Construction of a safety argument



- ✧ Establish the safe exit conditions for a component or a program.
- ✧ Starting from the END of the code, work backwards until you have identified all paths that lead to the exit of the code.
- ✧ Assume that the exit condition is false.
- ✧ Show that, for each path leading to the exit that the assignments made in that path contradict the assumption of an unsafe exit from the component.

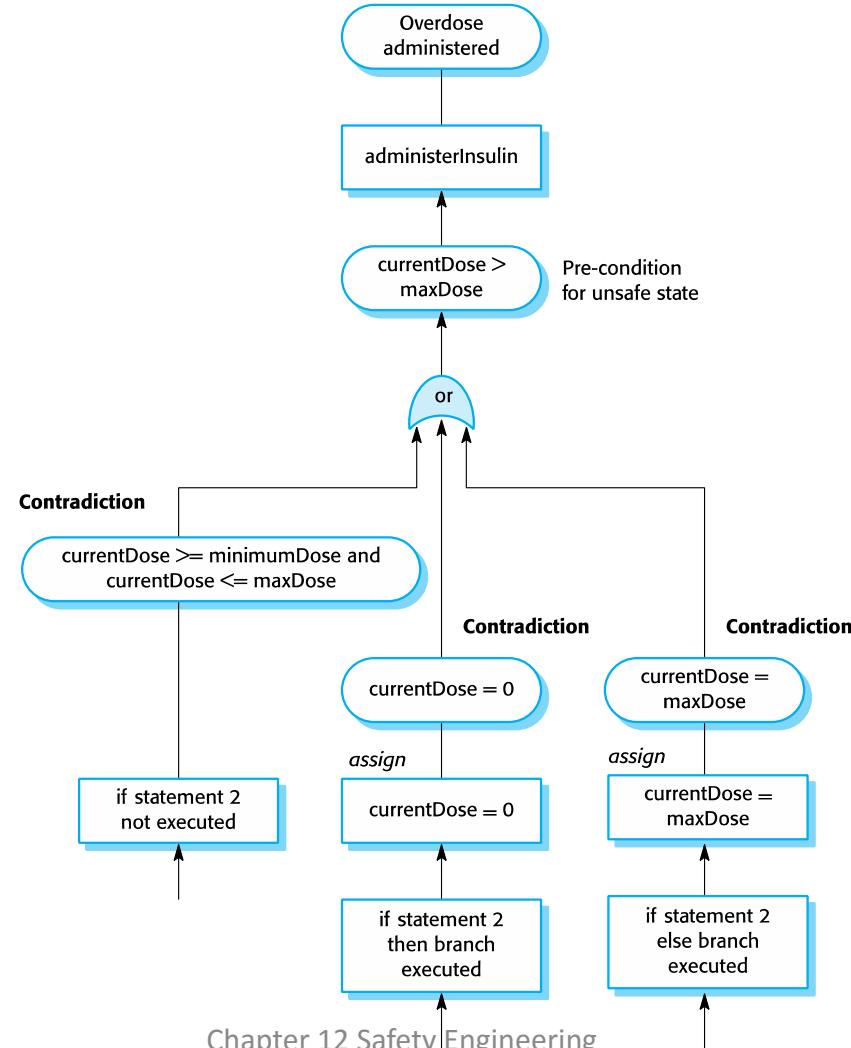
Insulin dose computation with safety checks

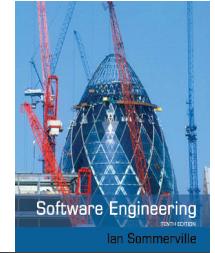


- The insulin dose to be delivered is a function of blood sugar level,
- the previous dose delivered and the time of delivery of the previous dose

```
currentDose = computeInsulin () ;  
  
// Safety check—adjust currentDose if necessary.  
// if statement 1  
if (previousDose == 0)  
{  
    if (currentDose > maxDose/2)  
        currentDose = maxDose/2 ;  
}  
else  
    if (currentDose > (previousDose * 2) )  
        currentDose = previousDose * 2 ;  
// if statement 2  
if ( currentDose < minimumDose )  
    currentDose = 0 ;  
else if ( currentDose > maxDose )  
    currentDose = maxDose ;  
administerInsulin (currentDose) ;
```

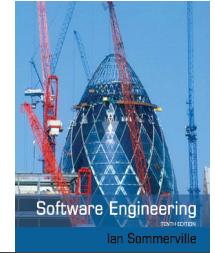
Informal safety argument based on demonstrating contradictions





Program paths

- ✧ Neither branch of if-statement 2 is executed
 - Can only happen if CurrentDose is \geq minimumDose and \leq maxDose.
- ✧ then branch of if-statement 2 is executed
 - currentDose = 0.
- ✧ else branch of if-statement 2 is executed
 - currentDose = maxDose.
- ✧ In all cases, the post conditions contradict the unsafe condition that the dose administered is greater than maxDose.



Software Engineering

Ian Sommerville

Key points

- ✧ Safety-critical systems are systems whose failure can lead to human injury or death.
- ✧ A hazard-driven approach is used to understand the safety requirements for safety-critical systems. You identify potential hazards and decompose these (using methods such as fault tree analysis) to discover their root causes. You then specify requirements to avoid or recover from these problems.
- ✧ It is important to have a well-defined, certified process for safety-critical systems development. This should include the identification and monitoring of potential hazards.

Key points



- ✧ Static analysis is an approach to V & V that examines the source code of a system, looking for errors and anomalies. It allows all parts of a program to be checked, not just those parts that are exercised by system tests.
- ✧ Model checking is a formal approach to static analysis that exhaustively checks all states in a system for potential errors.
- ✧ Safety and dependability cases collect the evidence that demonstrates a system is safe and dependable. Safety cases are required when an external regulator must certify the system before it is used.



Chapter 13 – Security Engineering

Topics covered



- ✧ Security and dependability
- ✧ Security and organizations
- ✧ Security requirements
- ✧ Secure systems design
- ✧ Security testing and assurance

Security engineering



- ✧ Tools, techniques and methods to support the development and maintenance of systems that can resist malicious attacks that are intended to damage a computer-based system or its data.
- ✧ A sub-field of the broader field of computer security.

Security dimensions



✧ *Confidentiality*

- Information in a system may be disclosed or made accessible to people or programs that are not authorized to have access to that information.

✧ *Integrity*

- Information in a system may be damaged or corrupted making it unusual or unreliable.

✧ *Availability*

- Access to a system or its data that is normally available may not be possible.

Security levels



- ✧ Infrastructure security, which is concerned with maintaining the security of all systems and networks that provide an infrastructure and a set of shared services to the organization.
- ✧ Application security, which is concerned with the security of individual application systems or related groups of systems.
- ✧ Operational security, which is concerned with the secure operation and use of the organization's systems.

System layers where security may be compromised



Application

Reusable components and libraries

Middleware

Database management

Generic, shared applications (browsers, e--mail, etc)

Operating System

Network

Computer hardware

Application/infrastructure security



- ✧ Application security is a software engineering problem where the system is designed to resist attacks.
- ✧ Infrastructure security is a systems management problem where the infrastructure is configured to resist attacks.
- ✧ The focus of this chapter is application security rather than infrastructure security.

System security management



- ✧ User and permission management
 - Adding and removing users from the system and setting up appropriate permissions for users
- ✧ Software deployment and maintenance
 - Installing application software and middleware and configuring these systems so that vulnerabilities are avoided.
- ✧ Attack monitoring, detection and recovery
 - Monitoring the system for unauthorized access, design strategies for resisting attacks and develop backup and recovery strategies.

Operational security



- ✧ Primarily a human and social issue
- ✧ Concerned with ensuring the people do not take actions that may compromise system security
 - E.g. Tell others passwords, leave computers logged on
- ✧ Users sometimes take insecure actions to make it easier for them to do their jobs
- ✧ There is therefore a trade-off between system security and system effectiveness.



Software Engineering

Ian Sommerville

Security and dependability

Security



- ✧ The security of a system is a system property that reflects the system's ability to protect itself from accidental or deliberate external attack.
- ✧ Security is essential as most systems are networked so that external access to the system through the Internet is possible.
- ✧ Security is an essential pre-requisite for availability, reliability and safety.

Fundamental security



- ✧ If a system is a networked system and is insecure then statements about its reliability and its safety are unreliable.
- ✧ These statements depend on the executing system and the developed system being the same. However, intrusion can change the executing system and/or its data.
- ✧ Therefore, the reliability and safety assurance is no longer valid.

Security terminology



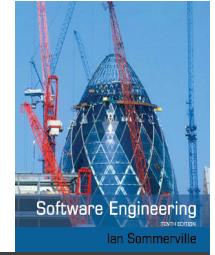
Term	Definition
Asset	Something of value which has to be protected. The asset may be the software system itself or data used by that system.
Attack	An exploitation of a system's vulnerability. Generally, this is from outside the system and is a deliberate attempt to cause some damage.
Control	A protective measure that reduces a system's vulnerability. Encryption is an example of a control that reduces a vulnerability of a weak access control system
Exposure	Possible loss or harm to a computing system. This can be loss or damage to data, or can be a loss of time and effort if recovery is necessary after a security breach.
Threat	Circumstances that have potential to cause loss or harm. You can think of these as a system vulnerability that is subjected to an attack.
Vulnerability	A weakness in a computer-based system that may be exploited to cause loss or harm.

Examples of security terminology (Mentcare)



Term	Example
Asset	The records of each patient that is receiving or has received treatment.
Exposure	Potential financial loss from future patients who do not seek treatment because they do not trust the clinic to maintain their data. Financial loss from legal action by the sports star. Loss of reputation.
Vulnerability	A weak password system which makes it easy for users to set guessable passwords. User ids that are the same as names.
Attack	An impersonation of an authorized user.
Threat	An unauthorized user will gain access to the system by guessing the credentials (login name and password) of an authorized user.
Control	A password checking system that disallows user passwords that are proper names or words that are normally included in a dictionary.

Threat types



- ✧ Interception threats that allow an attacker to gain access to an asset.
 - A possible threat to the Mentcare system might be a situation where an attacker gains access to the records of an individual patient.
- ✧ Interruption threats that allow an attacker to make part of the system unavailable.
 - A possible threat might be a denial of service attack on a system database server so that database connections become impossible.



Threat types

- ✧ Modification threats that allow an attacker to tamper with a system asset.
 - In the Mentcare system, a modification threat would be where an attacker alters or destroys a patient record.
- ✧ Fabrication threats that allow an attacker to insert false information into a system.
 - This is perhaps not a credible threat in the Mentcare system but would be a threat in a banking system, where false transactions might be added to the system that transfer money to the perpetrator's bank account.

Security assurance



✧ Vulnerability avoidance

- The system is designed so that vulnerabilities do not occur. For example, if there is no external network connection then external attack is impossible

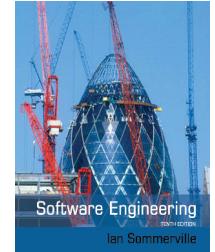
✧ Attack detection and elimination

- The system is designed so that attacks on vulnerabilities are detected and neutralised before they result in an exposure. For example, virus checkers find and remove viruses before they infect a system

✧ Exposure limitation and recovery

- The system is designed so that the adverse consequences of a successful attack are minimised. For example, a backup policy allows damaged information to be restored

Security and dependability



✧ *Security and reliability*

- If a system is attacked and the system or its data are corrupted as a consequence of that attack, then this may induce system failures that compromise the reliability of the system.

✧ *Security and availability*

- A common attack on a web-based system is a denial of service attack, where a web server is flooded with service requests from a range of different sources. The aim of this attack is to make the system unavailable.

Security and dependability



✧ *Security and safety*

- An attack that corrupts the system or its data means that assumptions about safety may not hold. Safety checks rely on analysing the source code of safety critical software and assume the executing code is a completely accurate translation of that source code. If this is not the case, safety-related failures may be induced and the safety case made for the software is invalid.

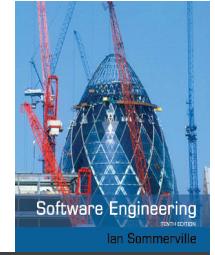
✧ *Security and resilience*

- Resilience is a system characteristic that reflects its ability to resist and recover from damaging events. The most probable damaging event on networked software systems is a cyberattack of some kind so most of the work now done in resilience is aimed at deterring, detecting and recovering from such attacks.



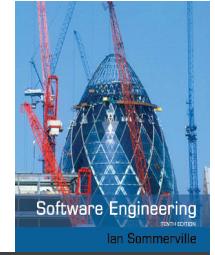
Security and organizations

Security is a business issue



- ✧ Security is expensive and it is important that security decisions are made in a cost-effective way
 - There is no point in spending more than the value of an asset to keep that asset secure.
- ✧ Organizations use a risk-based approach to support security decision making and should have a defined security policy based on security risk analysis
- ✧ Security risk analysis is a business rather than a technical process

Organizational security policies



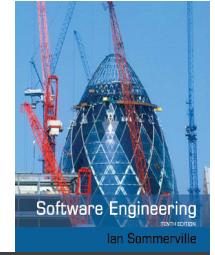
- ✧ Security policies should set out general information access strategies that should apply across the organization.
- ✧ The point of security policies is to inform everyone in an organization about security so these should not be long and detailed technical documents.
- ✧ From a security engineering perspective, the security policy defines, in broad terms, the security goals of the organization.
- ✧ The security engineering process is concerned with implementing these goals.

Security policies



- ✧ *The assets that must be protected*
 - It is not cost-effective to apply stringent security procedures to all organizational assets. Many assets are not confidential and can be made freely available.
- ✧ *The level of protection that is required for different types of asset*
 - For sensitive personal information, a high level of security is required; for other information, the consequences of loss may be minor so a lower level of security is adequate.

Security policies



- ✧ *The responsibilities of individual users, managers and the organization*
 - The security policy should set out what is expected of users e.g. strong passwords, log out of computers, office security, etc.
- ✧ *Existing security procedures and technologies that should be maintained*
 - For reasons of practicality and cost, it may be essential to continue to use existing approaches to security even where these have known limitations.

Security risk assessment and management



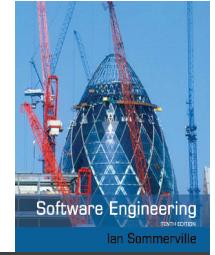
- ✧ Risk assessment and management is concerned with assessing the possible losses that might ensue from attacks on the system and balancing these losses against the costs of security procedures that may reduce these losses.
- ✧ Risk management should be driven by an organisational security policy.
- ✧ Risk management involves
 - Preliminary risk assessment
 - Life cycle risk assessment
 - Operational risk assessment

Preliminary risk assessment



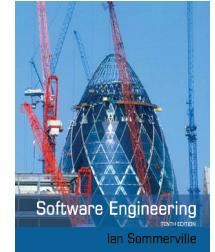
- ✧ The aim of this initial risk assessment is to identify generic risks that are applicable to the system and to decide if an adequate level of security can be achieved at a reasonable cost.
- ✧ The risk assessment should focus on the identification and analysis of high-level risks to the system.
- ✧ The outcomes of the risk assessment process are used to help identify security requirements.

Design risk assessment



- ✧ This risk assessment takes place during the system development life cycle and is informed by the technical system design and implementation decisions.
- ✧ The results of the assessment may lead to changes to the security requirements and the addition of new requirements.
- ✧ Known and potential vulnerabilities are identified, and this knowledge is used to inform decision making about the system functionality and how it is to be implemented, tested, and deployed.

Operational risk assessment



- ✧ This risk assessment process focuses on the use of the system and the possible risks that can arise from human behavior.
- ✧ Operational risk assessment should continue after a system has been installed to take account of how the system is used.
- ✧ Organizational changes may mean that the system is used in different ways from those originally planned. These changes lead to new security requirements that have to be implemented as the system evolves.



Security requirements



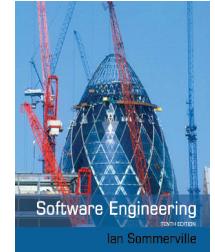
Security specification

Software Engineering

Ian Sommerville

- ✧ Security specification has something in common with safety requirements specification – in both cases, your concern is to avoid something bad happening.
- ✧ Four major differences
 - Safety problems are accidental – the software is not operating in a hostile environment. In security, you must assume that attackers have knowledge of system weaknesses
 - When safety failures occur, you can look for the root cause or weakness that led to the failure. When failure results from a deliberate attack, the attacker may conceal the cause of the failure.
 - Shutting down a system can avoid a safety-related failure. Causing a shut down may be the aim of an attack.
 - Safety-related events are not generated from an intelligent adversary. An attacker can probe defenses over time to discover weaknesses.

Types of security requirement



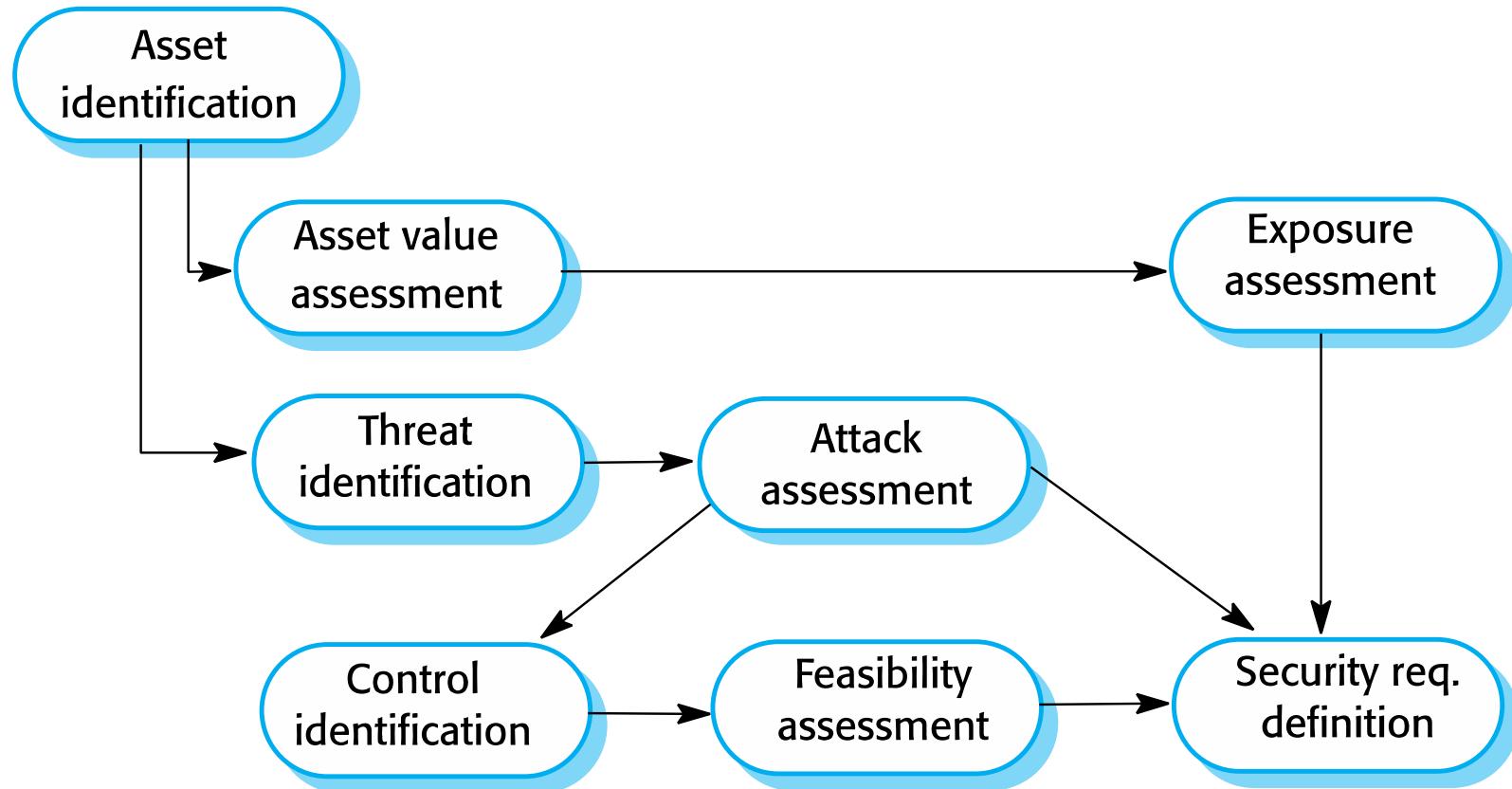
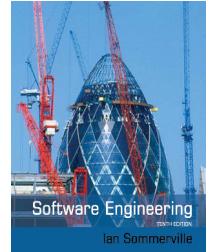
- ✧ Identification requirements.
- ✧ Authentication requirements.
- ✧ Authorisation requirements.
- ✧ Immunity requirements.
- ✧ Integrity requirements.
- ✧ Intrusion detection requirements.
- ✧ Non-repudiation requirements.
- ✧ Privacy requirements.
- ✧ Security auditing requirements.
- ✧ System maintenance security requirements.

Security requirement classification



- ✧ Risk avoidance requirements set out the risks that should be avoided by designing the system so that these risks simply cannot arise.
- ✧ Risk detection requirements define mechanisms that identify the risk if it arises and neutralise the risk before losses occur.
- ✧ Risk mitigation requirements set out how the system should be designed so that it can recover from and restore system assets after some loss has occurred.

The preliminary risk assessment process for security requirements



Security risk assessment



✧ Asset identification

- Identify the key system assets (or services) that have to be protected.

✧ Asset value assessment

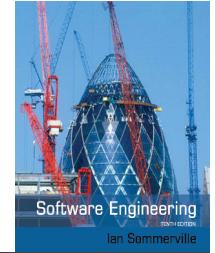
- Estimate the value of the identified assets.

✧ Exposure assessment

- Assess the potential losses associated with each asset.

✧ Threat identification

- Identify the most probable threats to the system assets



Security risk assessment

✧ Attack assessment

- Decompose threats into possible attacks on the system and the ways that these may occur.

✧ Control identification

- Propose the controls that may be put in place to protect an asset.

✧ Feasibility assessment

- Assess the technical feasibility and cost of the controls.

✧ Security requirements definition

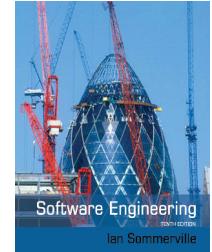
- Define system security requirements. These can be infrastructure or application system requirements.

Asset analysis in a preliminary risk assessment report for the Mentcare system



Asset	Value	Exposure
The information system	High. Required to support all clinical consultations. Potentially safety-critical.	High. Financial loss as clinics may have to be canceled. Costs of restoring system. Possible patient harm if treatment cannot be prescribed.
The patient database	High. Required to support all clinical consultations. Potentially safety-critical.	High. Financial loss as clinics may have to be canceled. Costs of restoring system. Possible patient harm if treatment cannot be prescribed.
An individual patient record	Normally low although may be high for specific high-profile patients.	Low direct losses but possible loss of reputation.

Threat and control analysis in a preliminary risk assessment report



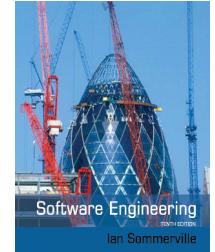
Threat	Probability	Control	Feasibility
An unauthorized user gains access as system manager and makes system unavailable	Low	Only allow system management from specific locations that are physically secure.	Low cost of implementation but care must be taken with key distribution and to ensure that keys are available in the event of an emergency.
An unauthorized user gains access as system user and accesses confidential information	High	Require all users to authenticate themselves using a biometric mechanism. Log all changes to patient information to track system usage.	Technically feasible but high-cost solution. Possible user resistance. Simple and transparent to implement and also supports recovery.

Security requirements for the Mentcare system



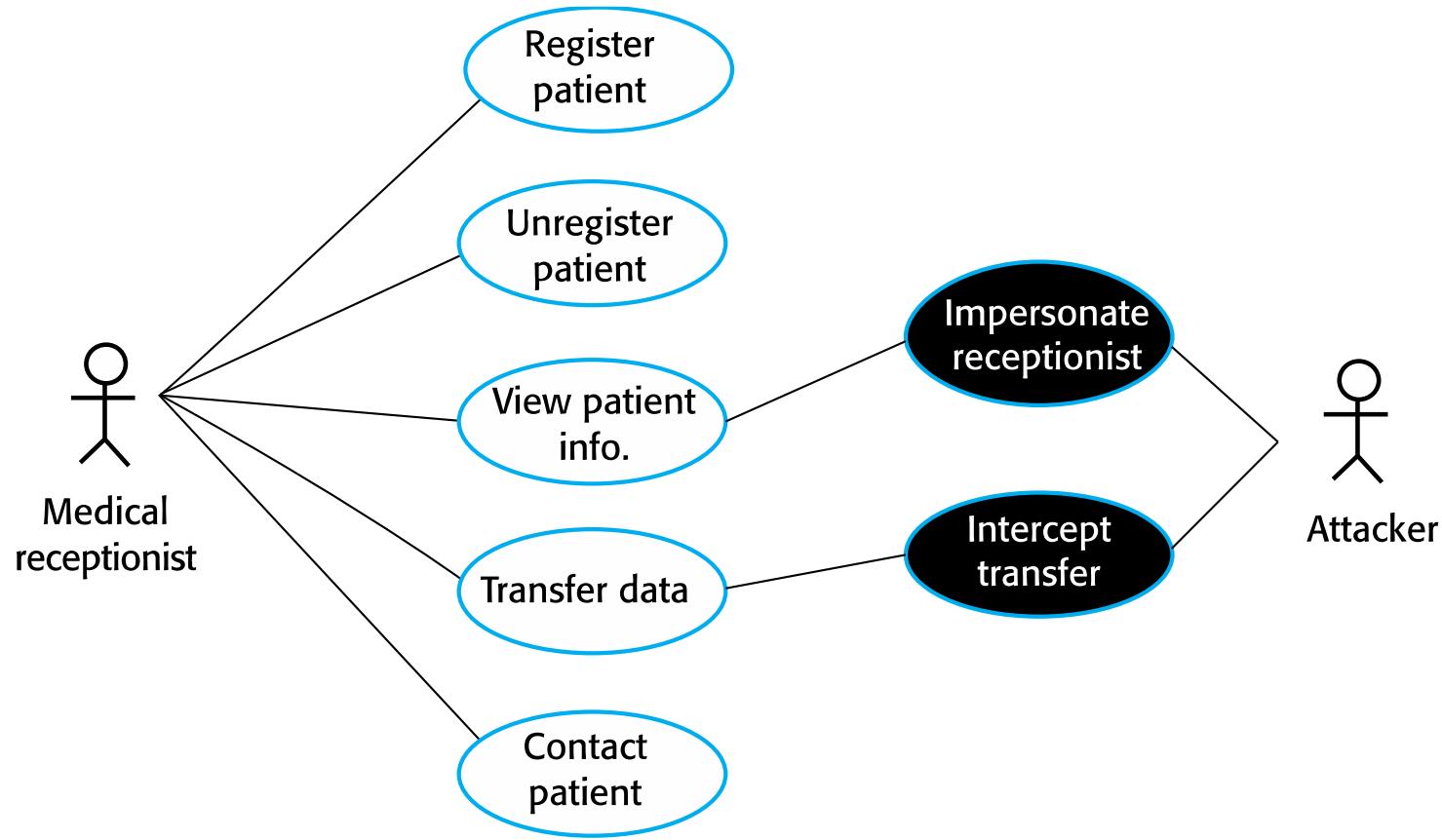
- ✧ Patient information shall be downloaded at the start of a clinic session to a secure area on the system client that is used by clinical staff.
- ✧ All patient information on the system client shall be encrypted.
- ✧ Patient information shall be uploaded to the database after a clinic session has finished and deleted from the client computer.
- ✧ A log on a separate computer from the database server must be maintained of all changes made to the system database.

Misuse cases

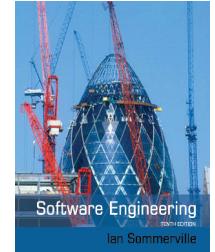


- ✧ Misuse cases are instances of threats to a system
- ✧ Interception threats
 - Attacker gains access to an asset
- ✧ Interruption threats
 - Attacker makes part of a system unavailable
- ✧ Modification threats
 - A system asset is tampered with
- ✧ Fabrication threats
 - False information is added to a system

Misuse cases



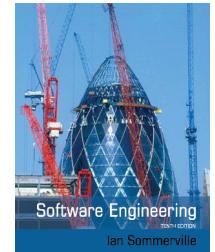
Mentcare use case – Transfer data



Mentcare system: Transfer data

Actors	Medical receptionist, Patient records system (PRS)
Description	A receptionist may transfer data from the Mentcare system 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.

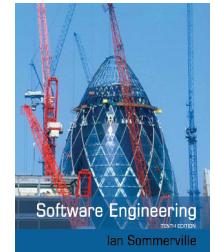
Mentcare misuse case: Intercept transfer



Mentcare system: Intercept transfer (Misuse case)

Actors	Medical receptionist, Patient records system (PRS), Attacker
Description	A receptionist transfers data from his or her PC to the Mentcare system on the server. An attacker intercepts the data transfer and takes a copy of that data.
Data (assets)	Patient's personal information, treatment summary
Attacks	A network monitor is added to the system and packets from the receptionist to the server are intercepted. A spoof server is set up between the receptionist and the database server so that receptionist believes they are interacting with the real system.

Misuse case: Intercept transfer



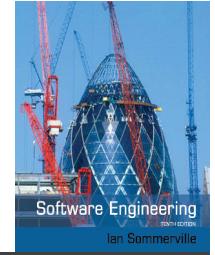
Mentcare system: Intercept transfer (Misuse case)

Mitigations	<p>All networking equipment must be maintained in a locked room. Engineers accessing the equipment must be accredited.</p> <p>All data transfers between the client and server must be encrypted.</p> <p>Certificate-based client-server communication must be used</p>
Requirements	<p>All communications between the client and the server must use the Secure Socket Layer (SSL). The https protocol uses certificate based authentication and encryption.</p>



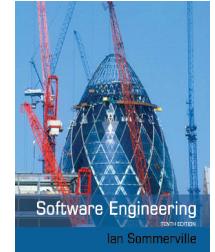
Secure systems design

Secure systems design



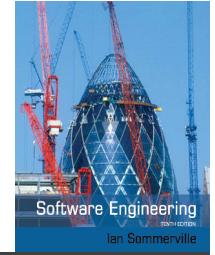
- ✧ Security should be designed into a system – it is very difficult to make an insecure system secure after it has been designed or implemented
- ✧ Architectural design
 - how do architectural design decisions affect the security of a system?
- ✧ Good practice
 - what is accepted good practice when designing secure systems?

Design compromises



- ✧ Adding security features to a system to enhance its security affects other attributes of the system
- ✧ Performance
 - Additional security checks slow down a system so its response time or throughput may be affected
- ✧ Usability
 - Security measures may require users to remember information or require additional interactions to complete a transaction. This makes the system less usable and can frustrate system users.

Design risk assessment

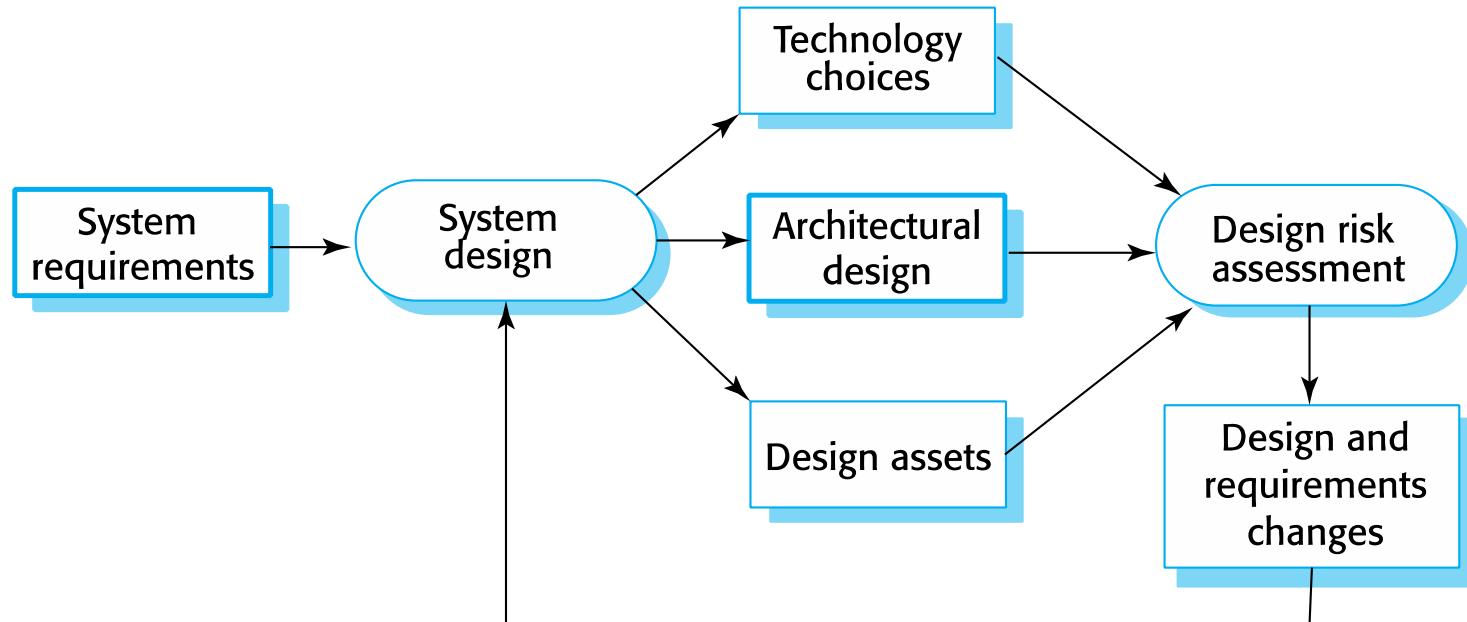


- ✧ Risk assessment while the system is being developed and after it has been deployed
- ✧ More information is available - system platform, middleware and the system architecture and data organisation.
- ✧ Vulnerabilities that arise from design choices may therefore be identified.

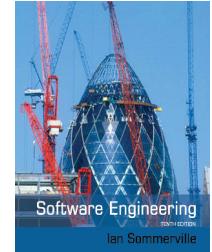
Design and risk assessment



Software Engineering
Ian Sommerville

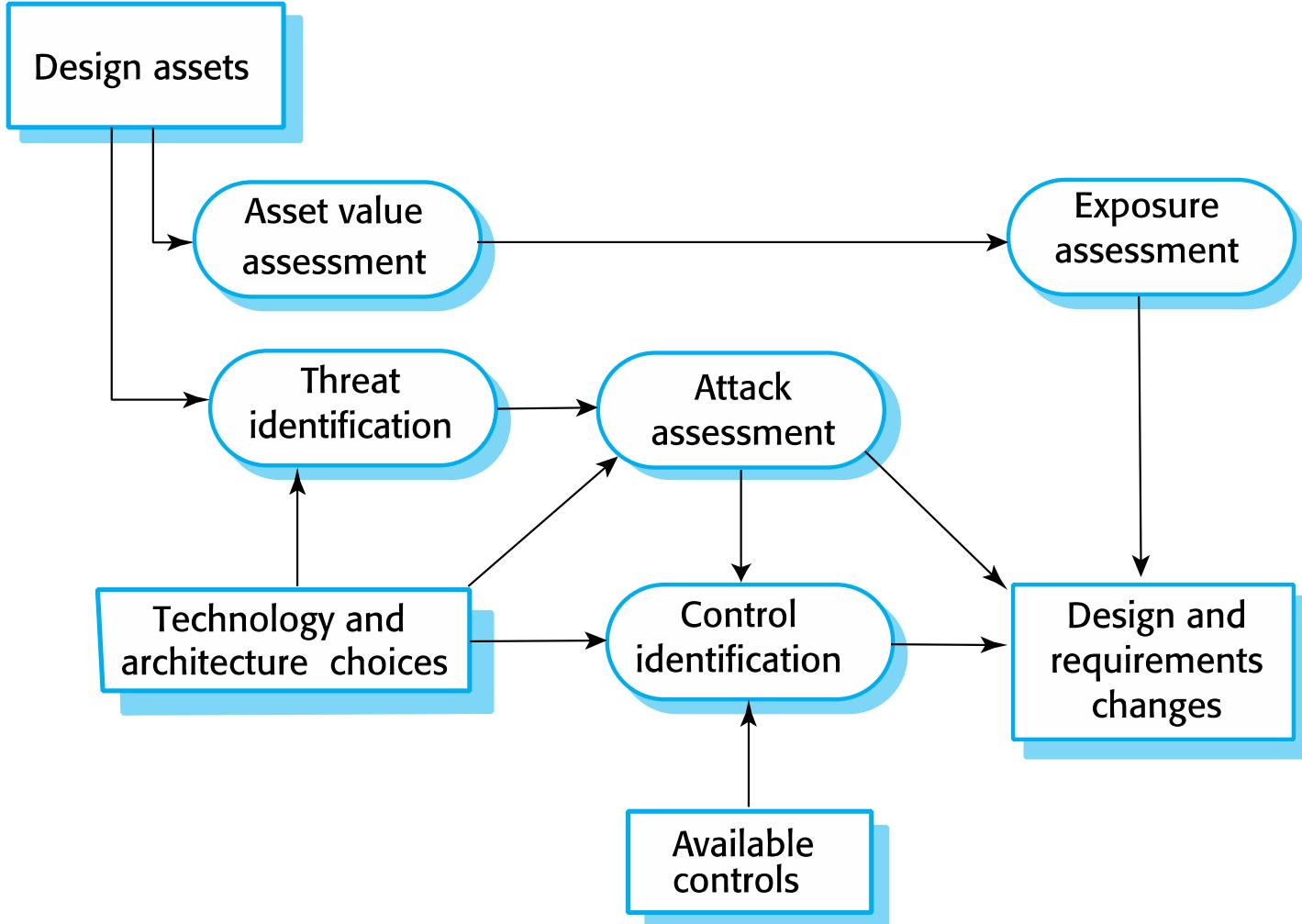
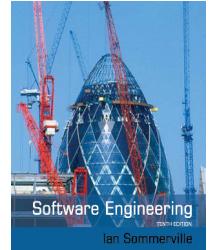


Protection requirements



- ✧ Protection requirements may be generated when knowledge of information representation and system distribution
- ✧ Separating patient and treatment information limits the amount of information (personal patient data) that needs to be protected
- ✧ Maintaining copies of records on a local client protects against denial of service attacks on the server
 - But these may need to be encrypted

Design risk assessment

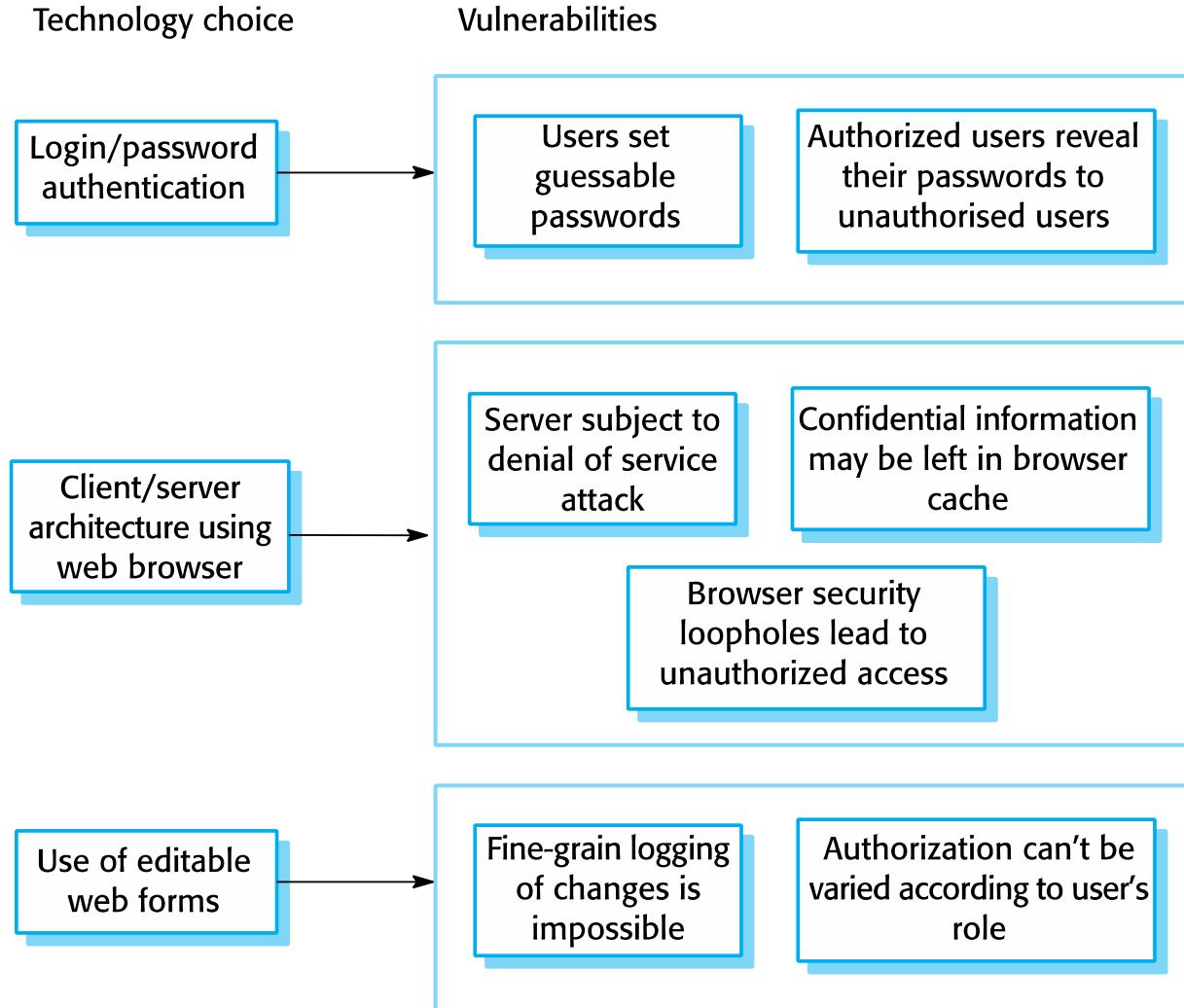
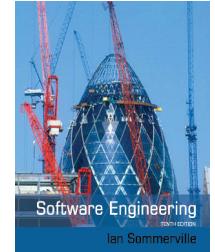


Design decisions from use of COTS

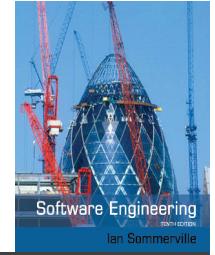


- ✧ System users authenticated using a name/password combination.
- ✧ The system architecture is client-server with clients accessing the system through a standard web browser.
- ✧ Information is presented as an editable web form.

Vulnerabilities associated with technology choices



Security requirements



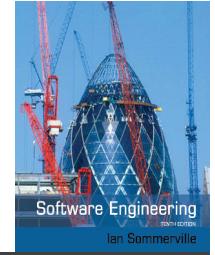
- ✧ A password checker shall be made available and shall be run daily. Weak passwords shall be reported to system administrators.
- ✧ Access to the system shall only be allowed by approved client computers.
- ✧ All client computers shall have a single, approved web browser installed by system administrators.

Architectural design



- ✧ Two fundamental issues have to be considered when designing an architecture for security.
 - Protection
 - How should the system be organised so that critical assets can be protected against external attack?
 - Distribution
 - How should system assets be distributed so that the effects of a successful attack are minimized?
- ✧ These are potentially conflicting
 - If assets are distributed, then they are more expensive to protect.
If assets are protected, then usability and performance requirements may be compromised.

Protection



- ✧ Platform-level protection
 - Top-level controls on the platform on which a system runs.
- ✧ Application-level protection
 - Specific protection mechanisms built into the application itself
 - e.g. additional password protection.
- ✧ Record-level protection
 - Protection that is invoked when access to specific information is requested
- ✧ These lead to a layered protection architecture



A layered protection architecture

Platform level protection

System authentication

System authorization

File integrity management

Application level protection

Database login

Database authorization

Transaction management

Database recovery

Record level protection

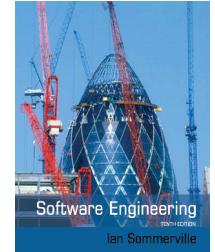
Record access authorization

Record encryption

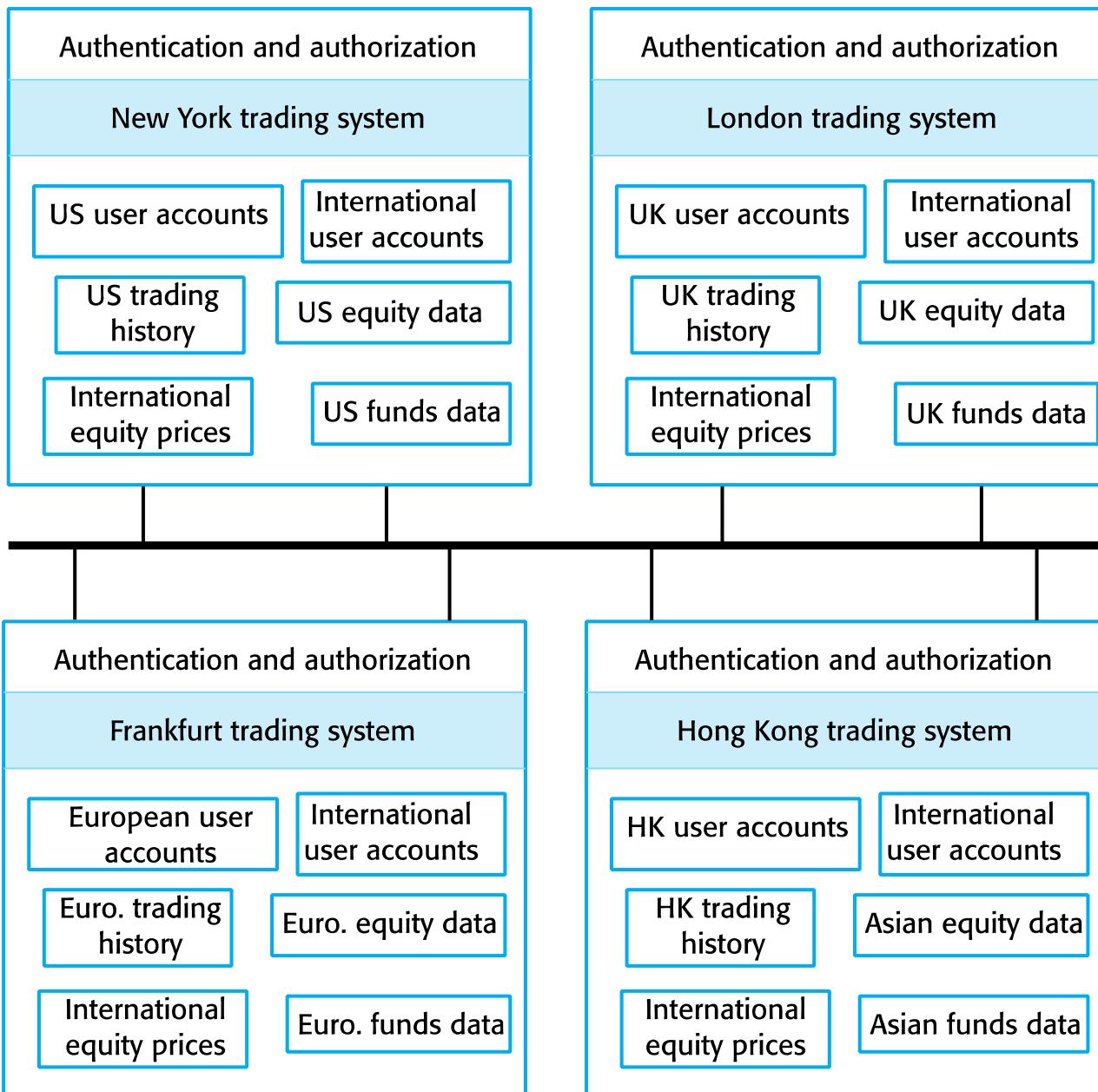
Record integrity management

Patient records

Distribution

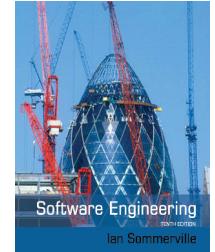


- ✧ Distributing assets means that attacks on one system do not necessarily lead to complete loss of system service
- ✧ Each platform has separate protection features and may be different from other platforms so that they do not share a common vulnerability
- ✧ Distribution is particularly important if the risk of denial of service attacks is high



Distributed assets in an equity trading system

Design guidelines for security engineering

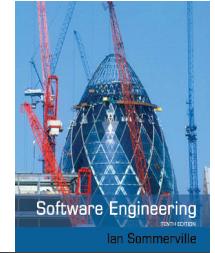


- ✧ Design guidelines encapsulate good practice in secure systems design
- ✧ Design guidelines serve two purposes:
 - They raise awareness of security issues in a software engineering team. Security is considered when design decisions are made.
 - They can be used as the basis of a review checklist that is applied during the system validation process.
- ✧ Design guidelines here are applicable during software specification and design

Design guidelines for secure systems engineering



Security guidelines	
Base security decisions on an explicit security policy	
Avoid a single point of failure	
Fail securely	
Balance security and usability	
Log user actions	
Use redundancy and diversity to reduce risk	
Specify the format of all system inputs	
Compartmentalize your assets	
Design for deployment	
Design for recoverability	



Design guidelines 1-3

- ✧ Base decisions on an explicit security policy
 - Define a security policy for the organization that sets out the fundamental security requirements that should apply to all organizational systems.
- ✧ Avoid a single point of failure
 - Ensure that a security failure can only result when there is more than one failure in security procedures. For example, have password and question-based authentication.
- ✧ Fail securely
 - When systems fail, for whatever reason, ensure that sensitive information cannot be accessed by unauthorized users even although normal security procedures are unavailable.



Design guidelines 4-6

✧ Balance security and usability

- Try to avoid security procedures that make the system difficult to use. Sometimes you have to accept weaker security to make the system more usable.

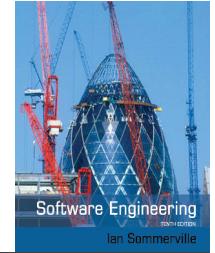
✧ Log user actions

- Maintain a log of user actions that can be analyzed to discover who did what. If users know about such a log, they are less likely to behave in an irresponsible way.

✧ Use redundancy and diversity to reduce risk

- Keep multiple copies of data and use diverse infrastructure so that an infrastructure vulnerability cannot be the single point of failure.

Design guidelines 7-10



- ✧ Specify the format of all system inputs
 - If input formats are known then you can check that all inputs are within range so that unexpected inputs don't cause problems.
- ✧ Compartmentalize your assets
 - Organize the system so that assets are in separate areas and users only have access to the information that they need rather than all system information.
- ✧ Design for deployment
 - Design the system to avoid deployment problems
- ✧ Design for recoverability
 - Design the system to simplify recoverability after a successful attack.

Secure systems programming



Aspects of secure systems programming



- ✧ Vulnerabilities are often language-specific.
 - Array bound checking is automatic in languages like Java so this is not a vulnerability that can be exploited in Java programs.
 - However, millions of programs are written in C and C++ as these allow for the development of more efficient software so simply avoiding the use of these languages is not a realistic option.
- ✧ Security vulnerabilities are closely related to program reliability.
 - Programs without array bound checking can crash so actions taken to improve program reliability can also improve system security.

Dependable programming guidelines

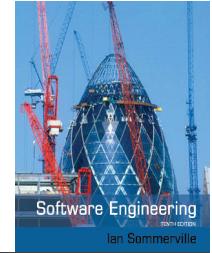


Dependable programming guidelines

- 1. Limit the visibility of information in a program**
- 2. Check all inputs for validity**
- 3. Provide a handler for all exceptions**
- 4. Minimize the use of error-prone constructs**
- 5. Provide restart capabilities**
- 6. Check array bounds**
- 7. Include timeouts when calling external components**
- 8. Name all constants that represent real-world values**



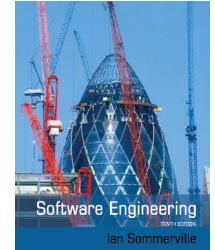
Security testing and assurance



Security testing

- ✧ Testing the extent to which the system can protect itself from external attacks.
- ✧ Problems with security testing
 - Security requirements are ‘shall not’ requirements i.e. they specify what should not happen. It is not usually possible to define security requirements as simple constraints that can be checked by the system.
 - The people attacking a system are intelligent and look for vulnerabilities. They can experiment to discover weaknesses and loopholes in the system.

Security validation



✧ Experience-based testing

- The system is reviewed and analysed against the types of attack that are known to the validation team.

✧ Penetration testing

- A team is established whose goal is to breach the security of the system by simulating attacks on the system.

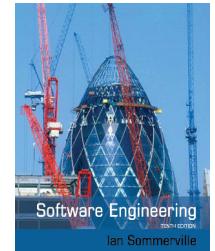
✧ Tool-based analysis

- Various security tools such as password checkers are used to analyse the system in operation.

✧ Formal verification

- The system is verified against a formal security specification.

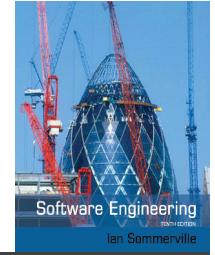
Examples of entries in a security checklist



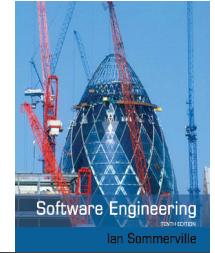
Security checklist

1. Do all files that are created in the application have appropriate access permissions? The wrong access permissions may lead to these files being accessed by unauthorized users.
2. Does the system automatically terminate user sessions after a period of inactivity? Sessions that are left active may allow unauthorized access through an unattended computer.
3. If the system is written in a programming language without array bound checking, are there situations where buffer overflow may be exploited? Buffer overflow may allow attackers to send code strings to the system and then execute them.
4. If passwords are set, does the system check that passwords are ‘strong’? Strong passwords consist of mixed letters, numbers, and punctuation, and are not normal dictionary entries. They are more difficult to break than simple passwords.
5. Are inputs from the system’s environment always checked against an input specification? Incorrect processing of badly formed inputs is a common cause of security vulnerabilities.

Key points



- ✧ Security engineering is concerned with how to develop systems that can resist malicious attacks
- ✧ Security threats can be threats to confidentiality, integrity or availability of a system or its data
- ✧ Security risk management is concerned with assessing possible losses from attacks and deriving security requirements to minimise losses
- ✧ To specify security requirements, you should identify the assets that are to be protected and define how security techniques and technology should be used to protect these assets.



Key points

- ✧ Key issues when designing a secure systems architecture include organizing the system structure to protect key assets and distributing the system assets to minimize the losses from a successful attack.
- ✧ Security design guidelines sensitize system designers to security issues that they may not have considered. They provide a basis for creating security review checklists.
- ✧ Security validation is difficult because security requirements state what should not happen in a system, rather than what should. Furthermore, system attackers are intelligent and may have more time to probe for weaknesses than is available for security testing.



Software Engineering

Ian Sommerville

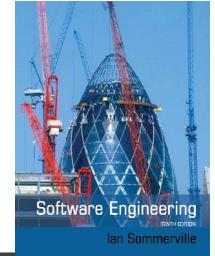
Chapter 14 – Resilience Engineering

Topics covered



- ✧ Cybersecurity
- ✧ Sociotechnical resilience
- ✧ Resilient systems design

Resilience



- ✧ *The resilience of a system is a judgment of how well that system can maintain the continuity of its critical services in the presence of disruptive events, such as equipment failure and cyberattacks.*

- ✧ Cyberattacks by malicious outsiders are perhaps the most serious threat faced by networked systems but resilience is also intended to cope with system failures and other disruptive events.

Essential resilience ideas



- ✧ The idea that some of the services offered by a system are critical services whose failure could have serious human, social or economic effects.
- ✧ The idea that some events are disruptive and can affect the ability of a system to deliver its critical services.
- ✧ The idea that resilience is a judgment – there are no resilience metrics and resilience cannot be measured. The resilience of a system can only be assessed by experts, who can examine the system and its operational processes.

Resilience engineering assumptions



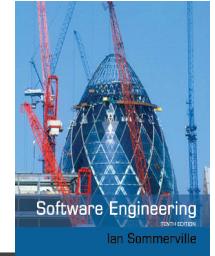
- ✧ Resilience engineering assumes that it is impossible to avoid system failures and so is concerned with limiting the costs of these failures and recovering from them.
- ✧ Resilience engineering assumes that good reliability engineering practices have been used to minimize the number of technical faults in a system.
- ✧ It therefore places more emphasis on limiting the number of system failures that arise from external events such as operator errors or cyberattacks.

Resilience activities



- ✧ *Recognition* The system or its operators should recognise early indications of system failure.
- ✧ *Resistance* If the symptoms of a problem or cyberattack are detected early, then resistance strategies may be used to reduce the probability that the system will fail.
- ✧ *Recovery* If a failure occurs, the recovery activity ensures that critical system services are restored quickly so that system users are not badly affected by failure.
- ✧ *Reinstatement* In this final activity, all of the system services are restored and normal system operation can continue.

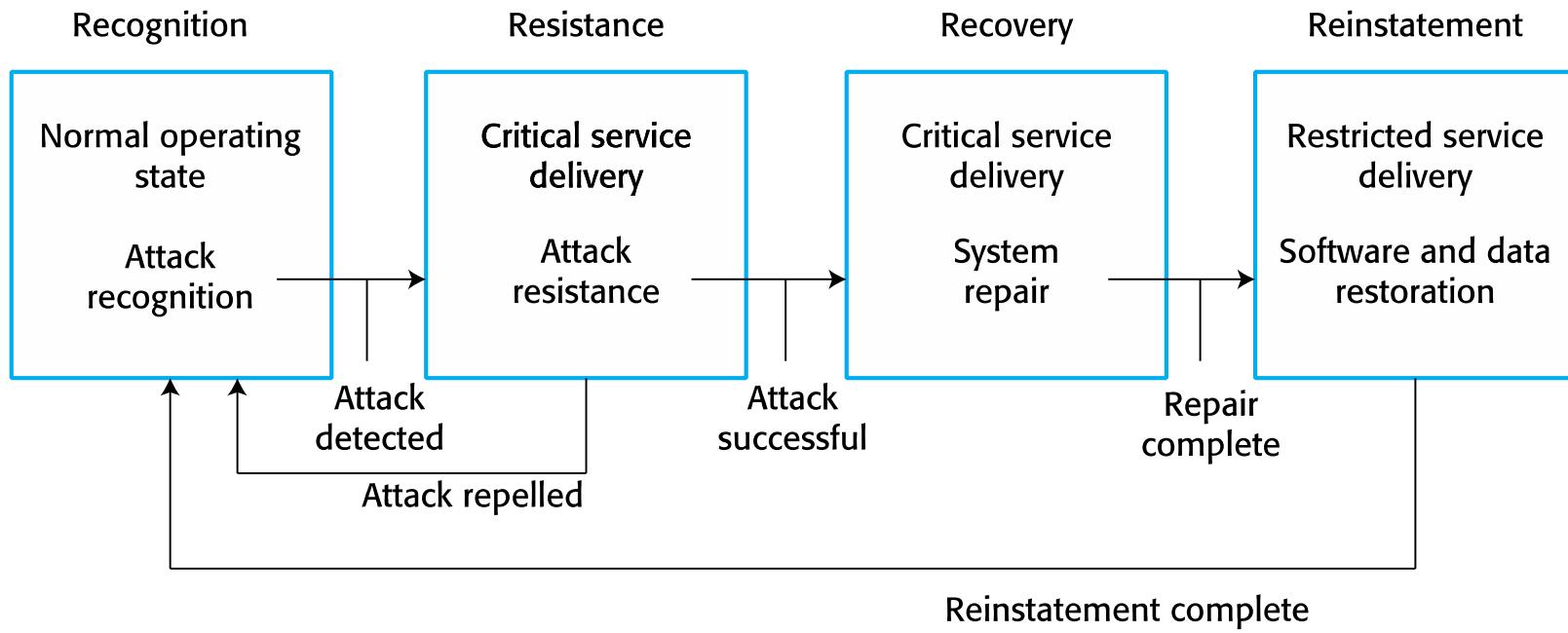
Resistance



Software Engineering
Ian Sommerville

- ✧ Resistance strategies may focus on isolating critical parts of the system so that they are unaffected by problems elsewhere.
- ✧ Resistance includes proactive resistance where defences are included in a system to trap problems and reactive resistance where actions are taken when a problem is discovered.

Resilience activities





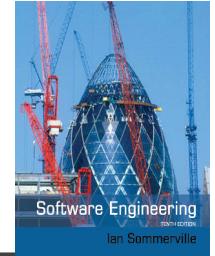
Cybersecurity

Cybersecurity



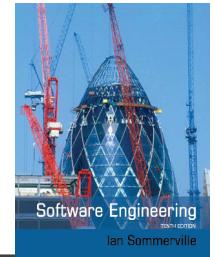
- ✧ Cybercrime is the illegal use of networked systems and is one of the most serious problems facing our society.
- ✧ Cybersecurity is a broader topic than system security engineering
 - Cybersecurity is a sociotchnical issue covering all aspects of ensuring the protection of citizens, businesses and critical infrastructures from threats that arise from their use of computers and the Internet.
- ✧ Cybersecurity is concerned with all of an organization's IT assets from networks through to application systems.

Factors contributing to cybersecurity failure



- ✧ organizational ignorance of the seriousness of the problem,
- ✧ poor design and lax application of security procedures,
- ✧ human carelessness,
- ✧ inappropriate trade-offs between usability and security.

Cybersecurity threats



- ✧ *Threats to the confidentiality of assets* Data is not damaged but it is made available to people who should not have access to it.
- ✧ *Threats to the integrity of assets* These are threats where systems or data are damaged in some way by a cyberattack.
- ✧ *Threats to the availability of assets* These are threats that aim to deny use of assets by authorized users.

Examples of controls



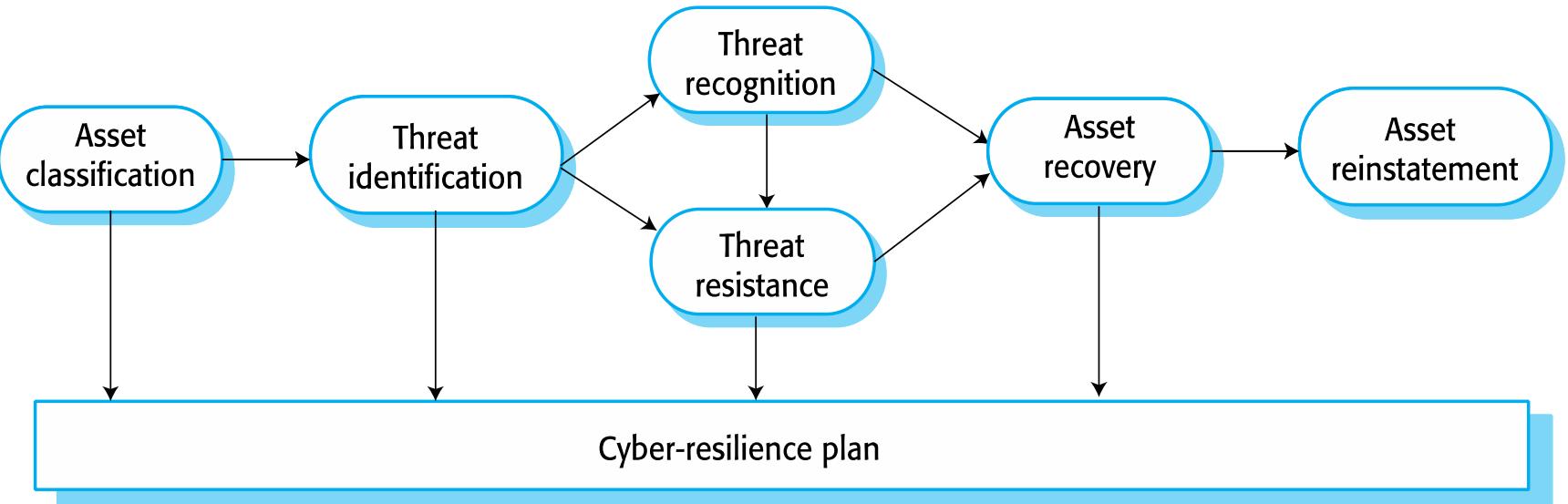
- ✧ Authentication, where users of a system have to show that they are authorized to access the system
- ✧ Encryption, where data is algorithmically scrambled so that an unauthorized reader cannot access the information.
- ✧ Firewalls, where incoming network packets are examined then accepted or rejected according to a set of organizational rules.
 - Firewalls can be used to ensure that only traffic from trusted sources is passed from the external Internet into the local organizational network.

Redundancy and diversity

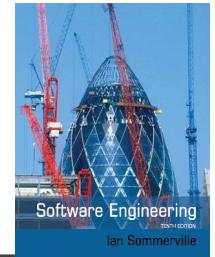


- ✧ Copies of data and software should be maintained on separate computer systems.
 - This supports recovery after a successful cyberattack. (recovery and reinstatement)
- ✧ Multi-stage diverse authentication can protect against password attacks.
 - This is a resistance measure
- ✧ Critical servers may be over-provisioned i.e. they may be more powerful than is required to handle their expected load. Attacks can be resisted without serious service degradation.

Cyber-resilience planning



Cyber resilience planning



✧ *Asset classification*

- The organization's hardware, software and human assets are examined and classified depending on how essential they are to normal operations.

✧ *Threat identification*

- For each of the assets (or, at least the critical and important assets), you should identify and classify threats to that asset.

✧ *Threat recognition*

- For each threat or, sometimes asset/threat pair, you should identify how an attack based on that threat might be recognised.

Cyber resilience planning



✧ *Threat resistance*

- For each threat or asset/threat pair, you should identify possible resistance strategies. These may be either embedded in the system (technical strategies) or may rely on operational procedures.

✧ *Asset recovery*

- For each critical asset or asset/threat pair, you should work out how that asset could be recovered in the event of a successful cyberattack.

✧ *Asset reinstatement*

- This is a more general process of asset recovery where you define procedures to bring the system back into normal operation.

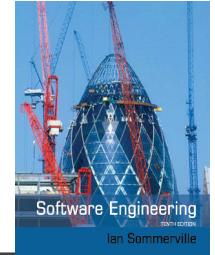


Software Engineering

Ian Sommerville

Sociotechnical resilience

Sociotechnical resilience



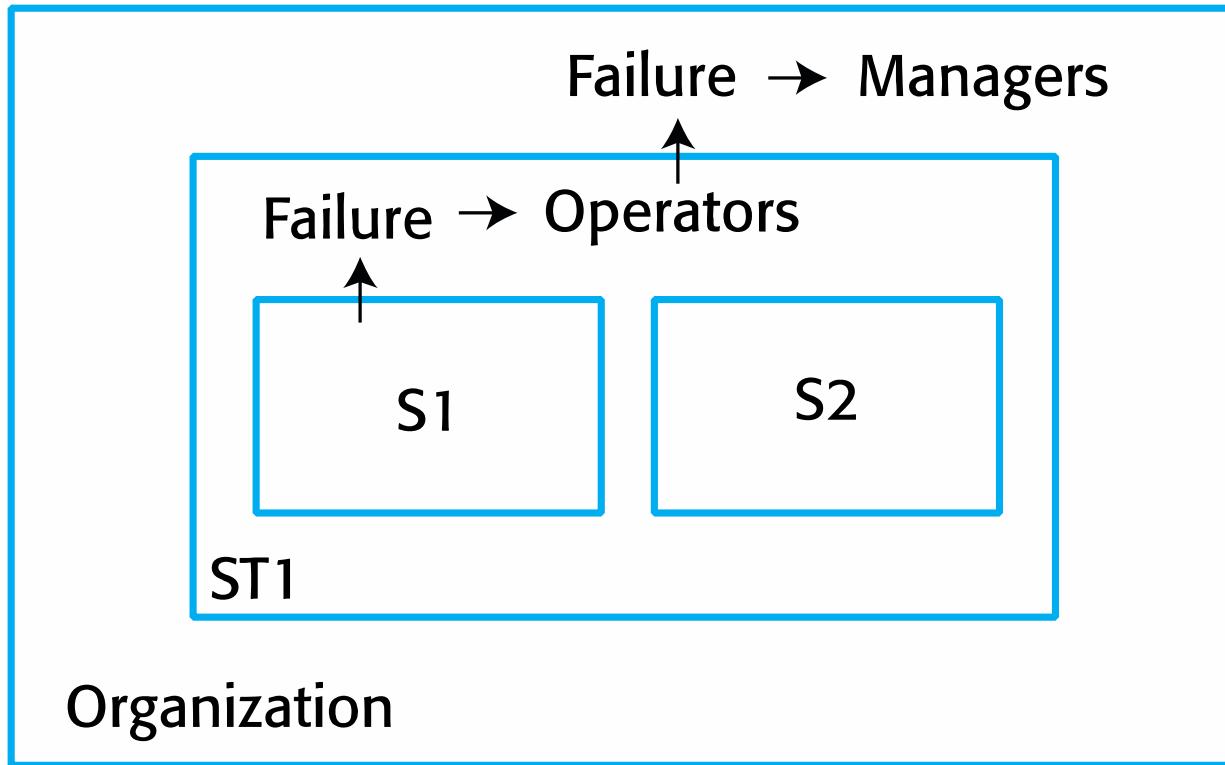
- ✧ Resilience engineering is concerned with adverse external events that can lead to system failure.
- ✧ To design a resilient system, you have to think about sociotechnical systems design and not exclusively focus on software.
- ✧ Dealing with these events is often easier and more effective in the broader sociotechnical system.

Mentcare example

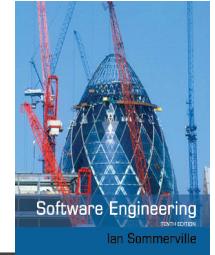


- ✧ Cyberattack may aim to steal data, gaining access using a legitimate user's credentials
- ✧ Technical solution may be to use more complex authentication procedures.
- ✧ These irritate users and may reduce security as users leave systems unattended without logging out.
- ✧ A better strategy may be to introduce organizational policies and procedures that emphasise the importance of not sharing login credentials and that tell users about easy ways to create and maintain strong passwords.

Nested technical and sociotechnical systems

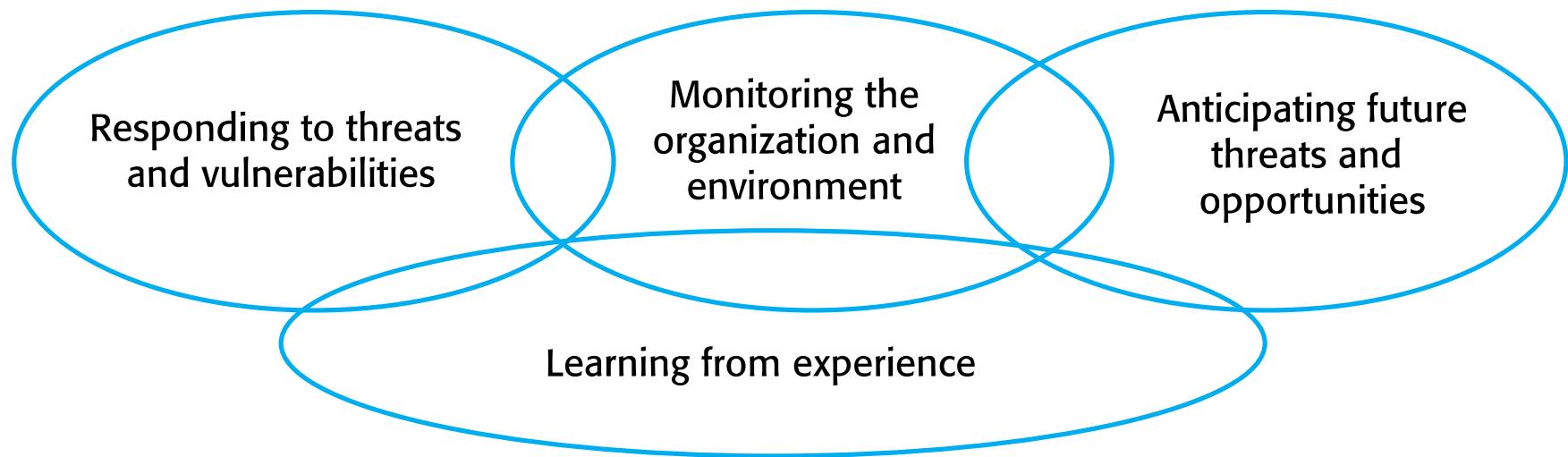
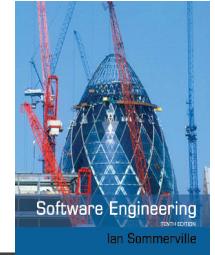


Failure hierarchy

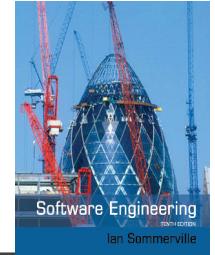


- ✧ A failure in system S1 may be trapped in the broader sociotechnical system ST1 through operator actions
- ✧ Organizational damage is therefore limited
- ✧ If the failure in S1 leads to a failure in ST1, then it is up to managers in the broader organization to deal with that failure.

Characteristics of resilient organizations



Organizational resilience



- ✧ There are four characteristics that reflect the resilience of an organization
 - Responsiveness, monitoring, anticipation, learning
- ✧ *The ability to respond*
 - Organizations have to be able to adapt their processes and procedures in response to risks. These risks may be anticipated risks or may be detected threats to the organization and its systems.
- ✧ *The ability to monitor*
 - Organizations should monitor both their internal operations and their external environment for threats before they arise.

Organizational resilience



✧ *The ability to anticipate*

- A resilient organization should not simply focus on its current operations but should anticipate possible future events and changes that may affect its operations and resilience.

✧ *The ability to learn*

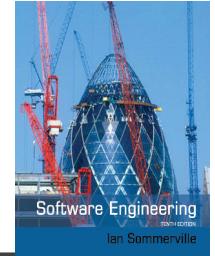
- Organizational resilience can be improved by learning from experience. It is particularly important to learn from successful responses to adverse events such as the effective resistance of a cyberattack. Learning from success allows

Human error



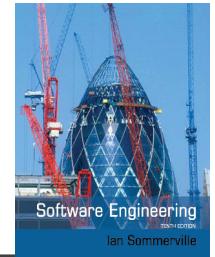
- ✧ People inevitably make mistakes (human errors) that sometimes lead to serious system failures.
- ✧ There are two ways to consider human error
 - *The person approach.* Errors are considered to be the responsibility of the individual and ‘unsafe acts’ (such as an operator failing to engage a safety barrier) are a consequence of individual carelessness or reckless behaviour.
 - *The systems approach.* The basic assumption is that people are fallible and will make mistakes. People make mistakes because they are under pressure from high workloads, poor training or because of inappropriate system design.

Systems approach

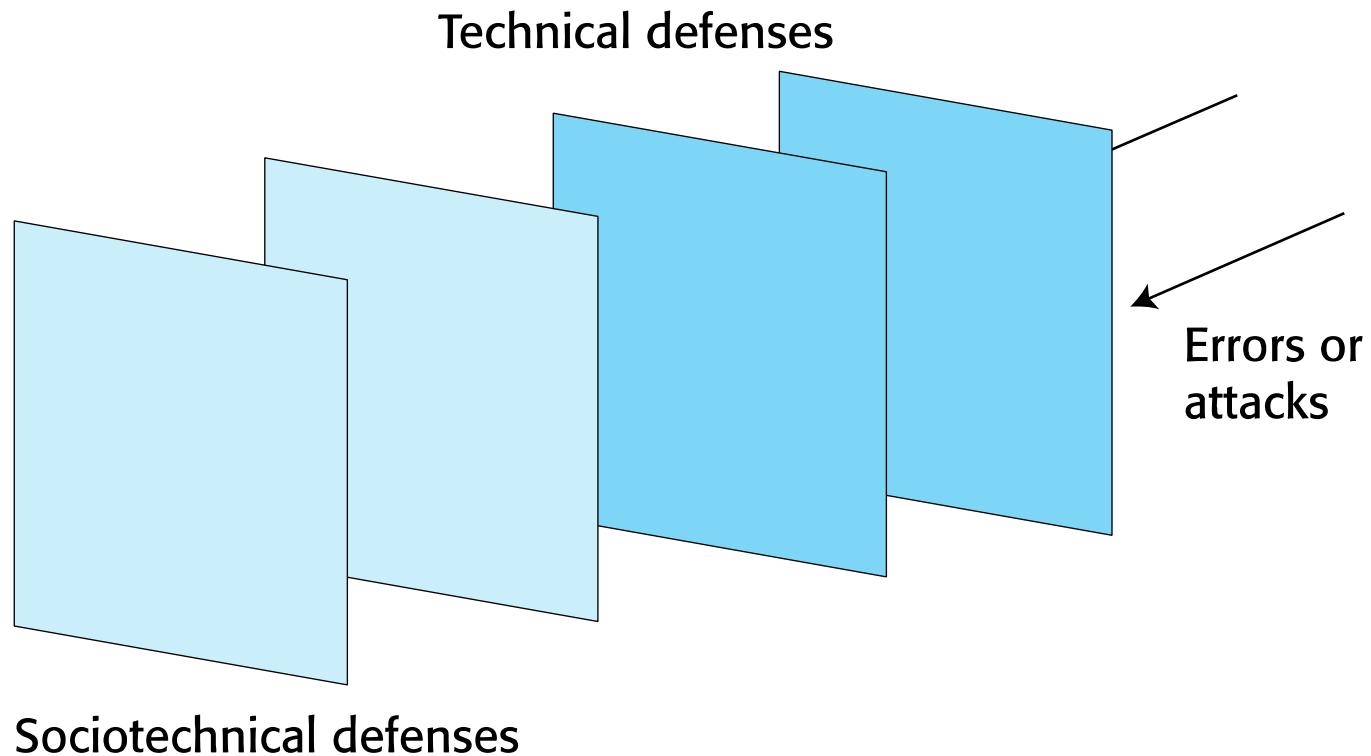


- ✧ Systems engineers should assume that human errors will occur during system operation.
- ✧ To improve the resilience of a system, designers have to think about the defences and barriers to human error that could be part of a system.
- ✧ Can these barriers should be built into the technical components of the system (technical barriers)? If not, they could be part of the processes, procedures and guidelines for using the system (sociotechnical barriers).

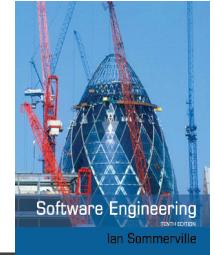
Defensive layers



Software Engineering
Ian Sommerville

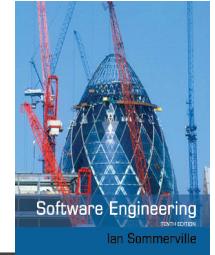


Defensive layers

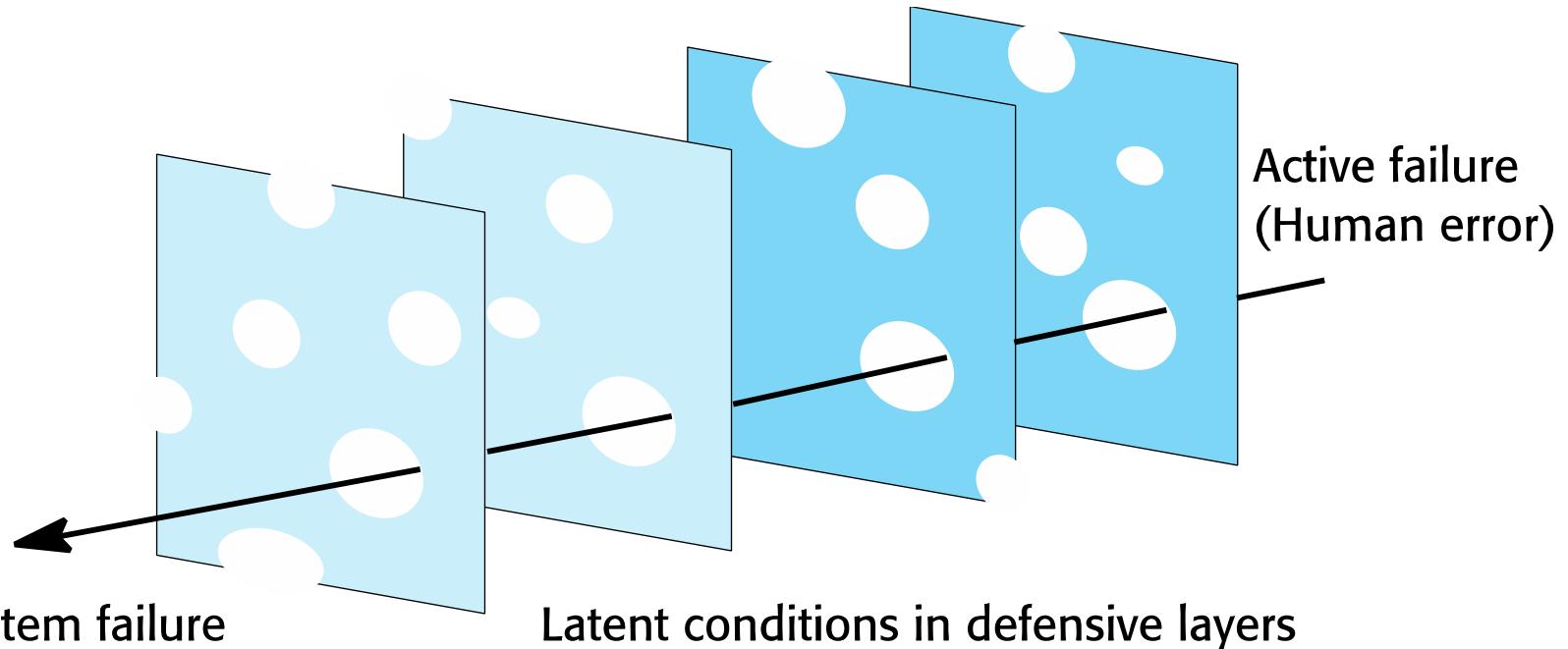


- ✧ You should use redundancy and diversity to create a set of defensive layers, where each layer uses a different approach to deter attackers or trap technical/human failures.
- ✧ ATC system examples
 - Conflict alert system
 - Formalized recording procedures
 - Collaborative checking

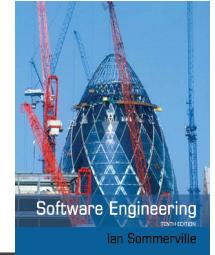
Reason's Swiss Cheese Model



Software Engineering
Ian Sommerville



Swiss Cheese model



- ✧ Defensive layers have vulnerabilities
 - They are like slices of Swiss cheese with holes in the layer corresponding to these vulnerabilities.
- ✧ Vulnerabilities are dynamic
 - The 'holes' are not always in the same place and the size of the holes may vary depending on the operating conditions.
- ✧ System failures occur when the holes line up and all of the defenses fail.

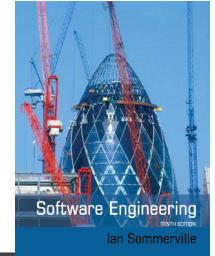
Increasing system resilience



Software Engineering
Ian Sommerville

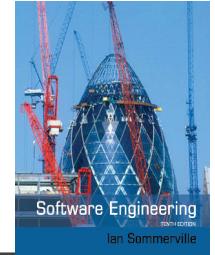
- ✧ Reduce the probability of the occurrence of an external event that might trigger system failures.
- ✧ Increase the number of defensive layers.
 - The more layers that you have in a system, the less likely it is that the holes will line up and a system failure occur.
- ✧ Design a system so that diverse types of barriers are included.
 - The ‘holes’ will probably be in different places and so there is less chance of the holes lining up and failing to trap an error.
- ✧ Minimize the number of latent conditions in a system.
 - This means reducing the number and size of system ‘holes’.

Operational and management processes



- ✧ All software systems have associated operational processes that reflect the assumptions of the designers about how these systems will be used.
- ✧ For example, in an imaging system in a hospital, the operator may have the responsibility of checking the quality of the images immediately after these have been processed.
- ✧ This allows the imaging procedure to be repeated if there is a problem.

Operational processes



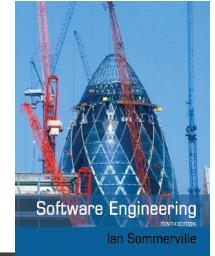
- ✧ Operational processes are the processes that are involved in using the system for its defined purpose.
- ✧ For new systems, these operational processes have to be defined and documented during the system development process.
- ✧ Operators may have to be trained and other work processes adapted to make effective use of the new system.

Personal and Enterprise IT processes



- ✧ For personal systems, the designers may describe the expected use of the system but have no control over how users will actually behave.
- ✧ For enterprise IT systems, however, there may be training for users to teach them how to use the system.
- ✧ Although user behaviour cannot be controlled, it is reasonable to expect that they will normally follow the defined process.

Process design



- ✧ Operational and management processes are an important defense mechanism and, in designing a process, you need to find a balance between efficient operation and problem management.
- ✧ Process improvement focuses on identifying and codifying good practice and developing software to support this.
- ✧ If process improvement focuses on efficiency, then this can make it more difficult to deal with problems when these arise.

Efficiency and resilience



Efficient process operation	Problem management
Process optimization and control	Process flexibility and adaptability
Information hiding and security	Information sharing and visibility
Automation to reduce operator workload with fewer operators and managers	Manual processes and spare operator/manager capacity to deal with problems
Role specialization	Role sharing

Coping with failures



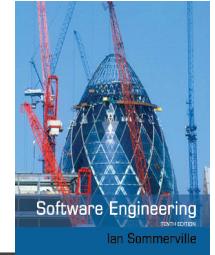
- ✧ What seems to be ‘inefficient’ practice often arises because people maintain redundant information or share information because they know this makes it easier to deal with problems when things go wrong.
- ✧ When things go wrong, operators and system managers can often recover the situation although this may sometimes mean that they have to break rules and ‘work around’ the defined process.
- ✧ You should therefore design operational processes to be flexible and adaptable.

Information provision and management



- ✧ To make a process more efficient, it may make sense to present operators with the information that they need, when they need it.
- ✧ If operators are only presented with information that the process designer thinks that they 'need to know' then they may be unable to detect problems that do not directly affect their immediate tasks.
- ✧ When things go wrong, the system operators do not have a broad picture of what is happening in the system, so it is more difficult for them to formulate strategies for dealing with problems.

Process automation



- ✧ Process automation can have both positive and negative effects on system resilience.
- ✧ If the automated system works properly, it can detect problems, invoke cyberattack resistance if necessary and start automated recovery procedures.
- ✧ However, if the problem can't be handled by the automated system, there are fewer people available to tackle the problem and the system may have been damaged by the process automation doing the wrong thing.

Disadvantages of process automation



- ✧ Automated management systems may go wrong and take incorrect actions. As problems develop, the system may take unexpected actions that make the situation worse and which cannot be understood by the system managers.
- ✧ Problem solving is a collaborative process. If fewer managers are available, it is likely to take longer to work out a strategy to recover from a problem or cyberattack.

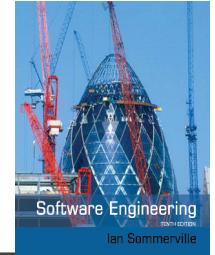


Software Engineering

Ian Sommerville

Resilient systems design

Resilient systems design



Software Engineering
Ian Sommerville

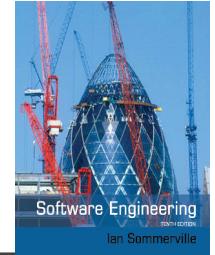
✧ *Identifying critical services and assets*

- Critical services and assets are those elements of the system that allow a system to fulfill its primary purpose.
- For example, the critical services in a system that handles ambulance dispatch are those concerned with taking calls and dispatching ambulances.

✧ *Designing system components that support problem recognition, resistance, recovery and reinstatement*

- For example, in an ambulance dispatch system, a watchdog timer may be included to detect if the system is not responding to events.

Survivable systems analysis



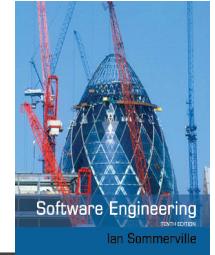
✧ *System understanding*

- For an existing or proposed system, review the goals of the system (sometimes called the mission objectives), the system requirements and the system architecture.

✧ *Critical service identification*

- The services that must always be maintained and the components that are required to maintain these services are identified.

Survivable systems analysis



✧ *Attack simulation*

- Scenarios or use cases for possible attacks are identified along with the system components that would be affected by these attacks.

✧ *Survivability analysis*

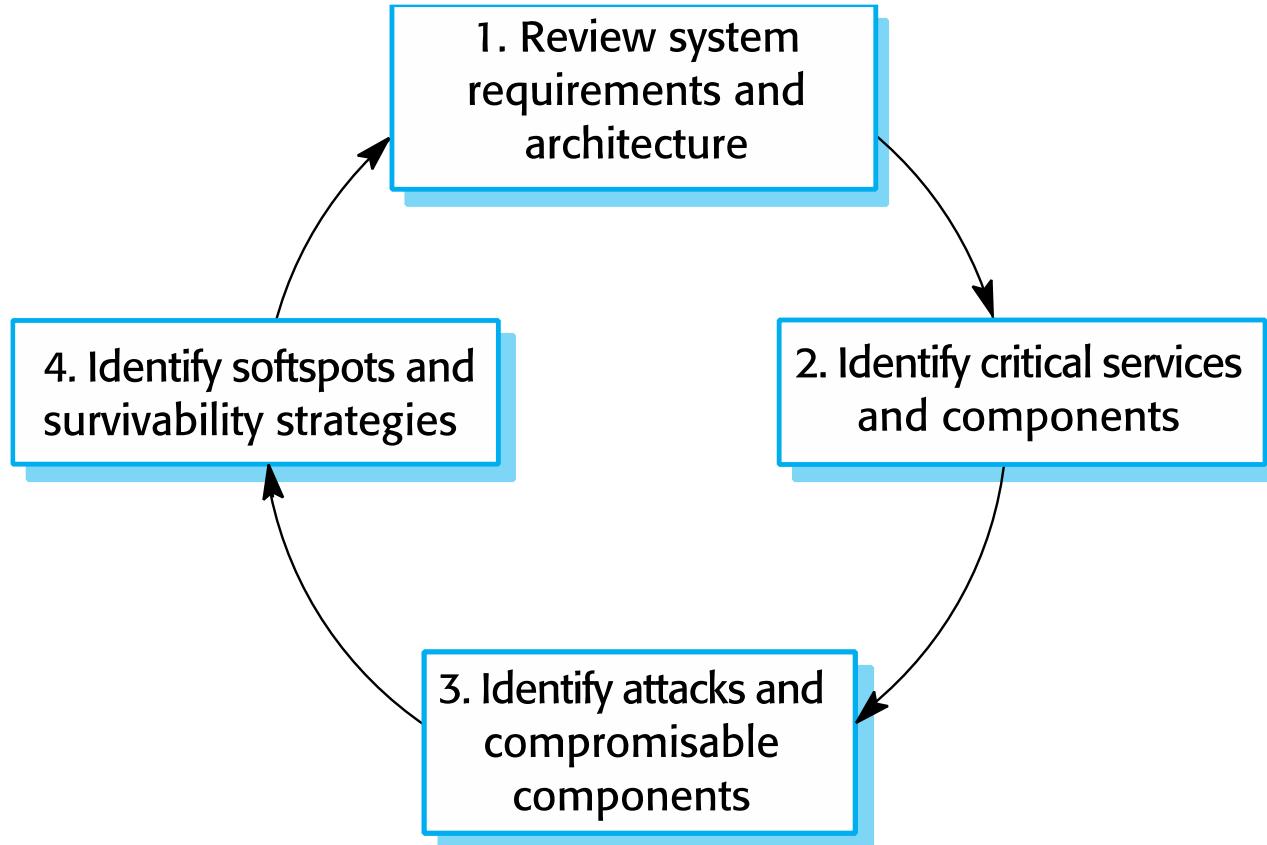
- Components that are both essential and compromisable by an attack are identified and survivability strategies based on resistance, recognition and recovery are identified.

Stages in survivability analysis

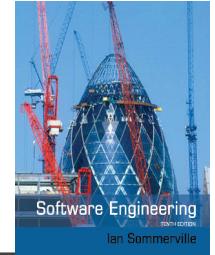


Software Engineering

Ian Sommerville

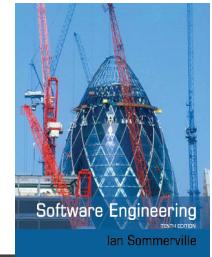


Problems for business systems

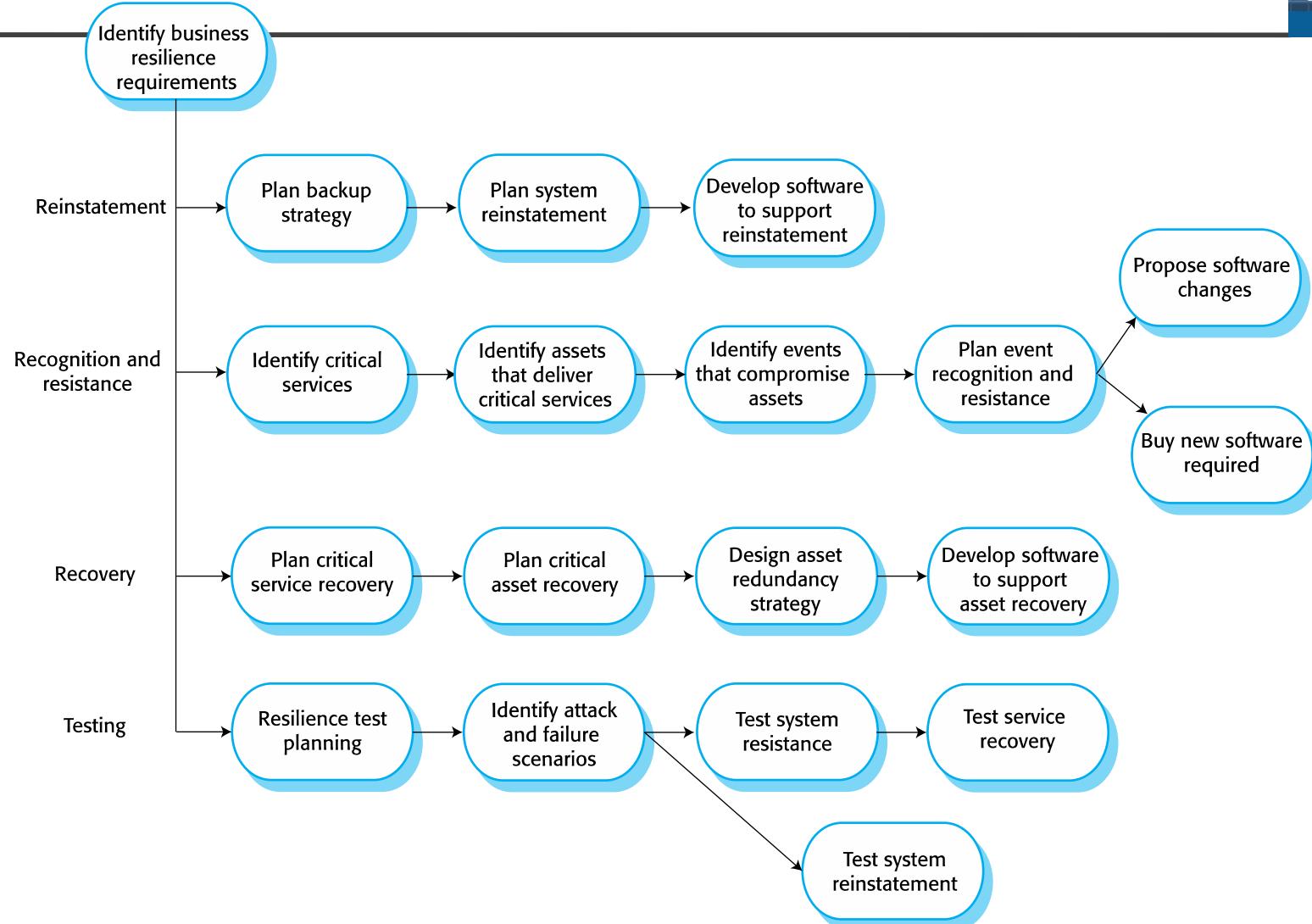


- ✧ The fundamental problem with this approach to survivability analysis is that its starting point is the requirements and architecture documentation for a system.
- ✧ However for business systems:
 - It is not explicitly related to the business requirements for resilience. I believe that these are a more appropriate starting point than technical system requirements.
 - It assumes that there is a detailed requirements statement for a system. In fact, resilience may have to be 'retrofitted' to a system where there is no complete or up-to-date requirements document.

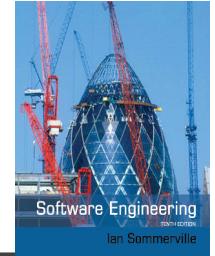
Resilience engineering



Software Engineering
Ian Sommerville



Streams of work in resilience engineering



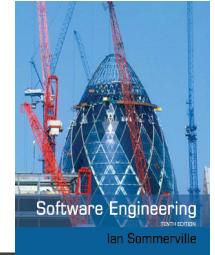
- ✧ Identify business resilience requirements
- ✧ Plan how to reinstate systems to their normal operating state
- ✧ Identify system failures and cyberattacks that can compromise a system
- ✧ Plan how to recover critical services quickly after damage or a cyberattack
- ✧ Test all aspects of resilience planning

Maintaining critical service availability



- ✧ To maintain availability, you need to know:
 - the system services that are the most critical for a business,
 - the minimal quality of service that must be maintained,
 - how these services might be compromised,
 - how these services can be protected,
 - how you can recover quickly if the services become unavailable.
- ✧ Critical assets are identified during service analysis.
 - Assets may be hardware, software, data or people.

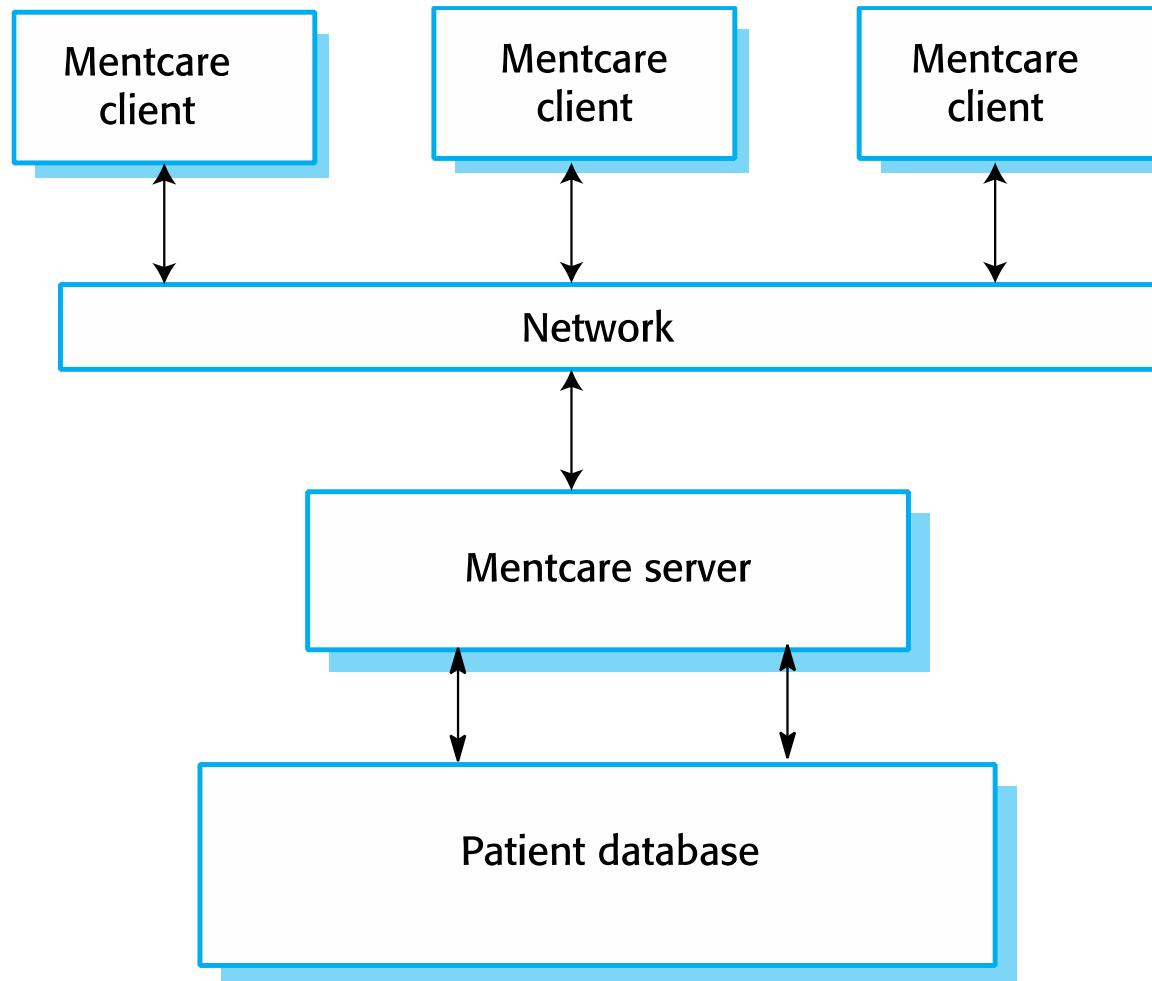
Mentcare system resilience



- ✧ The Mentcare system is a system used to support clinicians treating patients that suffer from mental health problems.
- ✧ It provides patient information and records of consultations with doctors and nurses.
- ✧ It includes checks that can flag patients who may be dangerous or suicidal.
- ✧ Based on a client-server architecture.



Client-server architecture (Mentcare)



Critical Mentcare services



- ✧ An information service that provides information about a patient's current diagnosis and treatment plan.
- ✧ A warning service that highlights patients that could pose a danger to others or to themselves.
- ✧ Availability of the complete patient record is NOT a critical service as routine patient information is not normally required during consultations.

Assets required for normal service operation



- ✧ The patient record database that maintains all patient information.
- ✧ A database server that provides access to the database for local client computers.
- ✧ A network for client/server communication.
- ✧ Local laptop or desktop computers used by clinicians to access patient information.
- ✧ A set of rules to identify patients who are potentially dangerous and which can flag patient records. Client software that highlights dangerous patients to system users.

Adverse events



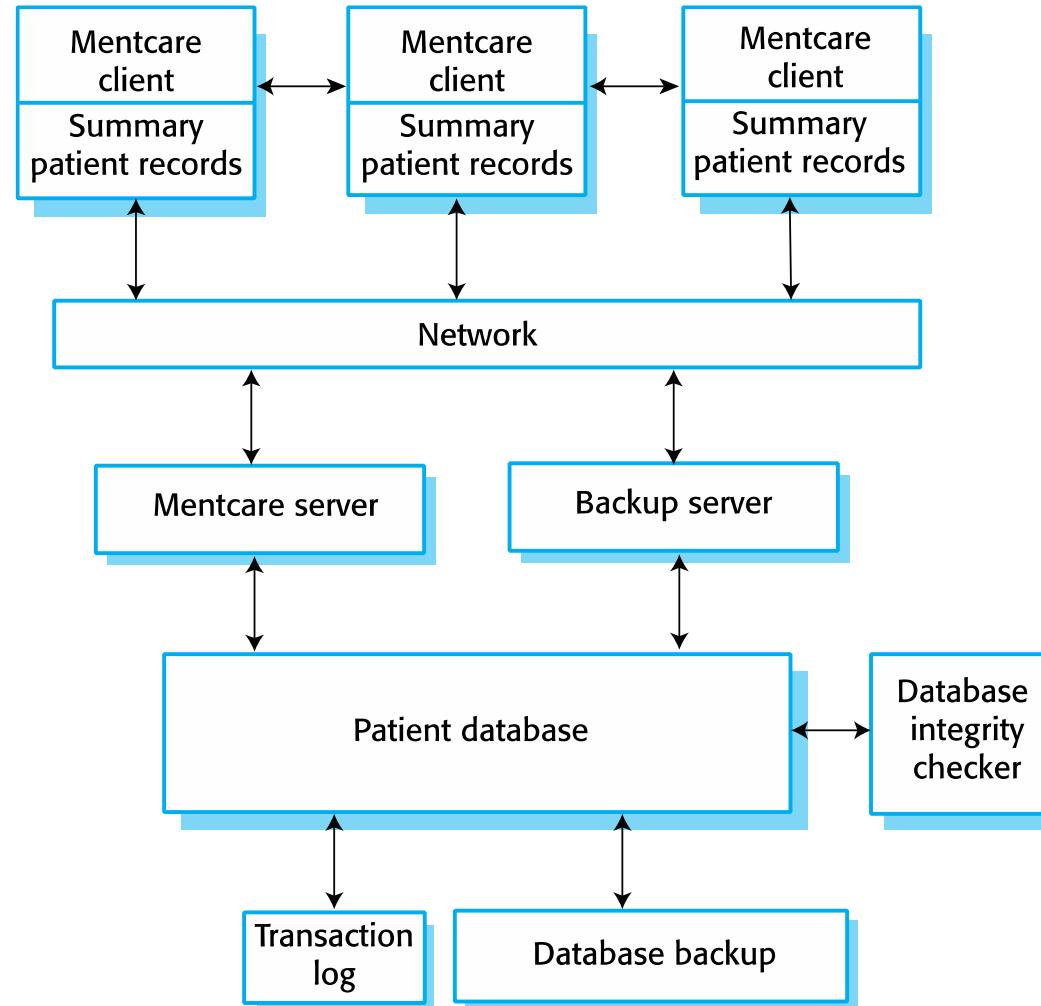
- ✧ Unavailability of the database server either through a system failure, a network failure or a denial of service cyberattack
- ✧ Deliberate or accidental corruption of the patient record database or the rules that define what is meant by a 'dangerous patient'
- ✧ Infection of client computers with malware
- ✧ Access to client computers by unauthorized people who gain access to patient records

Recognition and resistance strategies



Event	Recognition	Resistance
Server unavailability	<ol style="list-style-type: none">1. Watchdog timer on client that times out if no response to client access2. Text messages from system managers to clinical users	<ol style="list-style-type: none">1. Design system architecture to maintain local copies of critical information2. Provide peer-to-peer search across clients for patient data3. Provide staff with smart phones that can be used to access the network in the event of server failure4. Provide backup server
Patient database corruption	<ol style="list-style-type: none">1. Record level cryptographic checksums2. Regular auto-checking of database integrity3. Reporting system for incorrect information	<ol style="list-style-type: none">1. Replayable transaction log to update database backup with recent transactions2. Maintenance of local copies of patient information and software to restore database from local copies and backups
Malware infection of client computers	<ol style="list-style-type: none">1. Reporting system so that computer users can report unusual behaviour.2. Automated malware checks on startup.	<ol style="list-style-type: none">1. Security awareness workshops for all system users2. Disabling of USB ports on client computers3. Automated system setup for new clients4. Support access to system from mobile devices5. Installation of security software
Unauthorized access to patient information	<ol style="list-style-type: none">1. Warning text messages from users about possible intruders2. Log analysis for unusual activity	<ol style="list-style-type: none">1. Multi-level system authentication process2. Disabling of USB ports on client computers3. Access logging and real-time log analysis4. Security awareness workshops for all system users

Mentcare system resilience

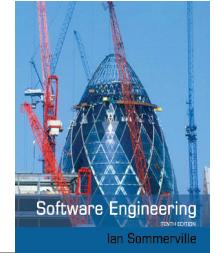


Architecture for resilience

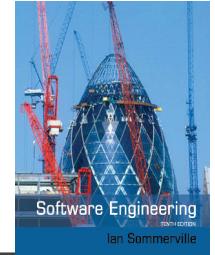


- ✧ Summary patient records that are maintained on local client computers.
 - The local computers can communicate directly with each other and exchange information using either the system network or using an *ad hoc* network created using mobile phones. If the database is unavailable, doctors and nurses can still access essential patient information.
- ✧ A backup server to allow for main server failure.
 - This server is responsible for taking regular snapshots of the database as backups. In the event of the failure of the main server, it can also act as the main server for the whole system.

Architecture for resilience



- ✧ Database integrity checking and recovery software.
 - Integrity checking runs as a background task checking for signs of database corruption. If corruption is discovered, it can automatically initiate the recovery of some or all of the data from backups. The transaction log allows these backups to be updated with details of recent changes



Critical service maintenance

Software Engineering
Ian Sommerville

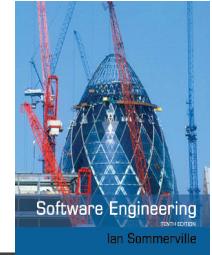
- ✧ By downloading information to the client at the start of a clinic session, the consultation can continue without server access.
 - Only the information about the patients who are scheduled to attend consultations that day needs to be downloaded.
- ✧ The service that provides a warning to staff of patients that may be dangerous can be implemented using this approach.
 - The records of possibly patients who may harm themselves or others are identified before the download process. When clinical staff access these records, the software can highlight them to indicate that this is a patient that requires special care.

Risks to confidentiality



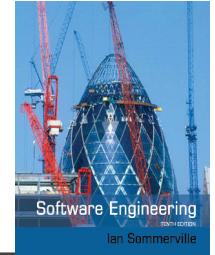
- ✧ To minimize risks to confidentiality that arise from multiple copies of information on laptops:
 - Only download the summary records of patients who are scheduled to attend a clinic. This limits the numbers of records that could be compromised.
 - Encrypt the disk on local client computers. An attacker who does not have the encryption key cannot read the disk if they gain access to the computer.
 - Securely delete the downloaded information at the end of a clinic session. This further reduces the chances of an attacker gaining access to confidential information.
 - Ensure that all network transactions are encrypted. If an attacker intercepts these transactions, they cannot get access to the information.

Key points



- ✧ Resilience is a judgment of how well a system can maintain the continuity of its critical services in the presence of disruptive events.
- ✧ Resilience should be based on the 4 R's model – recognition, resistance, recovery and reinstatement.
- ✧ Resilience planning should be based on the assumption of cyberattacks by malicious insiders and outsiders and that some of these attacks will be successful.
- ✧ Systems should be designed with defensive layers of different types. These layers trap human and technical failures and help resist cyberattacks.

Key points



- ✧ To allow system operators and managers to cope with problems, processes should be flexible and adaptable. Process automation can make it more difficult for people to cope with problems.
- ✧ Business resilience requirements should be the starting point for designing systems for resilience. To achieve system resilience, you have to focus on recognition and recovery from problems, recovery of critical services and assets and reinstatement of the system.
- ✧ An important part of design for resilience is identifying critical services. Systems should be designed so that these services are protected and, in the event of failure, recovered as quickly as possible.



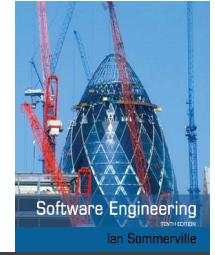
Chapter 15 – Software Reuse

Topics covered



- ✧ The reuse landscape
- ✧ Application frameworks
- ✧ Software product lines
- ✧ Application system reuse

Software reuse



- ✧ In most engineering disciplines, systems are designed by composing existing components that have been used in other systems.
- ✧ Software engineering has been more focused on original development but it is now recognised that to achieve better software, more quickly and at lower cost, we need a design process that is based on systematic software reuse.
- ✧ There has been a major switch to reuse-based development over the past 10 years.

Reuse-based software engineering



✧ System reuse

- Complete systems, which may include several application programs may be reused.

✧ Application reuse

- An application may be reused either by incorporating it without change into other or by developing application families.

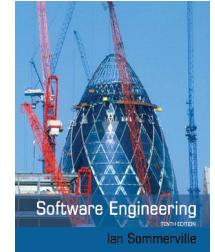
✧ Component reuse

- Components of an application from sub-systems to single objects may be reused.

✧ Object and function reuse

- Small-scale software components that implement a single well-defined object or function may be reused.

Benefits of software reuse



Benefit	Explanation
Accelerated development	Bringing a system to market as early as possible is often more important than overall development costs. Reusing software can speed up system production because both development and validation time may be reduced.
Effective use of specialists	Instead of doing the same work over and over again, application specialists can develop reusable software that encapsulates their knowledge.
Increased dependability	Reused software, which has been tried and tested in working systems, should be more dependable than new software. Its design and implementation faults should have been found and fixed.

Benefits of software reuse



Benefit	Explanation
Lower development costs	Development costs are proportional to the size of the software being developed. Reusing software means that fewer lines of code have to be written.
Reduced process risk	The cost of existing software is already known, whereas the costs of development are always a matter of judgment. This is an important factor for project management because it reduces the margin of error in project cost estimation. This is particularly true when relatively large software components such as subsystems are reused.
Standards compliance	Some standards, such as user interface standards, can be implemented as a set of reusable components. For example, if menus in a user interface are implemented using reusable components, all applications present the same menu formats to users. The use of standard user interfaces improves dependability because users make fewer mistakes when presented with a familiar interface.

Problems with reuse



Problem	Explanation
Creating, maintaining, and using a component library	Populating a reusable component library and ensuring the software developers can use this library can be expensive. Development processes have to be adapted to ensure that the library is used.
Finding, understanding, and adapting reusable components	Software components have to be discovered in a library, understood and, sometimes, adapted to work in a new environment. Engineers must be reasonably confident of finding a component in the library before they include a component search as part of their normal development process.
Increased maintenance costs	If the source code of a reused software system or component is not available then maintenance costs may be higher because the reused elements of the system may become increasingly incompatible with system changes.

Problems with reuse

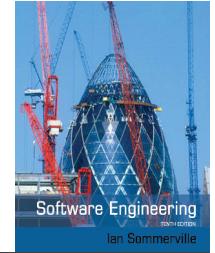


Problem	Explanation
Lack of tool support	Some software tools do not support development with reuse. It may be difficult or impossible to integrate these tools with a component library system. The software process assumed by these tools may not take reuse into account. This is particularly true for tools that support embedded systems engineering, less so for object-oriented development tools.
Not-invented-here syndrome	Some software engineers prefer to rewrite components because they believe they can improve on them. This is partly to do with trust and partly to do with the fact that writing original software is seen as more challenging than reusing other people's software.



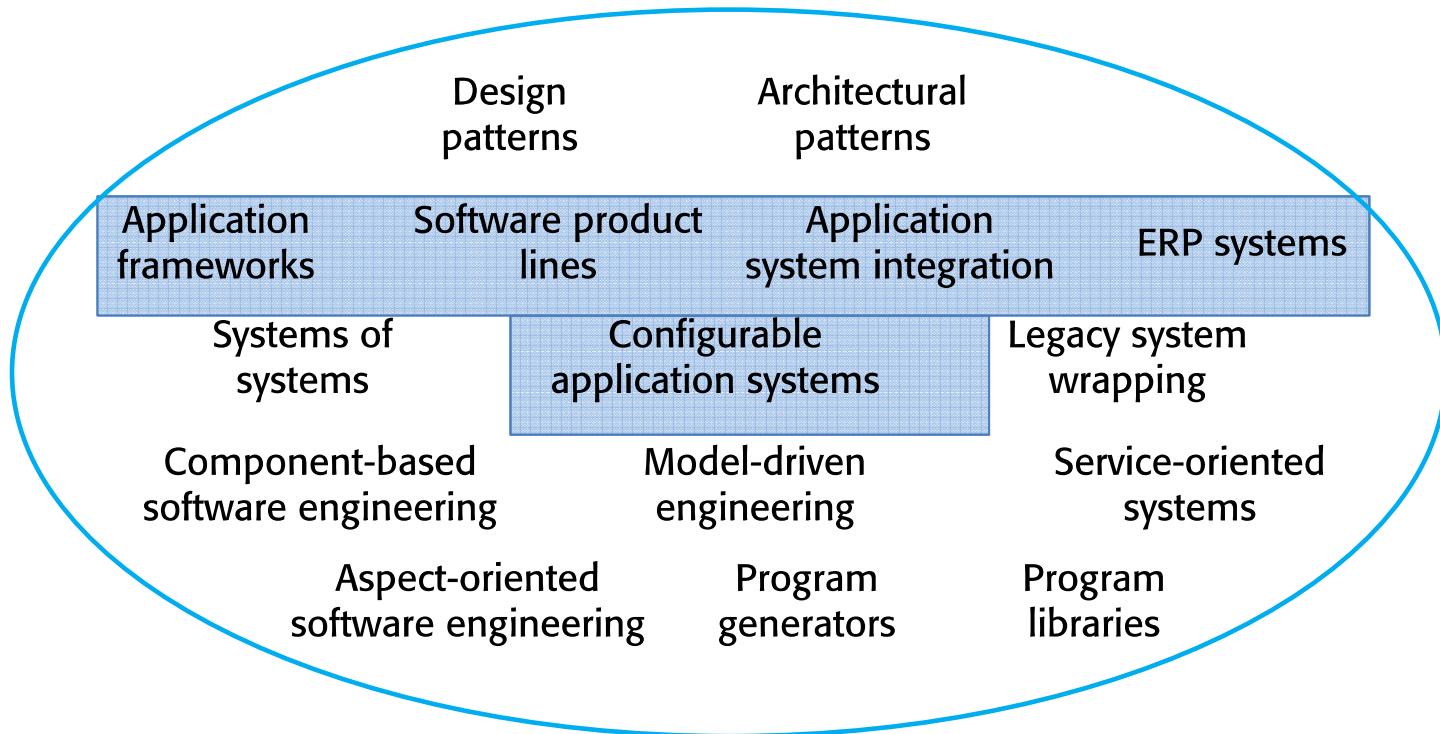
The reuse landscape

The reuse landscape



- ✧ Although reuse is often simply thought of as the reuse of system components, there are many different approaches to reuse that may be used.
- ✧ Reuse is possible at a range of levels from simple functions to complete application systems.
- ✧ The reuse landscape covers the range of possible reuse techniques.

The reuse landscape



Approaches that support software reuse



Approach	Description
Application frameworks	Collections of abstract and concrete classes are adapted and extended to create application systems.
Application system integration	Two or more application systems are integrated to provide extended functionality
Architectural patterns	Standard software architectures that support common types of application system are used as the basis of applications. Described in Chapters 6, 11 and 17.
Aspect-oriented software development	Shared components are woven into an application at different places when the program is compiled. Described in web chapter 31.
Component-based software engineering	Systems are developed by integrating components (collections of objects) that conform to component-model standards. Described in Chapter 16.

Approaches that support software reuse



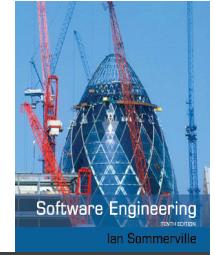
Approach	Description
Configurable application systems	Domain-specific systems are designed so that they can be configured to the needs of specific system customers.
Design patterns	Generic abstractions that occur across applications are represented as design patterns showing abstract and concrete objects and interactions. Described in Chapter 7.
ERP systems	Large-scale systems that encapsulate generic business functionality and rules are configured for an organization.
Legacy system wrapping	Legacy systems (Chapter 9) are ‘wrapped’ by defining a set of interfaces and providing access to these legacy systems through these interfaces.
Model-driven engineering	Software is represented as domain models and implementation independent models and code is generated from these models. Described in Chapter 5.

Approaches that support software reuse



Approach	Description
Program generators	A generator system embeds knowledge of a type of application and is used to generate systems in that domain from a user-supplied system model.
Program libraries	Class and function libraries that implement commonly used abstractions are available for reuse.
Service-oriented systems	Systems are developed by linking shared services, which may be externally provided. Described in Chapter 18.
Software product lines	An application type is generalized around a common architecture so that it can be adapted for different customers.
Systems of systems	Two or more distributed systems are integrated to create a new system. Described in Chapter 20.

Reuse planning factors



- ✧ The development schedule for the software.
- ✧ The expected software lifetime.
- ✧ The background, skills and experience of the development team.
- ✧ The criticality of the software and its non-functional requirements.
- ✧ The application domain.
- ✧ The execution platform for the software.



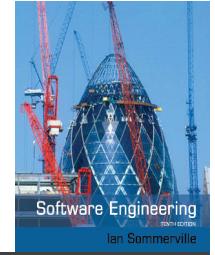
Application frameworks

Framework definition



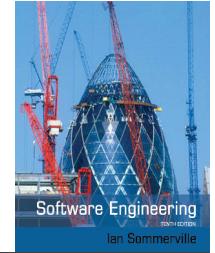
- ✧ “..an integrated set of software artefacts (such as classes, objects and components) that collaborate to provide a reusable architecture for a family of related applications.”

Application frameworks



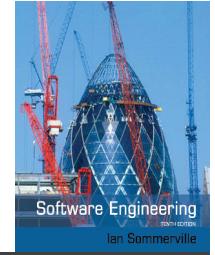
- ✧ Frameworks are moderately large entities that can be reused. They are somewhere between system and component reuse.
- ✧ Frameworks are a sub-system design made up of a collection of abstract and concrete classes and the interfaces between them.
- ✧ The sub-system is implemented by adding components to fill in parts of the design and by instantiating the abstract classes in the framework.

Web application frameworks



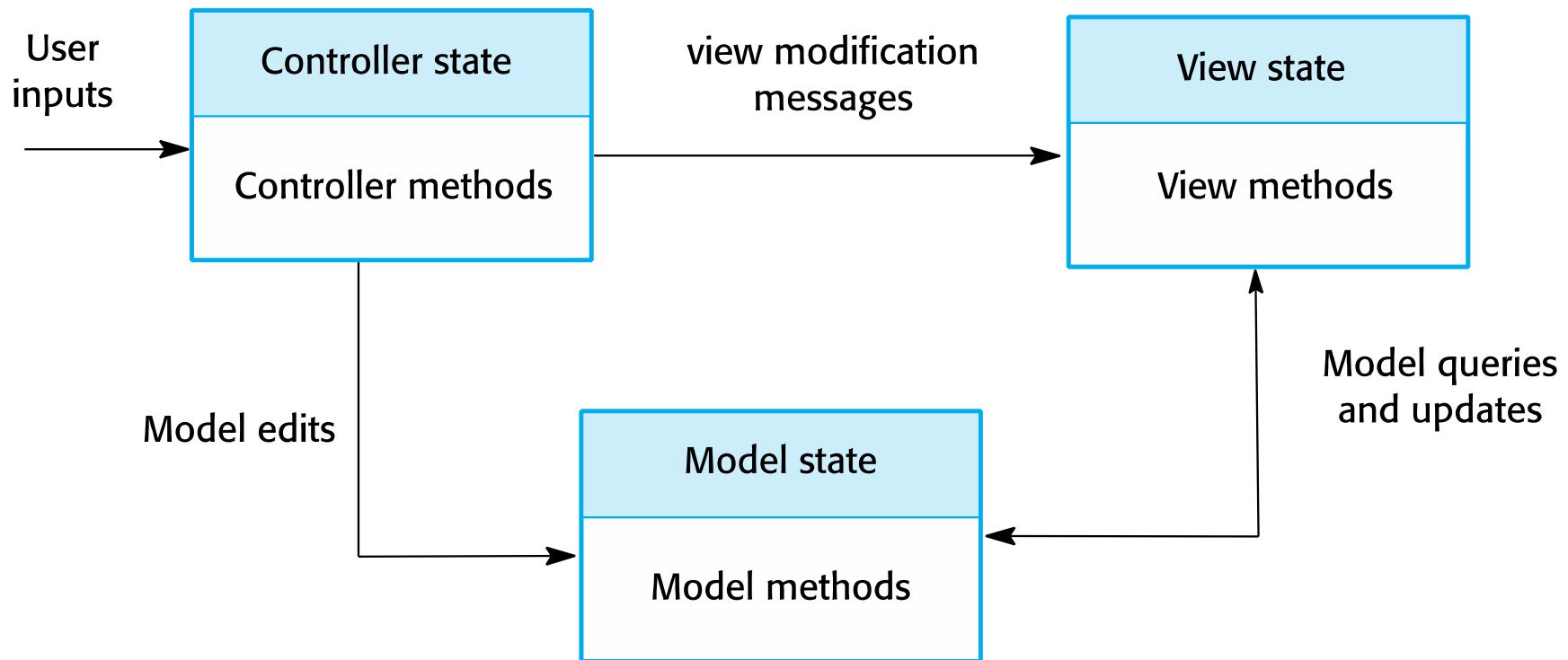
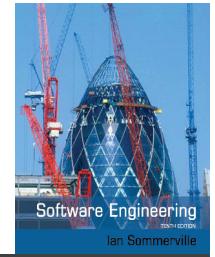
- ✧ Support the construction of dynamic websites as a front-end for web applications.
- ✧ WAFs are now available for all of the commonly used web programming languages e.g. Java, Python, Ruby, etc.
- ✧ Interaction model is based on the Model-View-Controller composite pattern.

Model-view controller

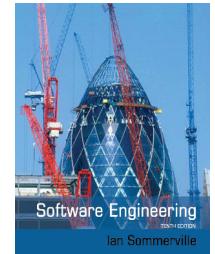


- ✧ System infrastructure framework for GUI design.
- ✧ Allows for multiple presentations of an object and separate interactions with these presentations.
- ✧ MVC framework involves the instantiation of a number of patterns (as discussed in Chapter 7).

The Model-View-Controller pattern



WAF features



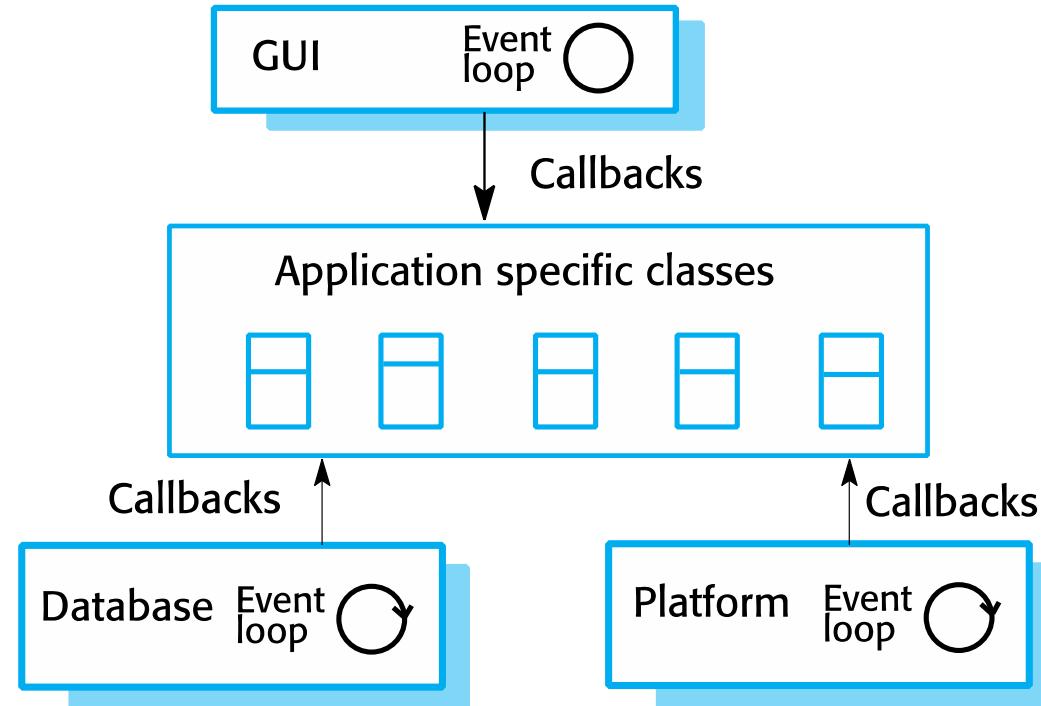
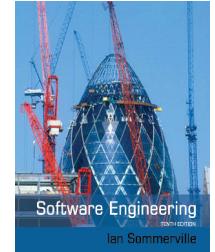
- ✧ *Security*
 - WAFs may include classes to help implement user authentication (login) and access.
- ✧ *Dynamic web pages*
 - Classes are provided to help you define web page templates and to populate these dynamically from the system database.
- ✧ *Database support*
 - The framework may provide classes that provide an abstract interface to different databases.
- ✧ *Session management*
 - Classes to create and manage sessions (a number of interactions with the system by a user) are usually part of a WAF.
- ✧ *User interaction*
 - Most web frameworks now provide AJAX support (Holdener, 2008), which allows more interactive web pages to be created.

Extending frameworks



- ✧ Frameworks are generic and are extended to create a more specific application or sub-system. They provide a skeleton architecture for the system.
- ✧ Extending the framework involves
 - Adding concrete classes that inherit operations from abstract classes in the framework;
 - Adding methods that are called in response to events that are recognised by the framework.
- ✧ Problem with frameworks is their complexity which means that it takes a long time to use them effectively.

Inversion of control in frameworks





Framework classes

✧ System infrastructure frameworks

- Support the development of system infrastructures such as communications, user interfaces and compilers.

✧ Middleware integration frameworks

- Standards and classes that support component communication and information exchange.

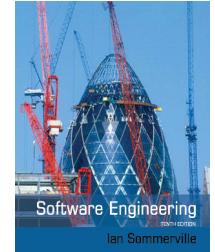
✧ Enterprise application frameworks

- Support the development of specific types of application such as telecommunications or financial systems.



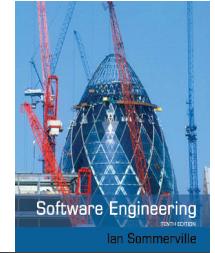
Software product lines

Software product lines



- ✧ Software product lines or application families are applications with generic functionality that can be adapted and configured for use in a specific context.
- ✧ A software product line is a set of applications with a common architecture and shared components, with each application specialized to reflect different requirements.
- ✧ Adaptation may involve:
 - Component and system configuration;
 - Adding new components to the system;
 - Selecting from a library of existing components;
 - Modifying components to meet new requirements.

Base systems for a software product line

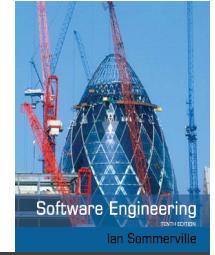


Specialized application components

Configurable application
components

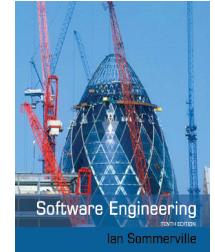
Core
components

Base applications



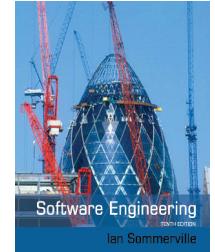
- ✧ Core components that provide infrastructure support.
These are not usually modified when developing a new instance of the product line.
- ✧ Configurable components that may be modified and configured to specialize them to a new application.
Sometimes, it is possible to reconfigure these components without changing their code by using a built-in component configuration language.
- ✧ Specialized, domain-specific components some or all of which may be replaced when a new instance of a product line is created.

Application frameworks and product lines

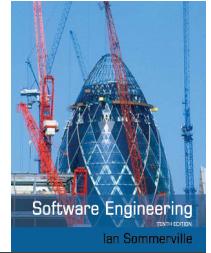


- ✧ Application frameworks rely on object-oriented features such as polymorphism to implement extensions. Product lines need not be object-oriented (e.g. embedded software for a mobile phone)
- ✧ Application frameworks focus on providing technical rather than domain-specific support. Product lines embed domain and platform information.
- ✧ Product lines often control applications for equipment.
- ✧ Software product lines are made up of a family of applications, usually owned by the same organization.

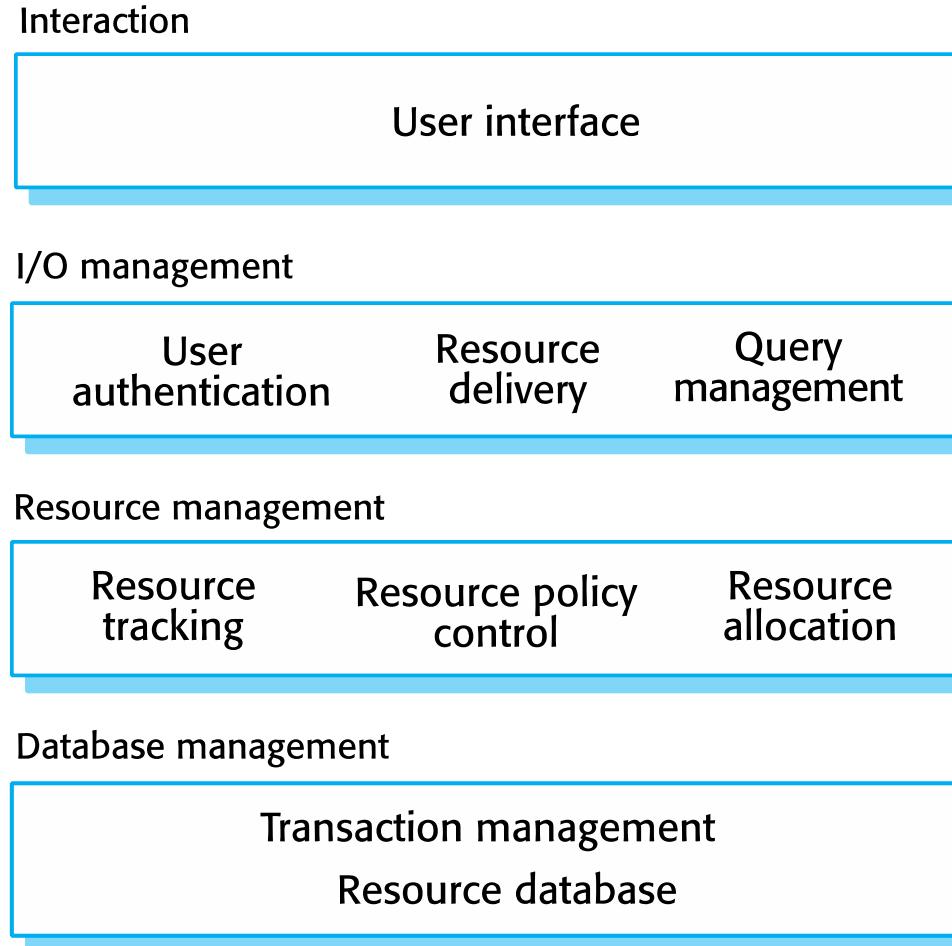
Product line architectures



- ✧ Architectures must be structured in such a way to separate different sub-systems and to allow them to be modified.
- ✧ The architecture should also separate entities and their descriptions and the higher levels in the system access entities through descriptions rather than directly.



The architecture of a resource allocation system



The product line architecture of a vehicle dispatcher



Interaction

Operator interface

Comms system interface

I/O management

Operator authentication

Map and route planner

Report generator

Query manager

Resource management

Vehicle status manager

Incident logger

Vehicle despatcher

Equipment manager

Vehicle locator

Database management

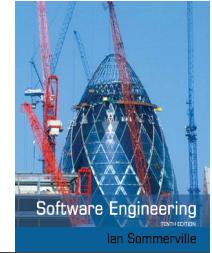
Equipment database

Transaction management

Incident log

Vehicle database

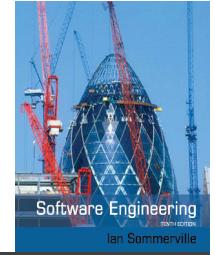
Map database



Vehicle dispatching

- ✧ A specialised resource management system where the aim is to allocate resources (vehicles) to handle incidents.
- ✧ Adaptations include:
 - At the UI level, there are components for operator display and communications;
 - At the I/O management level, there are components that handle authentication, reporting and route planning;
 - At the resource management level, there are components for vehicle location and despatch, managing vehicle status and incident logging;
 - The database includes equipment, vehicle and map databases.

Product line specialisation



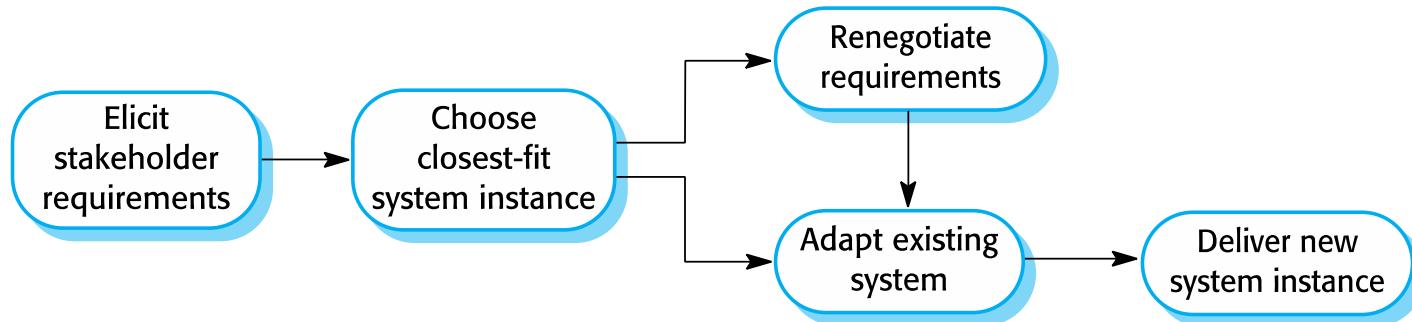
- ✧ Platform specialization
 - Different versions of the application are developed for different platforms.
- ✧ Environment specialization
 - Different versions of the application are created to handle different operating environments e.g. different types of communication equipment.
- ✧ Functional specialization
 - Different versions of the application are created for customers with different requirements.
- ✧ Process specialization
 - Different versions of the application are created to support different business processes.

Product instance development



Software Engineering

Ian Sommerville

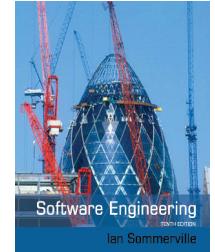


Product instance development



- ✧ Elicit stakeholder requirements
 - Use existing family member as a prototype
- ✧ Choose closest-fit family member
 - Find the family member that best meets the requirements
- ✧ Re-negotiate requirements
 - Adapt requirements as necessary to capabilities of the software
- ✧ Adapt existing system
 - Develop new modules and make changes for family member
- ✧ Deliver new family member
 - Document key features for further member development

Product line configuration



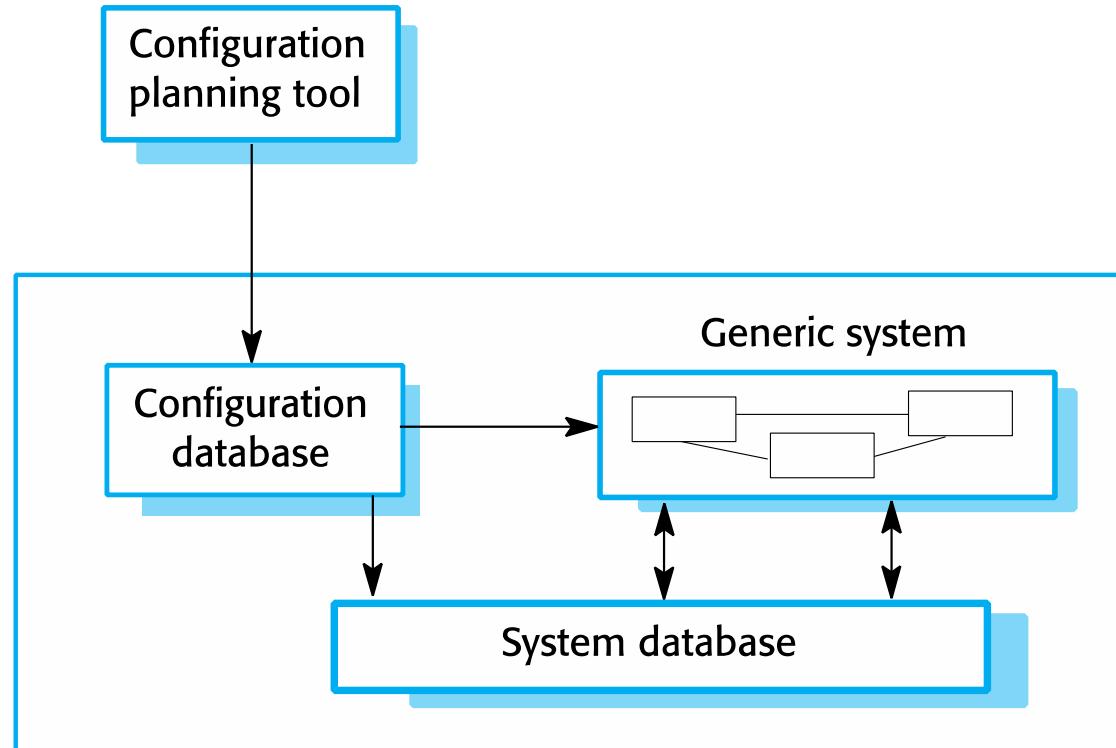
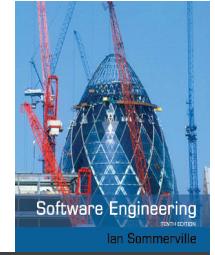
✧ Design time configuration

- The organization that is developing the software modifies a common product line core by developing, selecting or adapting components to create a new system for a customer.

✧ Deployment time configuration

- A generic system is designed for configuration by a customer or consultants working with the customer. Knowledge of the customer's specific requirements and the system's operating environment is embedded in configuration data that are used by the generic system.

Deployment-time configuration



Levels of deployment time configuration

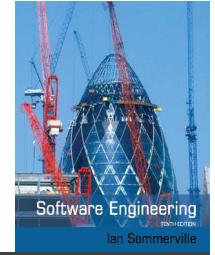


- ✧ Component selection, where you select the modules in a system that provide the required functionality.
- ✧ Workflow and rule definition, where you define workflows (how information is processed, stage-by-stage) and validation rules that should apply to information entered by users or generated by the system.
- ✧ Parameter definition, where you specify the values of specific system parameters that reflect the instance of the application that you are creating



Application system reuse

Application system reuse



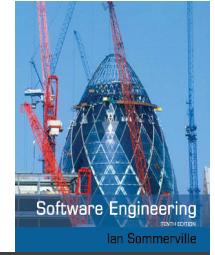
- ✧ An application system product is a software system that can be adapted for different customers without changing the source code of the system.
- ✧ Application systems have generic features and so can be used/reused in different environments.
- ✧ Application system products are adapted by using built-in configuration mechanisms that allow the functionality of the system to be tailored to specific customer needs.
 - For example, in a hospital patient record system, separate input forms and output reports might be defined for different types of patient.

Benefits of application system reuse



- ✧ As with other types of reuse, more rapid deployment of a reliable system may be possible.
- ✧ It is possible to see what functionality is provided by the applications and so it is easier to judge whether or not they are likely to be suitable.
- ✧ Some development risks are avoided by using existing software. However, this approach has its own risks, as I discuss below.
- ✧ Businesses can focus on their core activity without having to devote a lot of resources to IT systems development.
- ✧ As operating platforms evolve, technology updates may be simplified as these are the responsibility of the COTS product vendor rather than the customer.

Problems of application system reuse



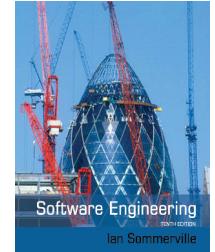
- ✧ Requirements usually have to be adapted to reflect the functionality and mode of operation of the COTS product.
- ✧ The COTS product may be based on assumptions that are practically impossible to change.
- ✧ Choosing the right COTS system for an enterprise can be a difficult process, especially as many COTS products are not well documented.
- ✧ There may be a lack of local expertise to support systems development.
- ✧ The COTS product vendor controls system support and evolution.

Configurable application systems



- ✧ Configurable application systems are generic application systems that may be designed to support a particular business type, business activity or, sometimes, a complete business enterprise.
 - For example, an application system may be produced for dentists that handles appointments, dental records, patient recall, etc.
- ✧ Domain-specific systems, such as systems to support a business function (e.g. document management) provide functionality that is likely to be required by a range of potential users.

COTS-solution and COTS-integrated systems

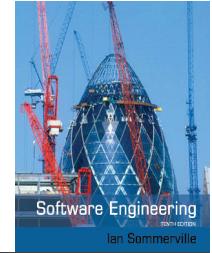


Configurable application systems	Application system integration
Single product that provides the functionality required by a customer	Several heterogeneous system products are integrated to provide customized functionality
Based around a generic solution and standardized processes	Flexible solutions may be developed for customer processes
Development focus is on system configuration	Development focus is on system integration
System vendor is responsible for maintenance	System owner is responsible for maintenance
System vendor provides the platform for the system	System owner provides the platform for the system

ERP systems

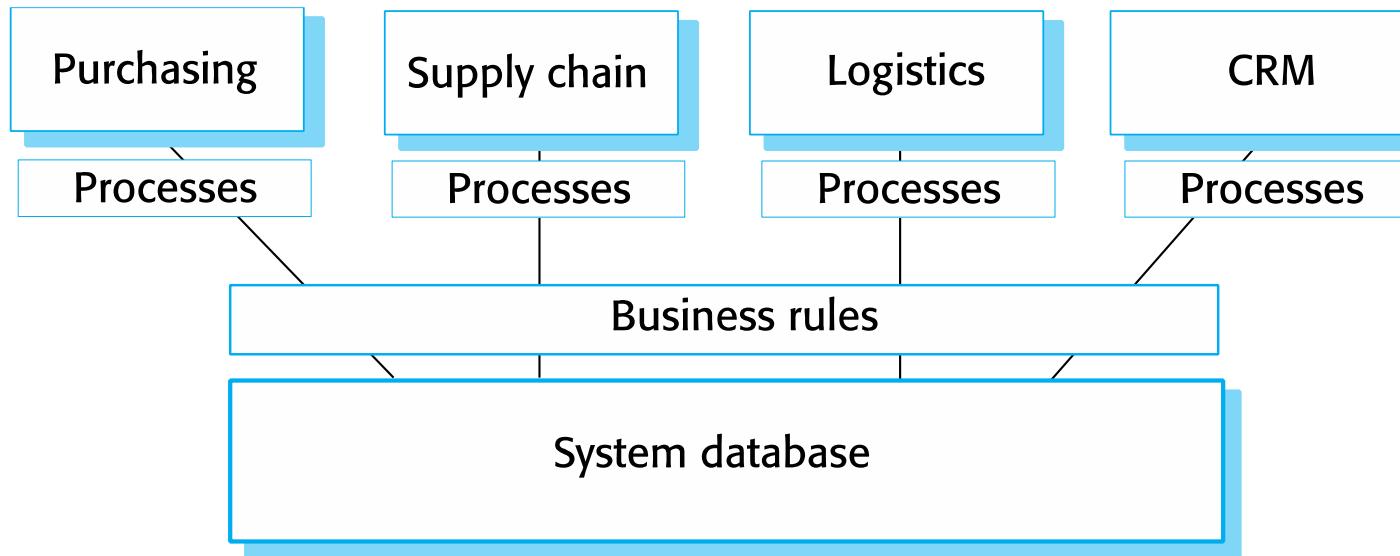


- ✧ An Enterprise Resource Planning (ERP) system is a generic system that supports common business processes such as ordering and invoicing, manufacturing, etc.
- ✧ These are very widely used in large companies - they represent probably the most common form of software reuse.
- ✧ The generic core is adapted by including modules and by incorporating knowledge of business processes and rules.

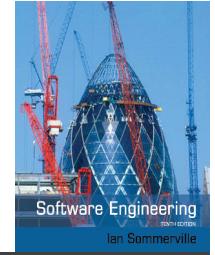


The architecture of an ERP system

Software Engineering
Ian Sommerville

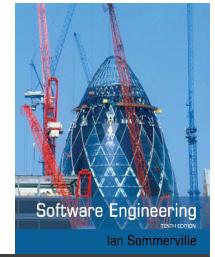


ERP architecture



- ✧ A number of modules to support different business functions.
- ✧ A defined set of business processes, associated with each module, which relate to activities in that module.
- ✧ A common database that maintains information about all related business functions.
- ✧ A set of business rules that apply to all data in the database.

ERP configuration



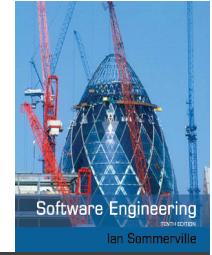
- ✧ Selecting the required functionality from the system.
- ✧ Establishing a data model that defines how the organization's data will be structured in the system database.
- ✧ Defining business rules that apply to that data.
- ✧ Defining the expected interactions with external systems.
- ✧ Designing the input forms and the output reports generated by the system.
- ✧ Designing new business processes that conform to the underlying process model supported by the system.
- ✧ Setting parameters that define how the system is deployed on its underlying platform.

Integrated application systems



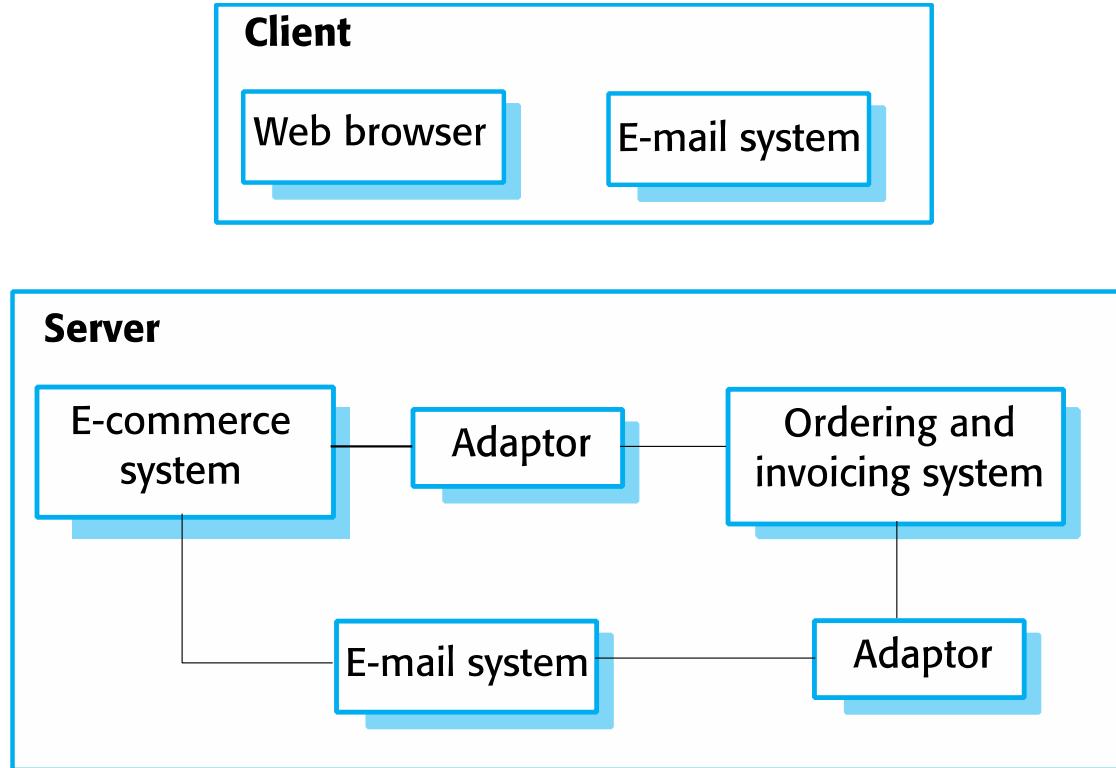
- ✧ Integrated application systems are applications that include two or more application system products and/or legacy application systems.
- ✧ You may use this approach when there is no single application system that meets all of your needs or when you wish to integrate a new application system with systems that you already use.

Design choices

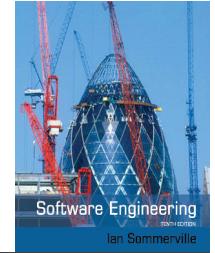


- ✧ Which individual application systems offer the most appropriate functionality?
 - Typically, there will be several application system products available, which can be combined in different ways.
- ✧ How will data be exchanged?
 - Different products normally use unique data structures and formats. You have to write adaptors that convert from one representation to another.
- ✧ What features of a product will actually be used?
 - Individual application systems may include more functionality than you need and functionality may be duplicated across different products.

An integrated procurement system

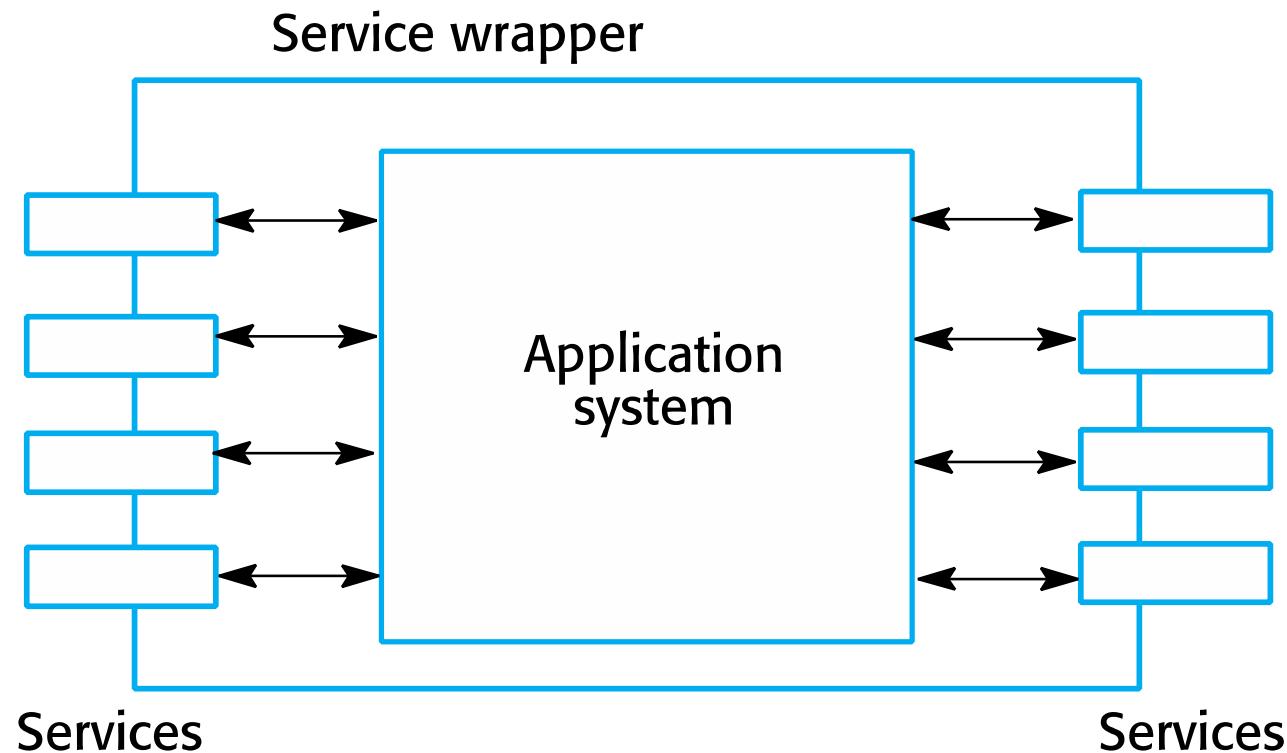


Service-oriented interfaces



- ✧ Application system integration can be simplified if a service-oriented approach is used.
- ✧ A service-oriented approach means allowing access to the application system's functionality through a standard service interface, with a service for each discrete unit of functionality.
- ✧ Some applications may offer a service interface but, sometimes, this service interface has to be implemented by the system integrator. You have to program a wrapper that hides the application and provides externally visible services.

Application wrapping

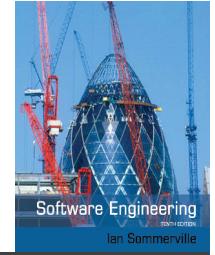


Application system integration problems

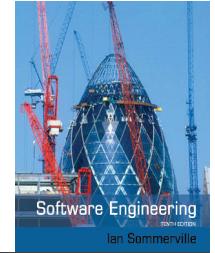


- ✧ Lack of control over functionality and performance
 - Application systems may be less effective than they appear
- ✧ Problems with application system inter-operability
 - Different application systems may make different assumptions that means integration is difficult
- ✧ No control over system evolution
 - Application system vendors not system users control evolution
- ✧ Support from system vendors
 - Application system vendors may not offer support over the lifetime of the product

Key points



- ✧ There are many different ways to reuse software. These range from the reuse of classes and methods in libraries to the reuse of complete application systems.
- ✧ The advantages of software reuse are lower costs, faster software development and lower risks. System dependability is increased. Specialists can be used more effectively by concentrating their expertise on the design of reusable components.
- ✧ Application frameworks are collections of concrete and abstract objects that are designed for reuse through specialization and the addition of new objects. They usually incorporate good design practice through design patterns.



Software Engineering
Ian Sommerville

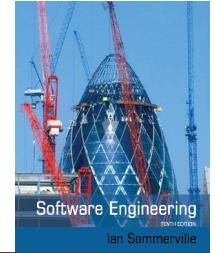
Key points

- ✧ Software product lines are related applications that are developed from one or more base applications. A generic system is adapted and specialized to meet specific requirements for functionality, target platform or operational configuration.
- ✧ Application system reuse is concerned with the reuse of large-scale, off-the-shelf systems. These provide a lot of functionality and their reuse can radically reduce costs and development time. Systems may be developed by configuring a single, generic application system or by integrating two or more application systems.
- ✧ Potential problems with application system reuse include lack of control over functionality and performance, lack of control over system evolution, the need for support from external vendors and difficulties in ensuring that systems can inter-operate.



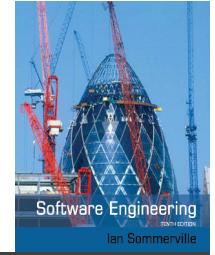
Chapter 16 - Component-based software engineering

Topics covered



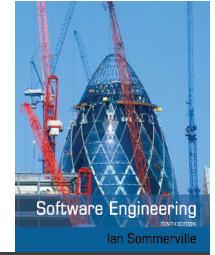
- ✧ Components and component models
- ✧ CBSE processes
- ✧ Component composition

Component-based development



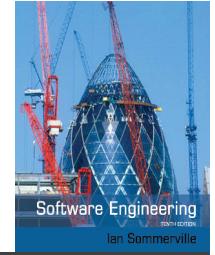
- ✧ Component-based software engineering (CBSE) is an approach to software development that relies on the reuse of entities called ‘software components’.
- ✧ It emerged from the failure of object-oriented development to support effective reuse. Single object classes are too detailed and specific.
- ✧ Components are more abstract than object classes and can be considered to be stand-alone service providers. They can exist as stand-alone entities.

CBSE essentials



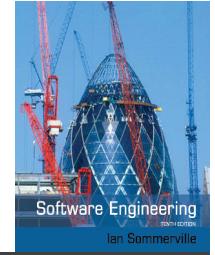
- ✧ **Independent components** specified by their interfaces.
- ✧ **Component standards** to facilitate component integration.
- ✧ **Middleware** that provides support for component interoperability.
- ✧ **A development process** that is geared to reuse.

CBSE and design principles



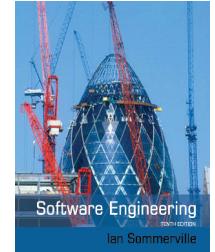
- ✧ Apart from the benefits of reuse, CBSE is based on sound software engineering design principles:
 - Components are independent so do not interfere with each other;
 - Component implementations are hidden;
 - Communication is through well-defined interfaces;
 - One components can be replaced by another if its interface is maintained;
 - Component infrastructures offer a range of standard services.

Component standards



- ✧ Standards need to be established so that components can communicate with each other and inter-operate.
- ✧ Unfortunately, several competing component standards were established:
 - Sun's Enterprise Java Beans
 - Microsoft's COM and .NET
 - CORBA's CCM
- ✧ In practice, these multiple standards have hindered the uptake of CBSE. It is impossible for components developed using different approaches to work together.

Service-oriented software engineering

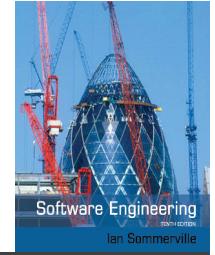


- ✧ An executable service is a type of independent component. It has a ‘provides’ interface but not a ‘requires’ interface.
- ✧ From the outset, services have been based around standards so there are no problems in communicating between services offered by different vendors.
- ✧ System performance may be slower with services but this approach is replacing CBSE in many systems.
- ✧ Covered in Chapter 18



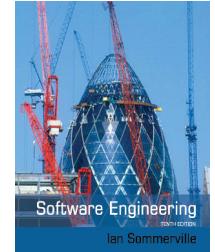
Components and component models

Components



- ❖ Components provide a service without regard to where the component is executing or its programming language
 - A component is an independent executable entity that can be made up of one or more executable objects;
 - The component interface is published and all interactions are through the published interface;

Component definitions



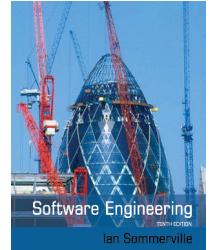
✧ Councill and Heinmann:

- *A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.*

✧ Szyperski:

- *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third-parties.*

Component characteristics



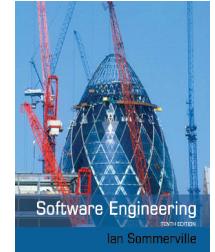
Component characteristic	Description
Composable	For a component to be composable, all external interactions must take place through publicly defined interfaces. In addition, it must provide external access to information about itself, such as its methods and attributes.
Deployable	To be deployable, a component has to be self-contained. It must be able to operate as a stand-alone entity on a component platform that provides an implementation of the component model. This usually means that the component is binary and does not have to be compiled before it is deployed. If a component is implemented as a service, it does not have to be deployed by a user of a component. Rather, it is deployed by the service provider.

Component characteristics



Component characteristic	Description
Documented	Components have to be fully documented so that potential users can decide whether or not the components meet their needs. The syntax and, ideally, the semantics of all component interfaces should be specified.
Independent	A component should be independent—it should be possible to compose and deploy it without having to use other specific components. In situations where the component needs externally provided services, these should be explicitly set out in a 'requires' interface specification.
Standardized	Component standardization means that a component used in a CBSE process has to conform to a standard component model. This model may define component interfaces, component metadata, documentation, composition, and deployment.

Component as a service provider



- ✧ The component is an independent, executable entity. It does not have to be compiled before it is used with other components.
- ✧ The services offered by a component are made available through an interface and all component interactions take place through that interface.
- ✧ The component interface is expressed in terms of parameterized operations and its internal state is never exposed.



Component interfaces

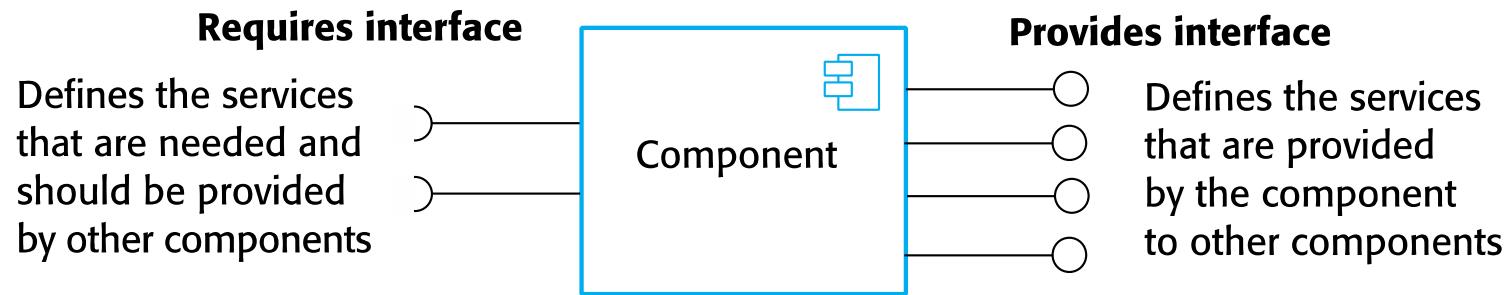
✧ Provides interface

- Defines the services that are provided by the component to other components.
- This interface, essentially, is the component API. It defines the methods that can be called by a user of the component.

✧ Requires interface

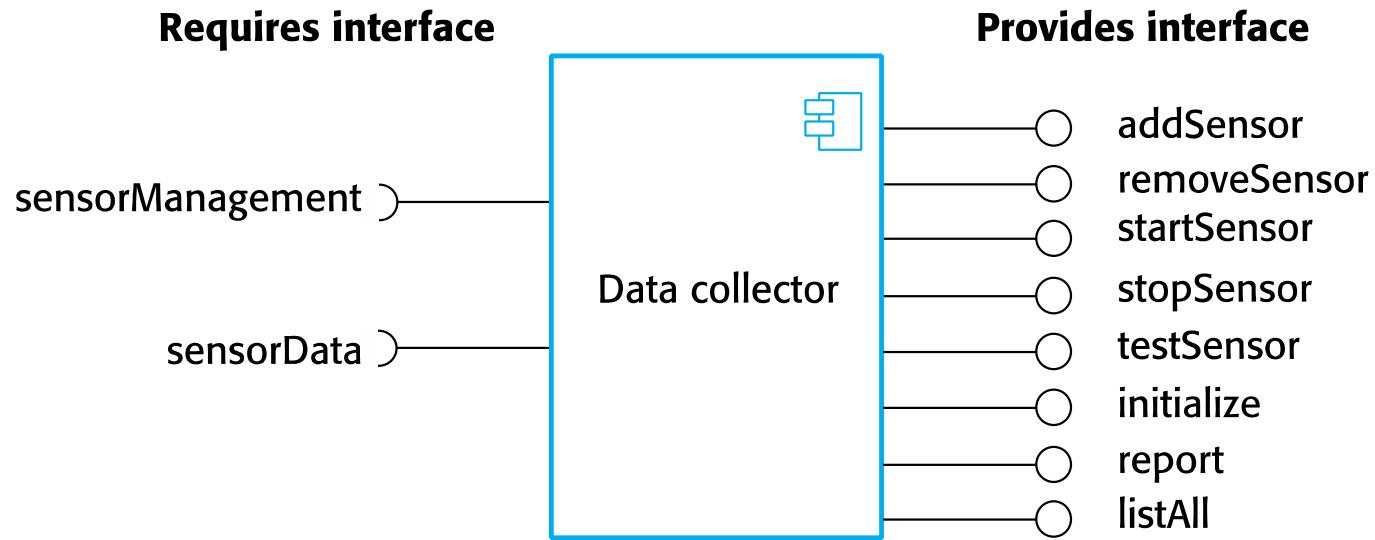
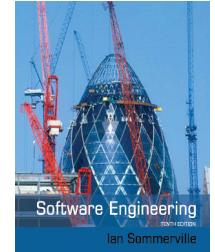
- Defines the services that specifies what services must be made available for the component to execute as specified.
- This does not compromise the independence or deployability of a component because the 'requires' interface does not define how these services should be provided.

Component interfaces

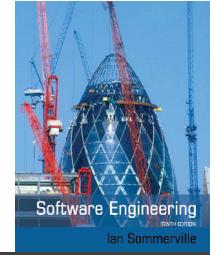


Note UML notation. Ball and sockets can fit together.

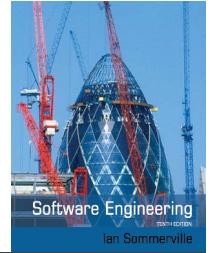
A model of a data collector component



Component access



- ✧ Components are accessed using remote procedure calls (RPCs).
- ✧ Each component has a unique identifier (usually a URL) and can be referenced from any networked computer.
- ✧ Therefore it can be called in a similar way as a procedure or method running on a local computer.



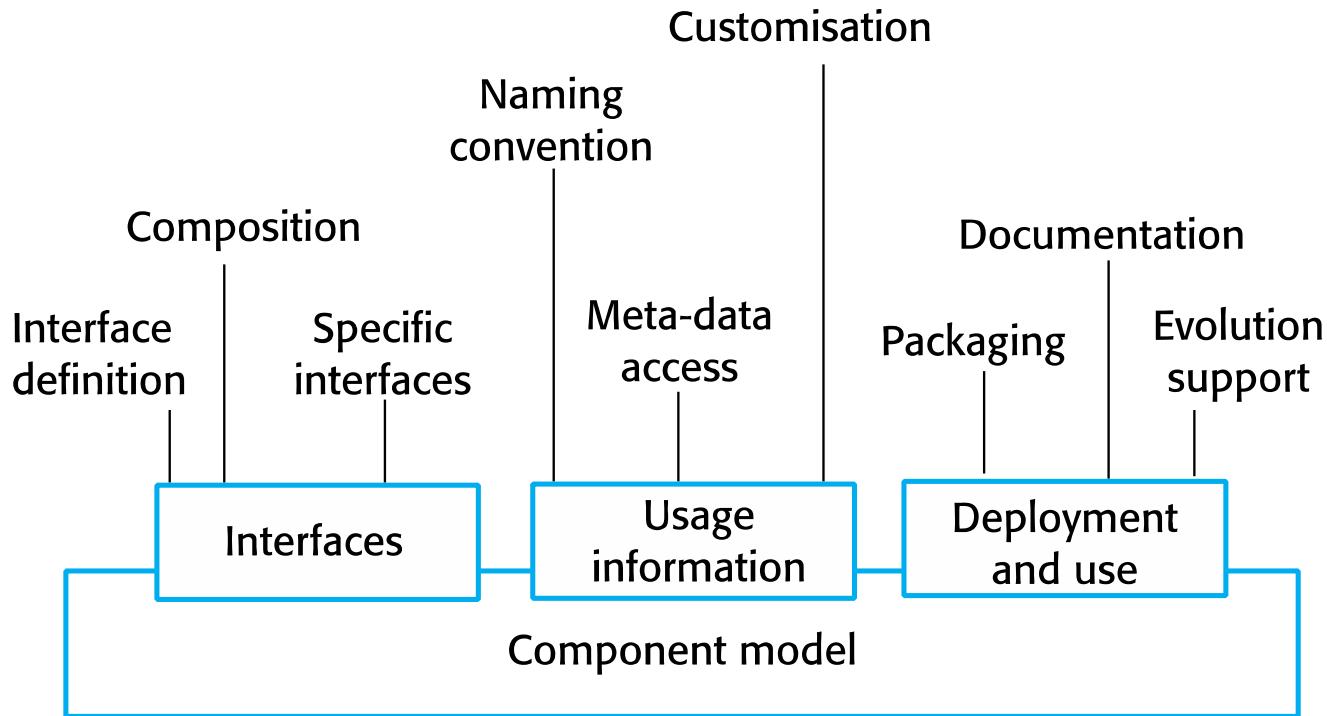
Component models

- ✧ A component model is a definition of standards for component implementation, documentation and deployment.
- ✧ Examples of component models
 - EJB model (Enterprise Java Beans)
 - COM+ model (.NET model)
 - Corba Component Model
- ✧ The component model specifies how interfaces should be defined and the elements that should be included in an interface definition.

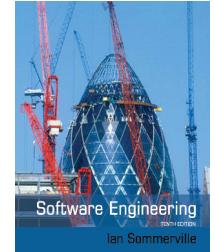


Basic elements of a component model

Software Engineering
Ian Sommerville



Elements of a component model



✧ Interfaces

- Components are defined by specifying their interfaces. The component model specifies how the interfaces should be defined and the elements, such as operation names, parameters and exceptions, which should be included in the interface definition.

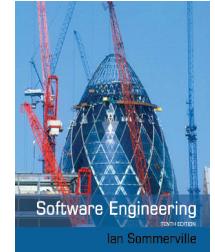
✧ Usage

- In order for components to be distributed and accessed remotely, they need to have a unique name or handle associated with them. This has to be globally unique.

✧ Deployment

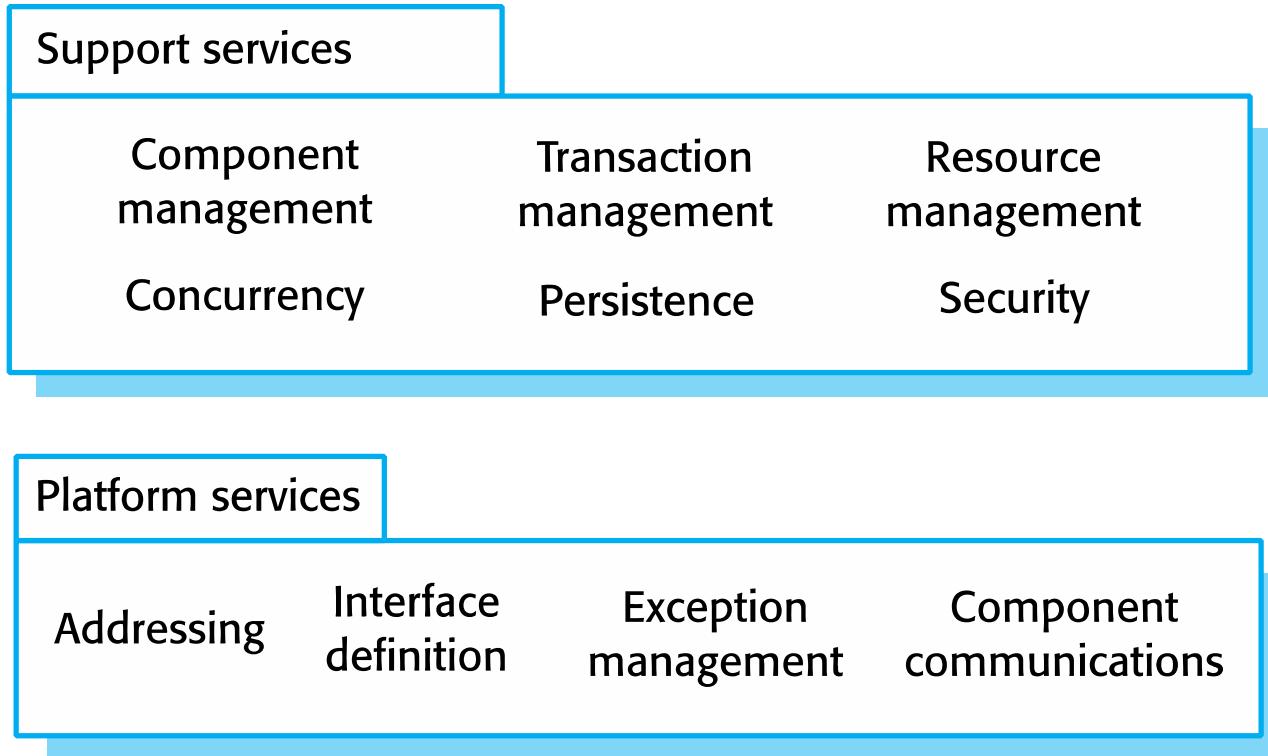
- The component model includes a specification of how components should be packaged for deployment as independent, executable entities.

Middleware support



- ✧ Component models are the basis for middleware that provides support for executing components.
- ✧ Component model implementations provide:
 - Platform services that allow components written according to the model to communicate;
 - Support services that are application-independent services used by different components.
- ✧ To use services provided by a model, components are deployed in a **container**. This is a set of interfaces used to access the service implementations.

Middleware services defined in a component model



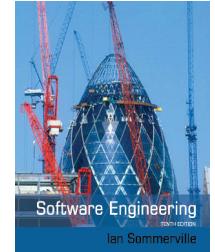


Software Engineering

Ian Sommerville

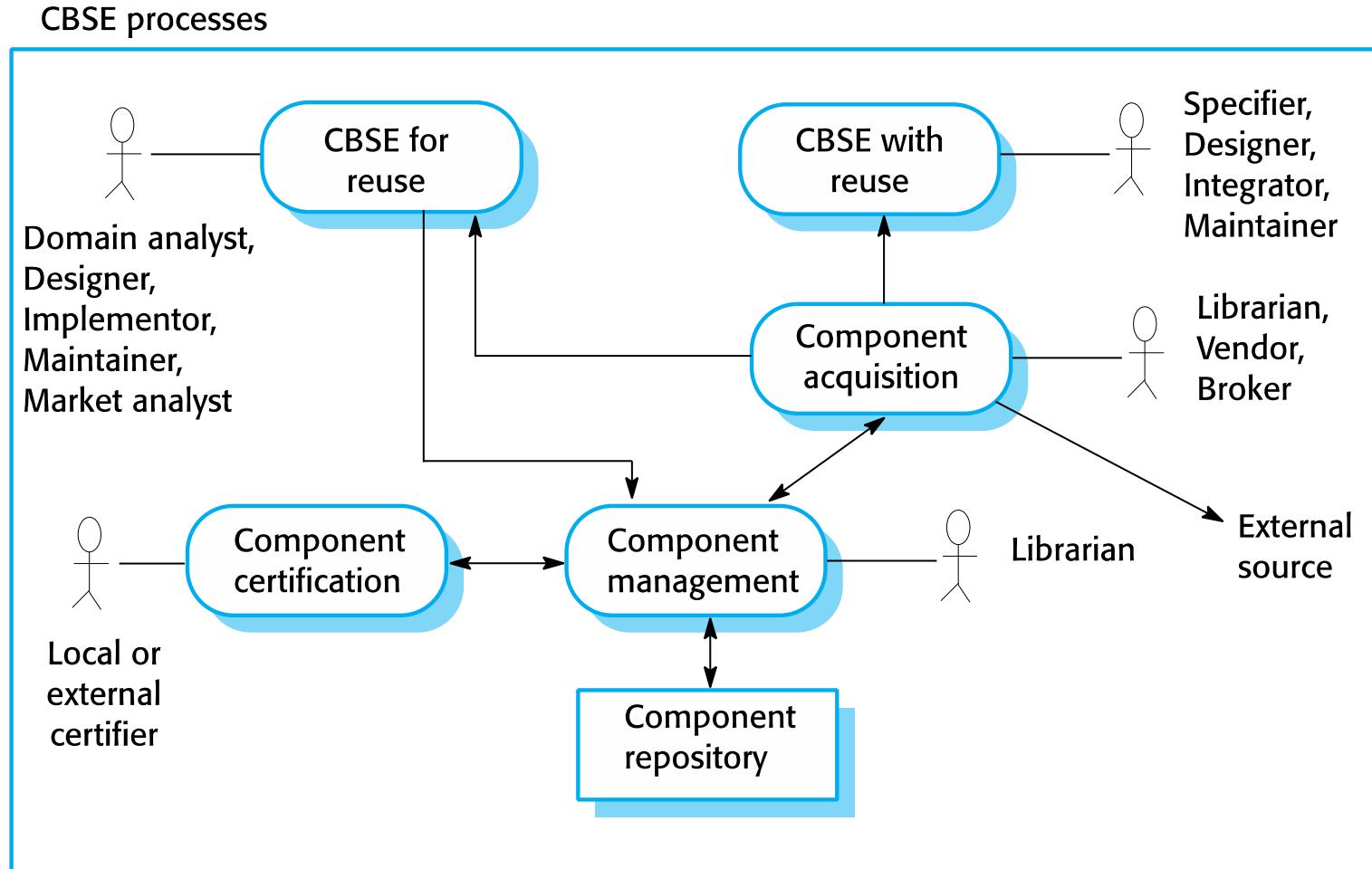
CBSE processes

CBSE processes

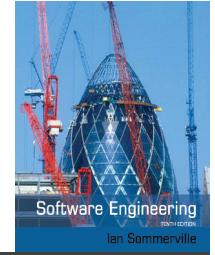


- ✧ CBSE processes are software processes that support component-based software engineering.
 - They take into account the possibilities of reuse and the different process activities involved in developing and using reusable components.
- ✧ Development for reuse
 - This process is concerned with developing components or services that will be reused in other applications. It usually involves generalizing existing components.
- ✧ Development with reuse
 - This process is the process of developing new applications using existing components and services.

CBSE processes

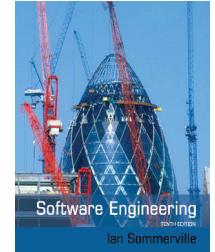


Supporting processes



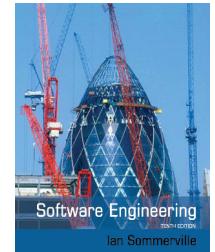
- ✧ Component acquisition is the process of acquiring components for reuse or development into a reusable component.
 - It may involve accessing locally- developed components or services or finding these components from an external source.
- ✧ Component management is concerned with managing a company's reusable components, ensuring that they are properly catalogued, stored and made available for reuse.
- ✧ Component certification is the process of checking a component and certifying that it meets its specification.

CBSE for reuse



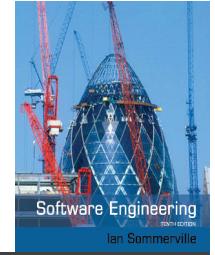
- ✧ CBSE for reuse focuses on component development.
- ✧ Components developed for a specific application usually have to be generalised to make them reusable.
- ✧ A component is most likely to be reusable if it associated with a stable domain abstraction (business object).
- ✧ For example, in a hospital stable domain abstractions are associated with the fundamental purpose - nurses, patients, treatments, etc.

Component development for reuse



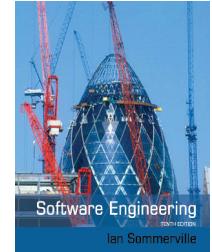
- ✧ Components for reuse may be specially constructed by generalising existing components.
- ✧ Component reusability
 - Should reflect stable domain abstractions;
 - Should hide state representation;
 - Should be as independent as possible;
 - Should publish exceptions through the component interface.
- ✧ There is a trade-off between reusability and usability
 - The more general the interface, the greater the reusability but it is then more complex and hence less usable.

Changes for reusability



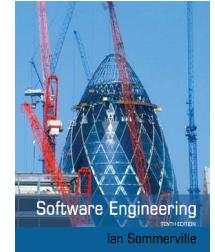
- ✧ Remove application-specific methods.
- ✧ Change names to make them general.
- ✧ Add methods to broaden coverage.
- ✧ Make exception handling consistent.
- ✧ Add a configuration interface for component adaptation.
- ✧ Integrate required components to reduce dependencies.

Exception handling



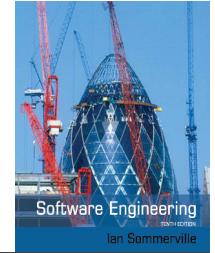
- ✧ Components should not handle exceptions themselves, because each application will have its own requirements for exception handling.
 - Rather, the component should define what exceptions can arise and should publish these as part of the interface.
- ✧ In practice, however, there are two problems with this:
 - Publishing all exceptions leads to bloated interfaces that are harder to understand. This may put off potential users of the component.
 - The operation of the component may depend on local exception handling, and changing this may have serious implications for the functionality of the component.

Legacy system components



- ✧ Existing legacy systems that fulfil a useful business function can be re-packaged as components for reuse.
- ✧ This involves writing a wrapper component that implements provides and requires interfaces then accesses the legacy system.
- ✧ Although costly, this can be much less expensive than rewriting the legacy system.

Reusable components



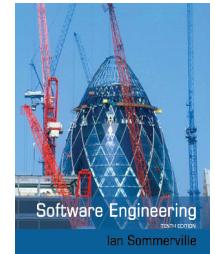
- ✧ The development cost of reusable components may be higher than the cost of specific equivalents. This extra reusability enhancement cost should be an organization rather than a project cost.
- ✧ Generic components may be less space-efficient and may have longer execution times than their specific equivalents.

Component management



- ✧ Component management involves deciding how to classify the component so that it can be discovered, making the component available either in a repository or as a service, maintaining information about the use of the component and keeping track of different component versions.
- ✧ A company with a reuse program may carry out some form of component certification before the component is made available for reuse.
 - Certification means that someone apart from the developer checks the quality of the component.

CBSE with reuse

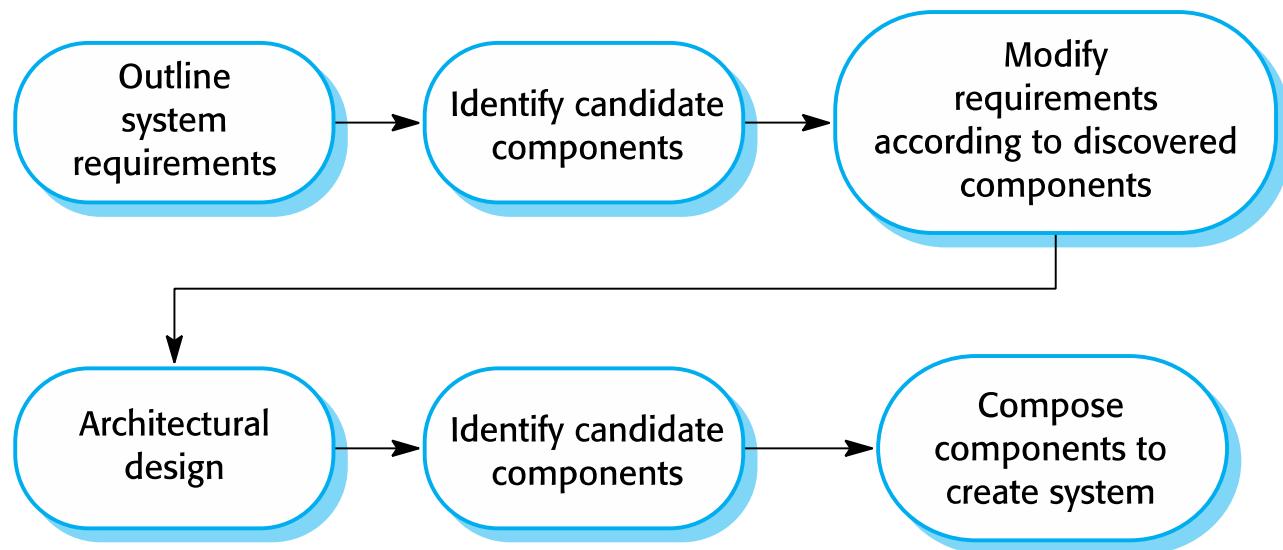


- ✧ CBSE with reuse process has to find and integrate reusable components.
- ✧ When reusing components, it is essential to make trade-offs between ideal requirements and the services actually provided by available components.
- ✧ This involves:
 - Developing outline requirements;
 - Searching for components then modifying requirements according to available functionality.
 - Searching again to find if there are better components that meet the revised requirements.
 - Composing components to create the system.

CBSE with reuse



Software Engineering
Ian Sommerville



The component identification process



Component identification issues



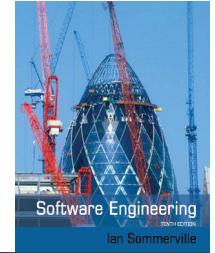
- ❖ **Trust.** You need to be able to trust the supplier of a component. At best, an untrusted component may not operate as advertised; at worst, it can breach your security.
- ❖ **Requirements.** Different groups of components will satisfy different requirements.
- ❖ **Validation.**
 - The component specification may not be detailed enough to allow comprehensive tests to be developed.
 - Components may have unwanted functionality. How can you test this will not interfere with your application?

Component validation



- ✧ Component validation involves developing a set of test cases for a component (or, possibly, extending test cases supplied with that component) and developing a test harness to run component tests.
 - The major problem with component validation is that the component specification may not be sufficiently detailed to allow you to develop a complete set of component tests.
- ✧ As well as testing that a component for reuse does what you require, you may also have to check that the component does not include any malicious code or functionality that you don't need.

Ariane launcher failure – validation failure?

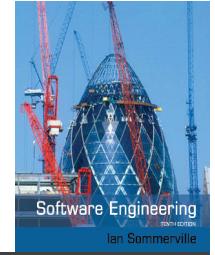


- ✧ In 1996, the 1st test flight of the Ariane 5 rocket ended in disaster when the launcher went out of control 37 seconds after take off.
- ✧ The problem was due to a reused component from a previous version of the launcher (the Inertial Navigation System) that failed because assumptions made when that component was developed did not hold for Ariane 5.
- ✧ The functionality that failed in this component was not required in Ariane 5.

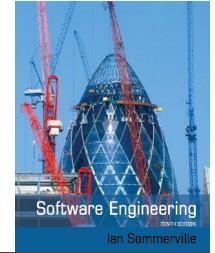


Component composition

Component composition



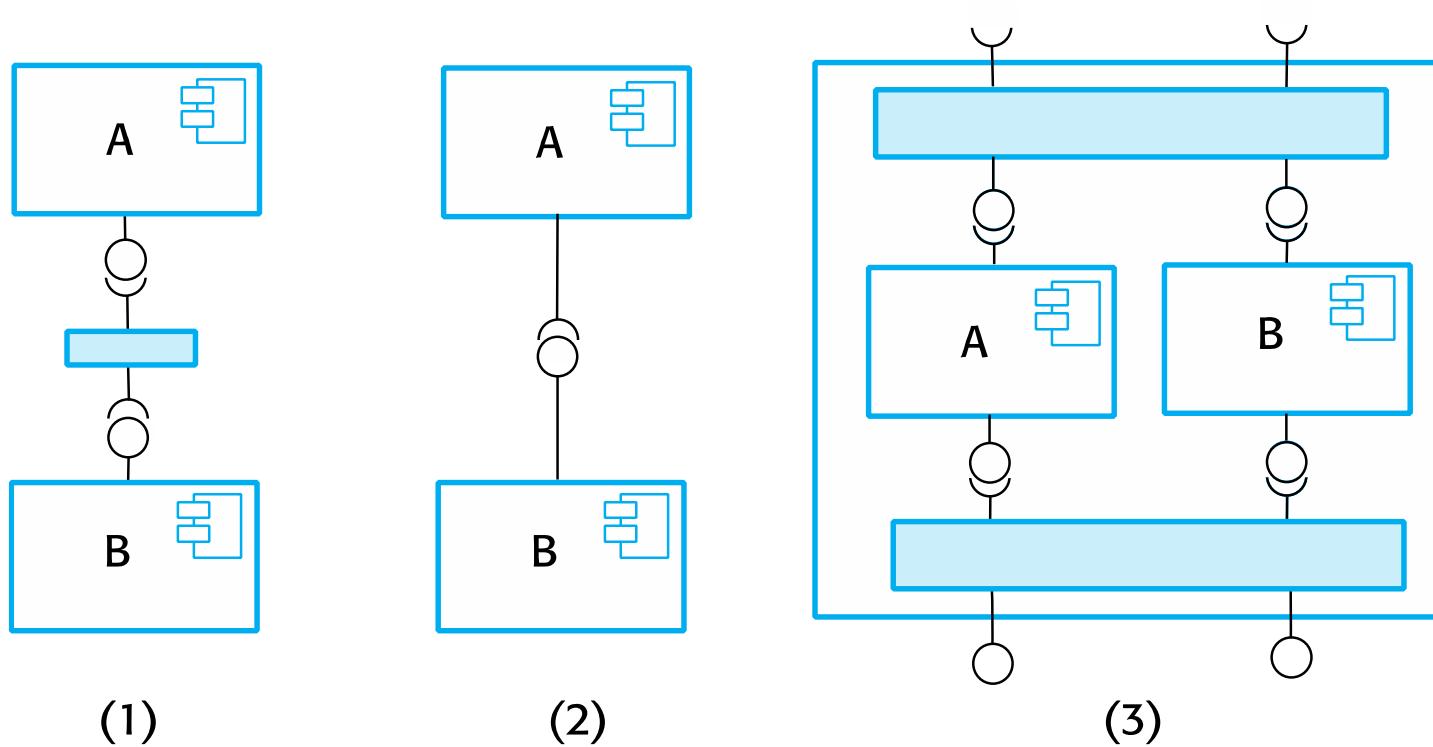
- ✧ The process of assembling components to create a system.
- ✧ Composition involves integrating components with each other and with the component infrastructure.
- ✧ Normally you have to write ‘glue code’ to integrate components.



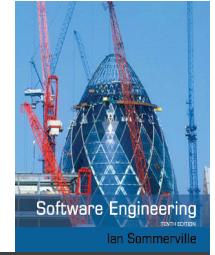
Types of composition

- ✧ **Sequential composition** (1) where the composed components are executed in sequence. This involves composing the provides interfaces of each component.
- ✧ **Hierarchical composition** (2) where one component calls on the services of another. The provides interface of one component is composed with the requires interface of another.
- ✧ **Additive composition** (3) where the interfaces of two components are put together to create a new component. Provides and requires interfaces of integrated component is a combination of interfaces of constituent components.

Types of component composition

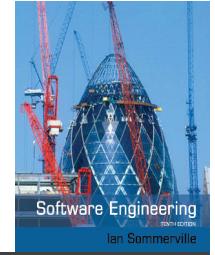


Glue code



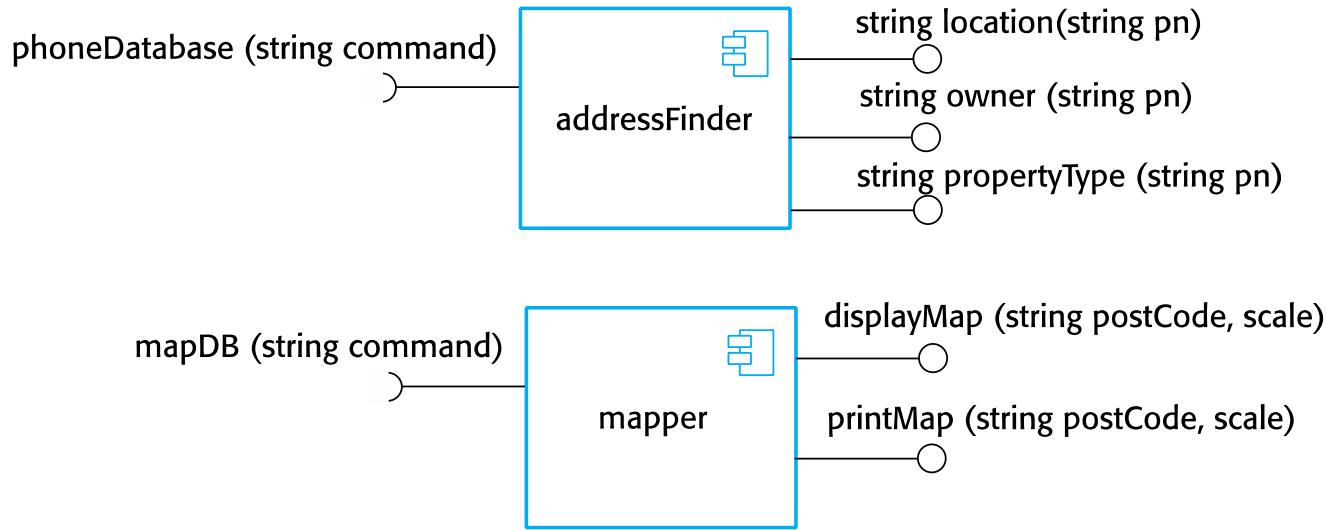
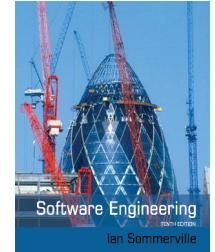
- ✧ Code that allows components to work together
- ✧ If A and B are composed sequentially, then glue code has to call A, collect its results then call B using these results, transforming them into the format required by B.
- ✧ Glue code may be used to resolve interface incompatibilities.

Interface incompatibility

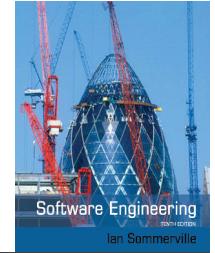


- ✧ **Parameter incompatibility** where operations have the same name but are of different types.
- ✧ **Operation incompatibility** where the names of operations in the composed interfaces are different.
- ✧ **Operation incompleteness** where the provides interface of one component is a subset of the requires interface of another.

Components with incompatible interfaces

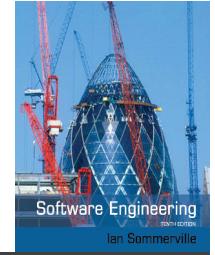


Adaptor components



- ✧ Address the problem of component incompatibility by reconciling the interfaces of the components that are composed.
- ✧ Different types of adaptor are required depending on the type of composition.
- ✧ An addressFinder and a mapper component may be composed through an adaptor that strips the postal code from an address and passes this to the mapper component.

Composition through an adaptor



- ✧ The component `postCodeStripper` is the adaptor that facilitates the sequential composition of `addressFinder` and `mapper` components.

```
address = addressFinder.location (phonenumber) ;  
postCode = postCodeStripper.getPostCode (address) ;  
mapper.displayMap(postCode, 10000)
```

An adaptor linking a data collector and a sensor

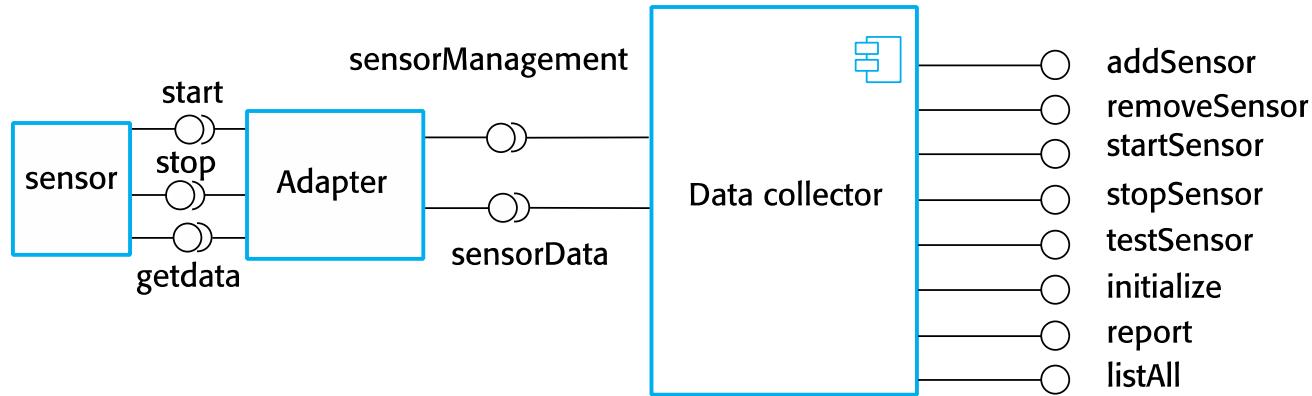
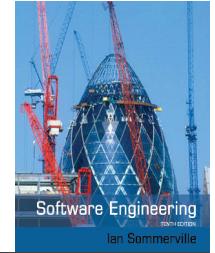
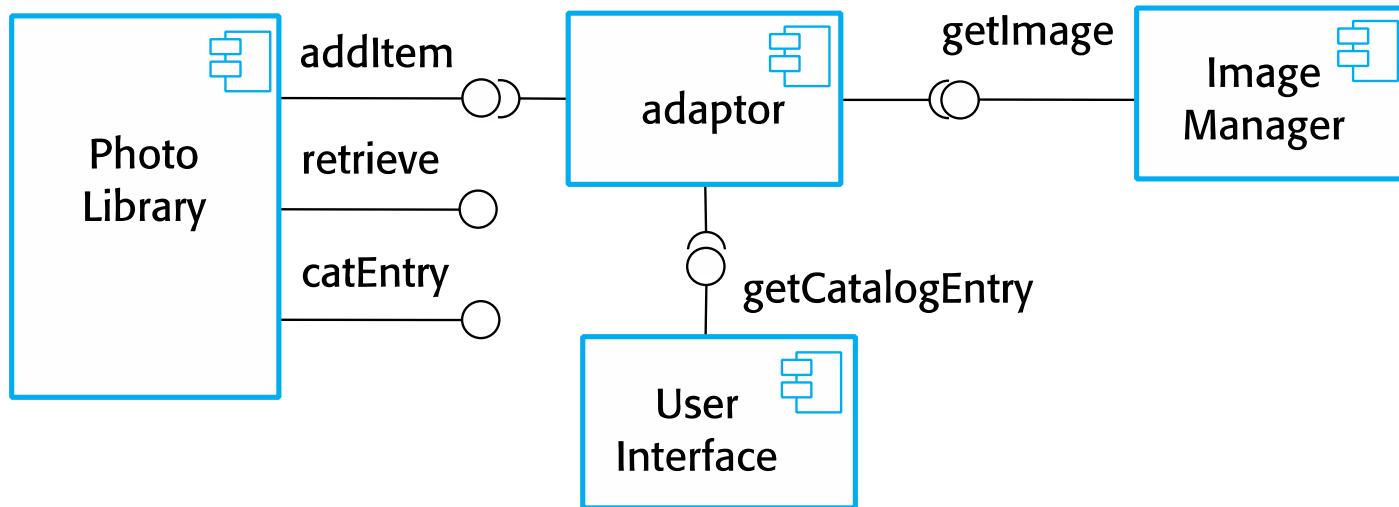


Photo library composition



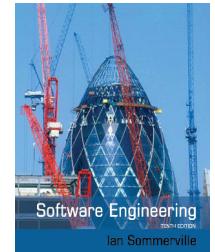
Interface semantics



- ✧ You have to rely on component documentation to decide if interfaces that are syntactically compatible are actually compatible.
- ✧ Consider an interface for a PhotoLibrary component:

```
public void addItem (Identifier pid ; Photograph p; CatalogEntry photodesc) ;  
public Photograph retrieve (Identifier pid) ;  
public CatalogEntry catEntry (Identifier pid) ;
```

Photo Library documentation



Software Engineering

Ian Sommerville

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

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

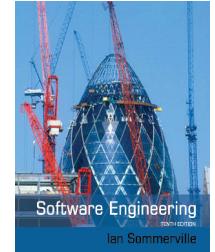
“is the photograph descriptor associated with the catalogue entry as well as the photograph i.e. if I delete the photograph, do I also delete the catalogue information?”

The Object Constraint Language



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

The OCL description of the Photo Library interface



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

context addItem

-- The preconditions specify what must be true before execution of addItem

pre: PhotoLibrary.libSize() > 0

PhotoLibrary.retrieve(pid) = null

-- The postconditions specify what is true after execution

post: libSize () = libSize()@pre + 1

PhotoLibrary.retrieve(pid) = p

PhotoLibrary.catEntry(pid) = photodesc

context delete

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

post: PhotoLibrary.retrieve(pid) = null

PhotoLibrary.catEntry(pid) = PhotoLibrary.catEntry(pid)@pre

PhotoLibrary.libSize() = libSize()@pre—1

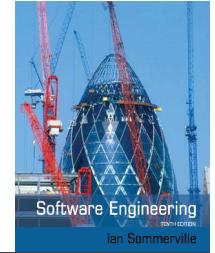


Photo library conditions

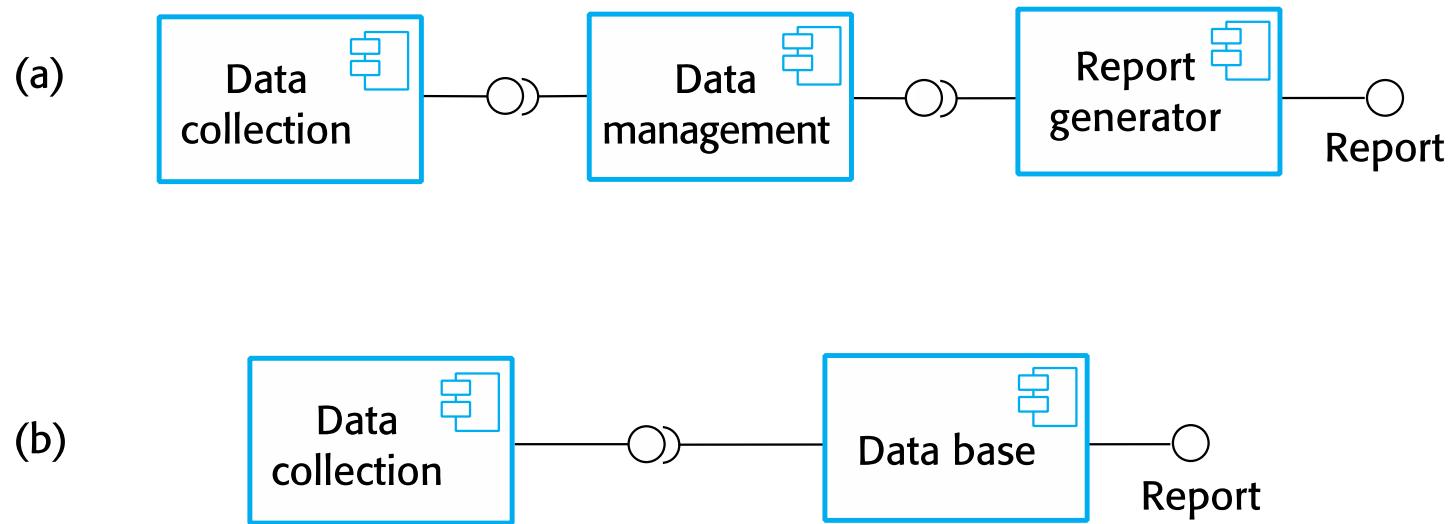
- ✧ As specified, the OCL associated with the Photo Library component states that:
 - There must not be a photograph in the library with the same identifier as the photograph to be entered;
 - The library must exist - assume that creating a library adds a single item to it;
 - Each new entry increases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

Composition trade-offs

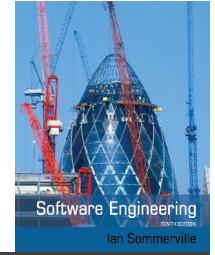


- ✧ When composing components, you may find conflicts between functional and non-functional requirements, and conflicts between the need for rapid delivery and system evolution.
- ✧ You need to make decisions such as:
 - What composition of components is effective for delivering the functional requirements?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

Data collection and report generation components

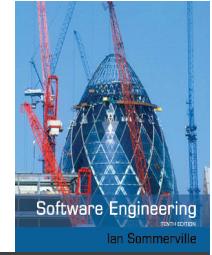


Key points



- ✧ CBSE is a reuse-based approach to defining and implementing loosely coupled components into systems.
- ✧ A component is a software unit whose functionality and dependencies are completely defined by its interfaces.
- ✧ Components may be implemented as executable elements included in a system or as external services.
- ✧ A component model defines a set of standards that component providers and composers should follow.
- ✧ The key CBSE processes are CBSE for reuse and CBSE with reuse.

Key points



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

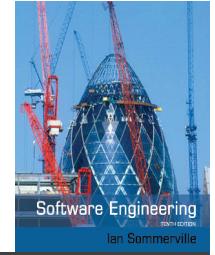


Software Engineering

Ian Sommerville

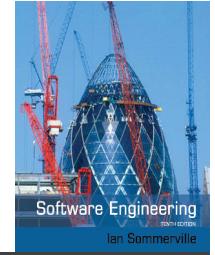
Chapter 17 – Distributed software engineering

Topics covered

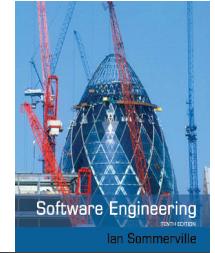


- ✧ Distributed systems
- ✧ Client–server computing
- ✧ Architectural patterns for distributed systems
- ✧ Software as a service

Distributed systems



- ✧ Virtually all large computer-based systems are now distributed systems.
“... a collection of independent computers that appears to the user as a single coherent system.”
- ✧ Information processing is distributed over several computers rather than confined to a single machine.
- ✧ Distributed software engineering is therefore very important for enterprise computing systems.



Distributed system characteristics

- ✧ Resource sharing
 - Sharing of hardware and software resources.
- ✧ Openness
 - Use of equipment and software from different vendors.
- ✧ Concurrency
 - Concurrent processing to enhance performance.
- ✧ Scalability
 - Increased throughput by adding new resources.
- ✧ Fault tolerance
 - The ability to continue in operation after a fault has occurred.



Distributed systems

Distributed systems issues



- ✧ Distributed systems are more complex than systems that run on a single processor.
- ✧ Complexity arises because different parts of the system are independently managed as is the network.
- ✧ There is no single authority in charge of the system so top-down control is impossible.

Design issues



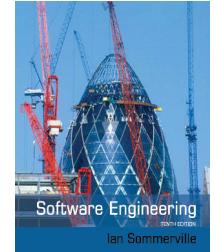
- ✧ *Transparency* To what extent should the distributed system appear to the user as a single system?
- ✧ *Openness* Should a system be designed using standard protocols that support interoperability?
- ✧ *Scalability* How can the system be constructed so that it is scaleable?
- ✧ *Security* How can usable security policies be defined and implemented?
- ✧ *Quality of service* How should the quality of service be specified.
- ✧ *Failure management* How can system failures be detected, contained and repaired?

Transparency



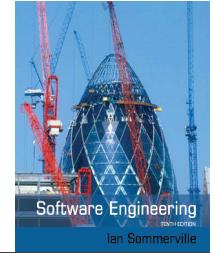
- ✧ Ideally, users should not be aware that a system is distributed and services should be independent of distribution characteristics.
- ✧ In practice, this is impossible because parts of the system are independently managed and because of network delays.
 - Often better to make users aware of distribution so that they can cope with problems
- ✧ To achieve transparency, resources should be abstracted and addressed logically rather than physically. Middleware maps logical to physical resources.

Openness



- ✧ Open distributed systems are systems that are built according to generally accepted standards.
- ✧ Components from any supplier can be integrated into the system and can inter-operate with the other system components.
- ✧ Openness implies that system components can be independently developed in any programming language and, if these conform to standards, they will work with other components.
- ✧ Web service standards for service-oriented architectures were developed to be open standards.

Scalability

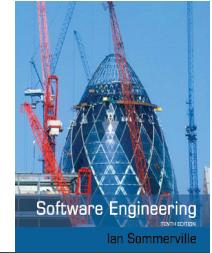


- ✧ The scalability of a system reflects its ability to deliver a high quality service as demands on the system increase
 - *Size* It should be possible to add more resources to a system to cope with increasing numbers of users.
 - *Distribution* It should be possible to geographically disperse the components of a system without degrading its performance.
 - *Manageability* It should be possible to manage a system as it increases in size, even if parts of the system are located in independent organizations.
- ✧ There is a distinction between scaling-up and scaling-out. Scaling up is more powerful system; scaling out is more system instances.

Security



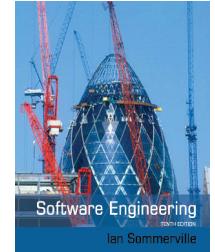
- ✧ When a system is distributed, the number of ways that the system may be attacked is significantly increased, compared to centralized systems.
- ✧ If a part of the system is successfully attacked then the attacker may be able to use this as a 'back door' into other parts of the system.
- ✧ Difficulties in a distributed system arise because different organizations may own parts of the system. These organizations may have mutually incompatible security policies and security mechanisms.



Types of attack

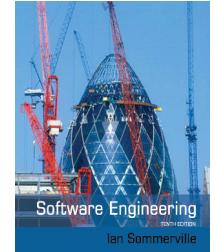
- ✧ The types of attack that a distributed system must defend itself against are:
 - Interception, where communications between parts of the system are intercepted by an attacker so that there is a loss of confidentiality.
 - Interruption, where system services are attacked and cannot be delivered as expected.
 - Denial of service attacks involve bombarding a node with illegitimate service requests so that it cannot deal with valid requests.
 - Modification, where data or services in the system are changed by an attacker.
 - Fabrication, where an attacker generates information that should not exist and then uses this to gain some privileges.

Quality of service



- ✧ The quality of service (QoS) offered by a distributed system reflects the system's ability to deliver its services dependably and with a response time and throughput that is acceptable to its users.
- ✧ Quality of service is particularly critical when the system is dealing with time-critical data such as sound or video streams.
 - In these circumstances, if the quality of service falls below a threshold value then the sound or video may become so degraded that it is impossible to understand.

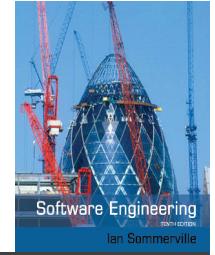
Failure management



- ✧ In a distributed system, it is inevitable that failures will occur, so the system has to be designed to be resilient to these failures.

"You know that you have a distributed system when the crash of a system that you've never heard of stops you getting any work done."
- ✧ Distributed systems should include mechanisms for discovering if a component of the system has failed, should continue to deliver as many services as possible in spite of that failure and, as far as possible, automatically recover from the failure.

Models of interaction



- ✧ Two types of interaction between components in a distributed system
 - Procedural interaction, where one computer calls on a known service offered by another computer and waits for a response.
 - Message-based interaction, involves the sending computer sending information about what is required to another computer. There is no necessity to wait for a response.

Procedural interaction between a diner and a waiter



Tomato soup please

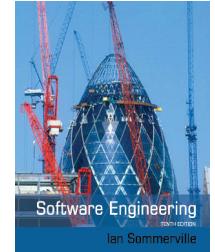
And to follow?

Fillet steak

How would you like it cooked?

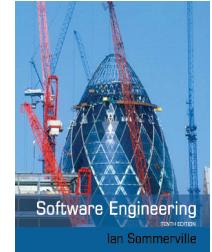
Rare please

Message-based interaction between a waiter and the kitchen



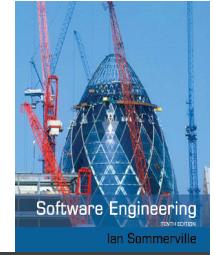
```
<starter>
    <dish name = "soup" type = "tomato" />
    <dish name = "soup" type = "fish" />
    <dish name = "pigeon salad" />
</starter>
<main course>
    <dish name = "steak" type = "sirloin" cooking = "medium" />
    <dish name = "steak" type = "fillet" cooking = "rare" />
    <dish name = "sea bass">
</main>
<accompaniment>
    <dish name = "french fries" portions = "2" />
    <dish name = "salad" portions = "1" />
</accompaniment>
```

Remote procedure calls



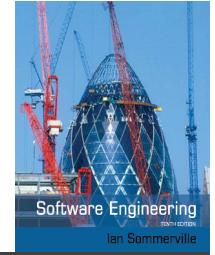
- ✧ Procedural communication in a distributed system is implemented using remote procedure calls (RPC).
- ✧ In a remote procedure call, one component calls another component as if it was a local procedure or method. The middleware in the system intercepts this call and passes it to a remote component.
- ✧ This carries out the required computation and, via the middleware, returns the result to the calling component.
- ✧ A problem with RPCs is that the caller and the callee need to be available at the time of the communication, and they must know how to refer to each other.

Message passing



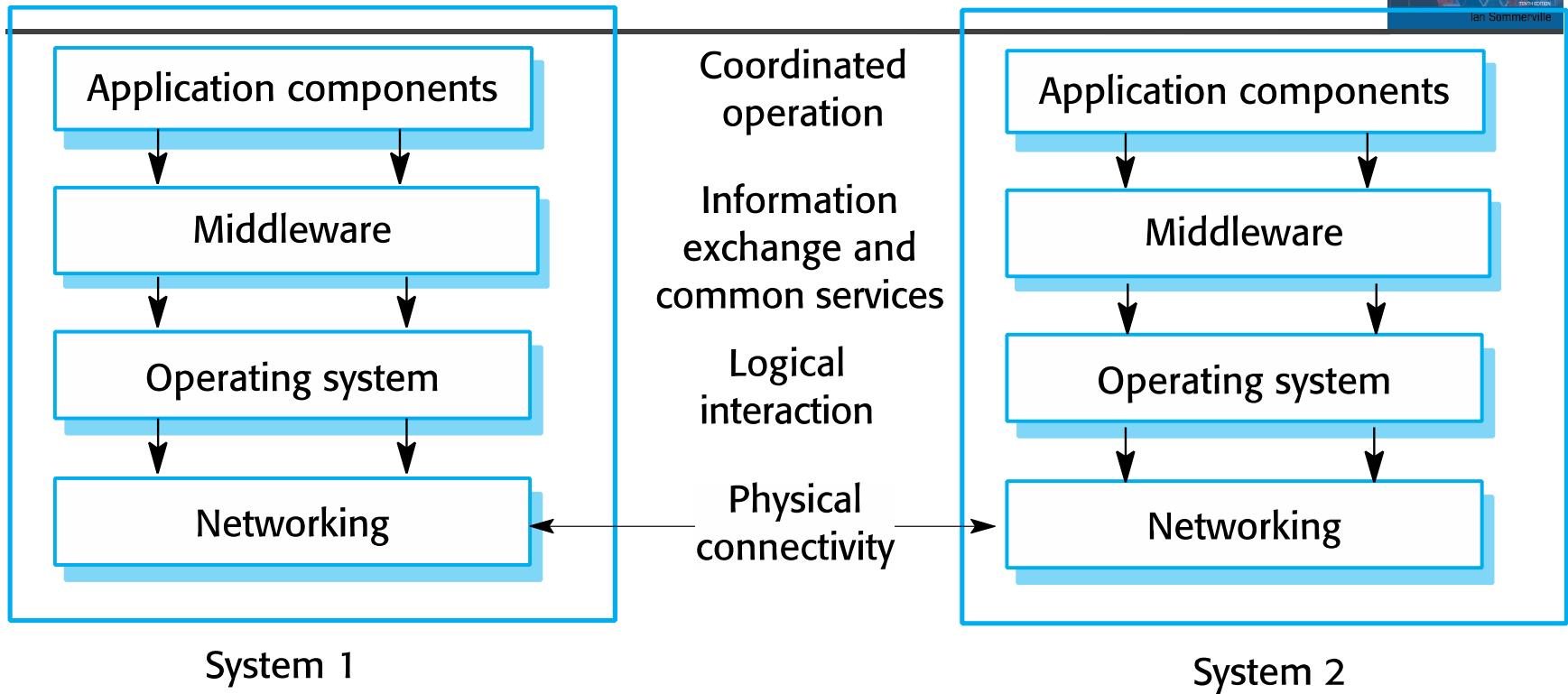
- ✧ Message-based interaction normally involves one component creating a message that details the services required from another component.
- ✧ Through the system middleware, this is sent to the receiving component.
- ✧ The receiver parses the message, carries out the computations and creates a message for the sending component with the required results.
- ✧ In a message-based approach, it is not necessary for the sender and receiver of the message to be aware of each other. They simple communicate with the middleware.

Middleware

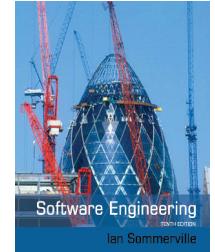


- ✧ The components in a distributed system may be implemented in different programming languages and may execute on completely different types of processor. Models of data, information representation and protocols for communication may all be different.
- ✧ Middleware is software that can manage these diverse parts, and ensure that they can communicate and exchange data.

Middleware in a distributed system



Middleware support

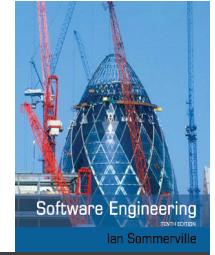


- ✧ Interaction support, where the middleware coordinates interactions between different components in the system
 - The middleware provides location transparency in that it isn't necessary for components to know the physical locations of other components.
- ✧ The provision of common services, where the middleware provides reusable implementations of services that may be required by several components in the distributed system.
 - By using these common services, components can easily inter-operate and provide user services in a consistent way.



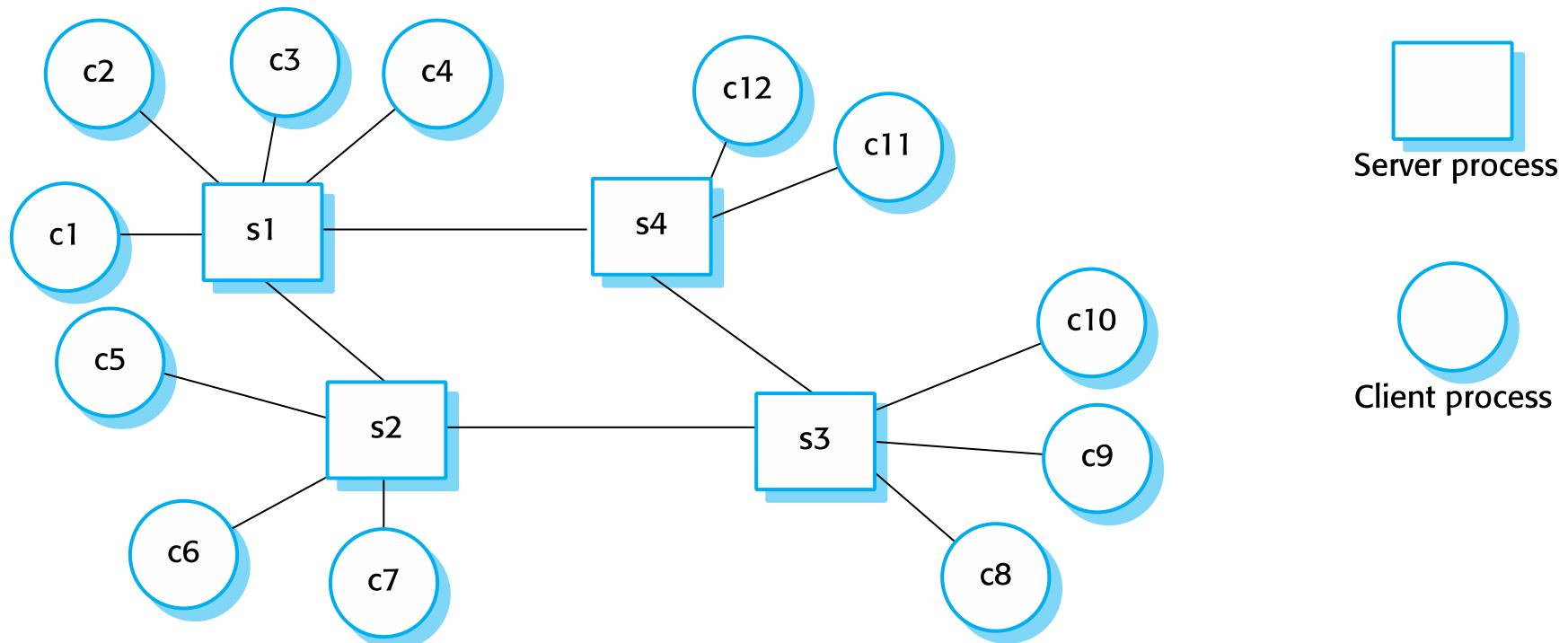
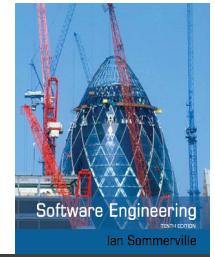
Client-server computing

Client-server computing

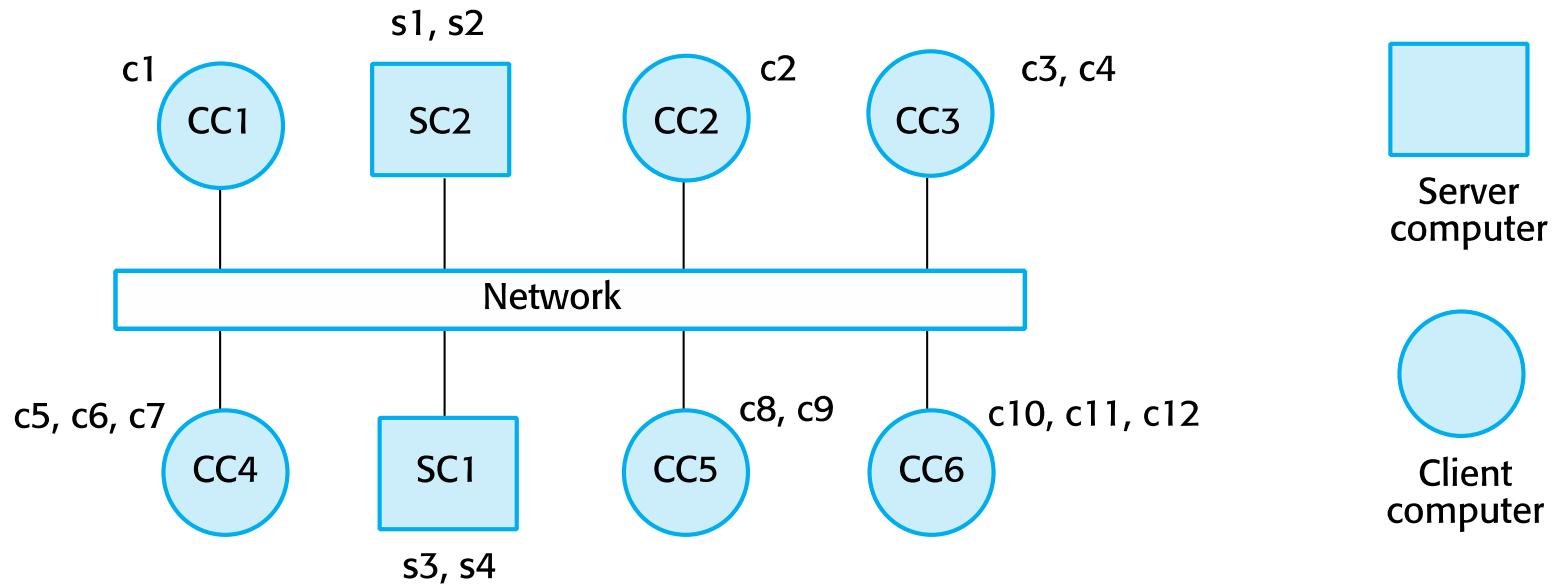


- ✧ Distributed systems that are accessed over the Internet are normally organized as client-server systems.
- ✧ In a client-server system, the user interacts with a program running on their local computer (e.g. a web browser or mobile application). This interacts with another program running on a remote computer (e.g. a web server).
- ✧ The remote computer provides services, such as access to web pages, which are available to external clients.

Client–server interaction



Mapping of clients and servers to networked computers



Layered architectural model for client–server applications



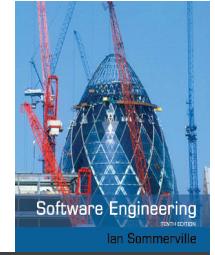
Presentation

Data handling

Application processing

Database

Layers in a client/server system



✧ *Presentation*

- concerned with presenting information to the user and managing all user interaction.

✧ *Data handling*

- manages the data that is passed to and from the client.
Implement checks on the data, generate web pages, etc.

✧ Application processing layer

- concerned with implementing the logic of the application and so providing the required functionality to end users.

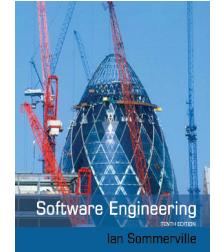
✧ Database

- Stores data and provides transaction management services, etc.



Architectural patterns for distributed systems

Architectural patterns



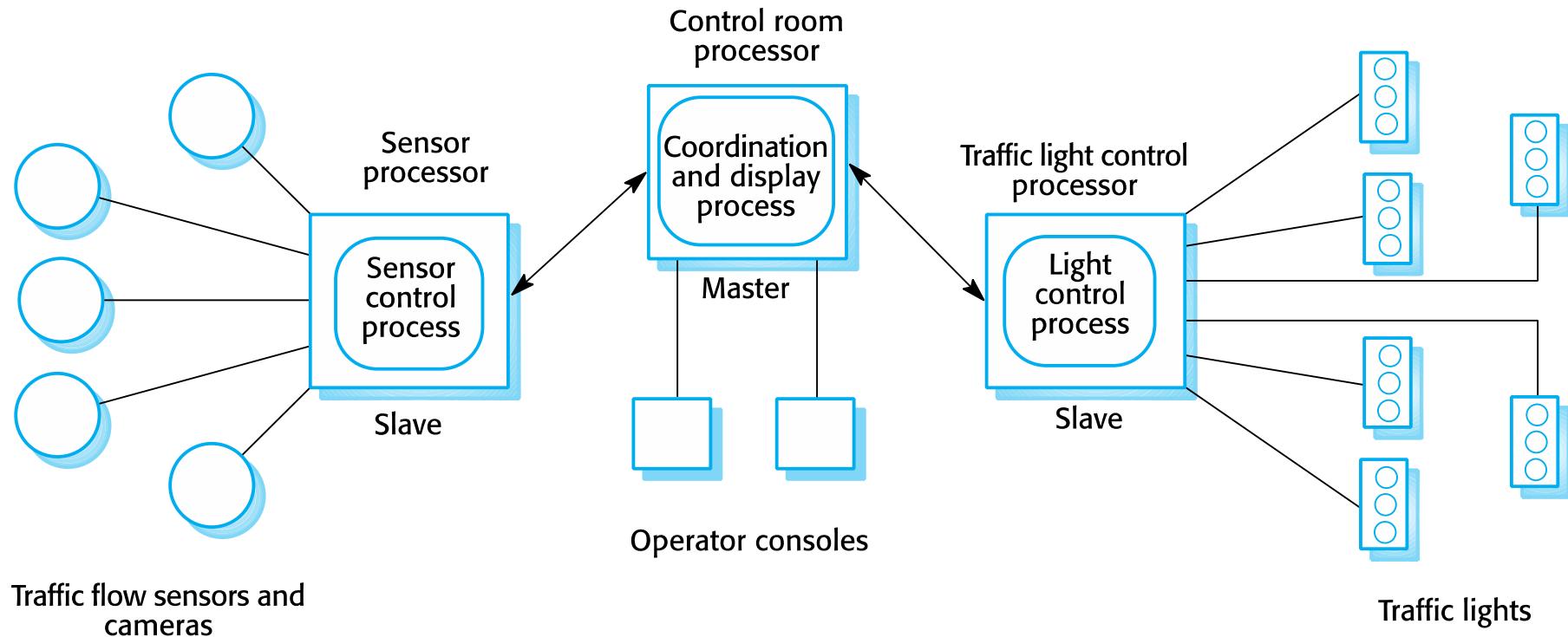
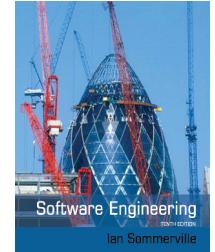
- ✧ Widely used ways of organizing the architecture of a distributed system:
 - *Master-slave architecture*, which is used in real-time systems in which guaranteed interaction response times are required.
 - *Two-tier client-server architecture*, which is used for simple client-server systems, and where the system is centralized for security reasons.
 - *Multi-tier client-server architecture*, which is used when there is a high volume of transactions to be processed by the server.
 - *Distributed component architecture*, which is used when resources from different systems and databases need to be combined, or as an implementation model for multi-tier client-server systems.
 - *Peer-to-peer architecture*, which is used when clients exchange locally stored information and the role of the server is to introduce clients to each other

Master-slave architectures



- ✧ Master-slave architectures are commonly used in real-time systems where there may be separate processors associated with data acquisition from the system's environment, data processing and computation and actuator management.
- ✧ The 'master' process is usually responsible for computation, coordination and communications and it controls the 'slave' processes.
- ✧ 'Slave' processes are dedicated to specific actions, such as the acquisition of data from an array of sensors.

A traffic management system with a master-slave architecture

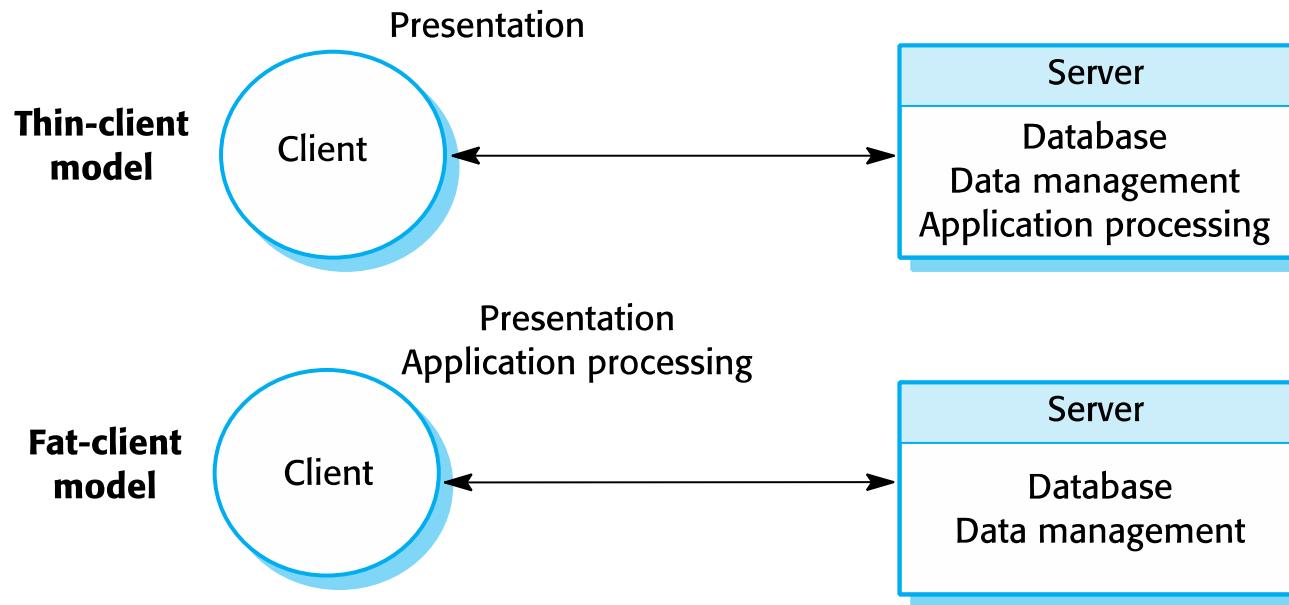


Two-tier client server architectures

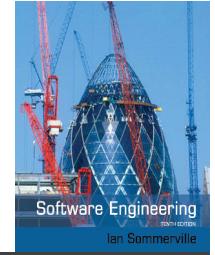


- ✧ In a two-tier client-server architecture, the system is implemented as a single logical server plus an indefinite number of clients that use that server.
 - Thin-client model, where the presentation layer is implemented on the client and all other layers (data management, application processing and database) are implemented on a server.
 - Fat-client model, where some or all of the application processing is carried out on the client. Data management and database functions are implemented on the server.

Thin- and fat-client architectural models

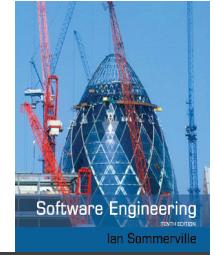


Thin client model



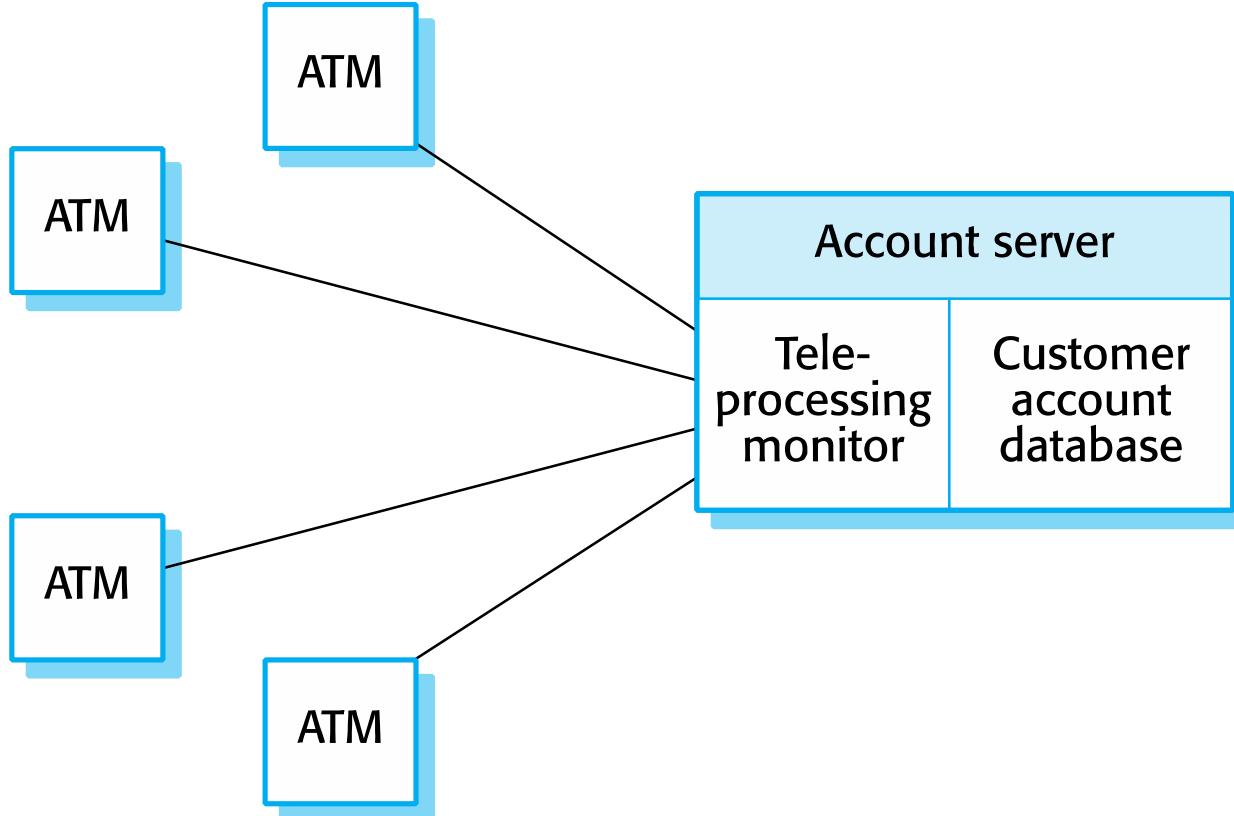
- ✧ Used when legacy systems are migrated to client server architectures.
 - The legacy system acts as a server in its own right with a graphical interface implemented on a client.
- ✧ A major disadvantage is that it places a heavy processing load on both the server and the network.

Fat client model



- ✧ More processing is delegated to the client as the application processing is locally executed.
- ✧ Most suitable for new C/S systems where the capabilities of the client system are known in advance.
- ✧ More complex than a thin client model especially for management. New versions of the application have to be installed on all clients.

A fat-client architecture for an ATM system

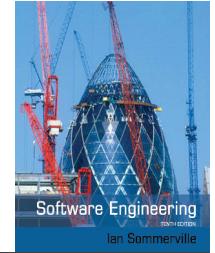


Thin and fat clients



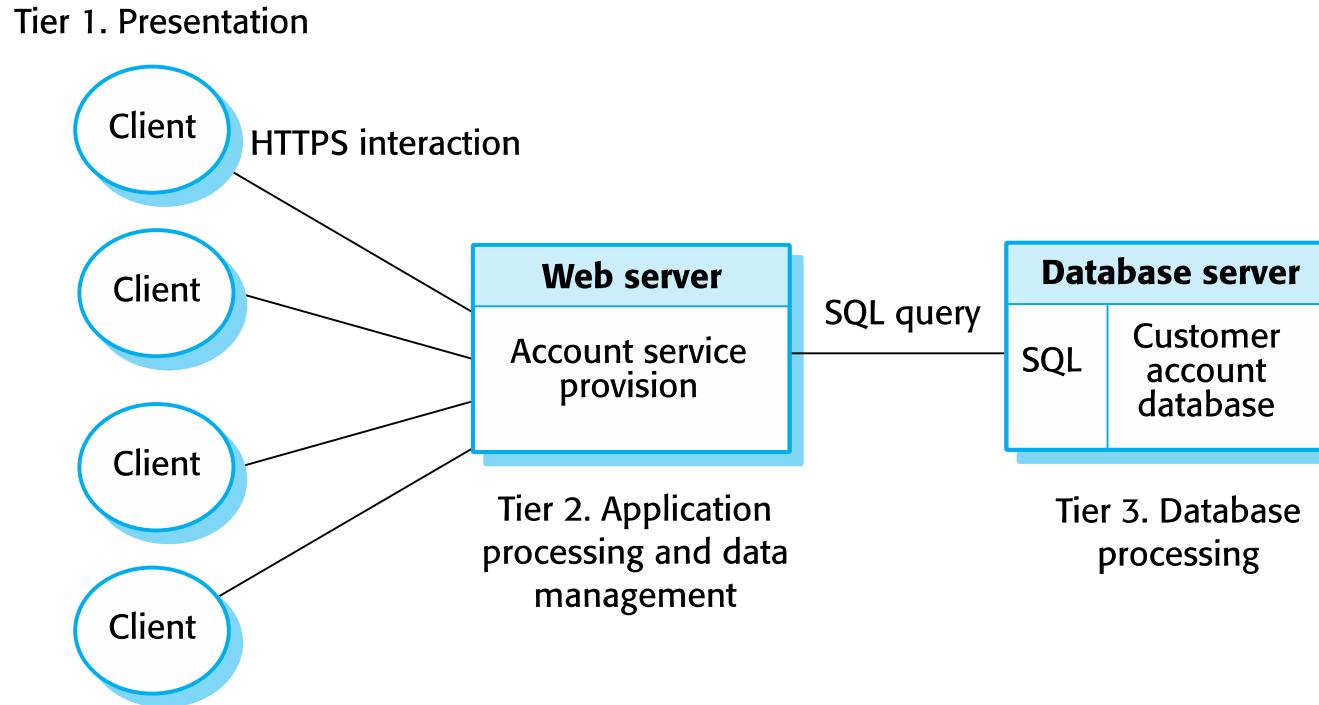
- ✧ Distinction between thin and fat client architectures has become blurred
- ✧ Javascript allows local processing in a browser so ‘fat-client’ functionality available without software installation
- ✧ Mobile apps carry out some local processing to minimize demands on network
- ✧ Auto-update of apps reduces management problems
- ✧ There are now very few thin-client applications with all processing carried out on remote server.

Multi-tier client-server architectures



- ✧ In a ‘multi-tier client–server’ architecture, the different layers of the system, namely presentation, data management, application processing, and database, are separate processes that may execute on different processors.
- ✧ This avoids problems with scalability and performance if a thin-client two-tier model is chosen, or problems of system management if a fat-client model is used.

Three-tier architecture for an Internet banking system



Use of client–server architectural patterns



Architecture	Applications
Two-tier client–server architecture with thin clients	<p>Legacy system applications that are used when separating application processing and data management is impractical. Clients may access these as services, as discussed in Section 18.4.</p> <p>Computationally intensive applications such as compilers with little or no data management.</p> <p>Data-intensive applications (browsing and querying) with nonintensive application processing. Browsing the Web is the most common example of a situation where this architecture is used.</p>

Use of client–server architectural patterns



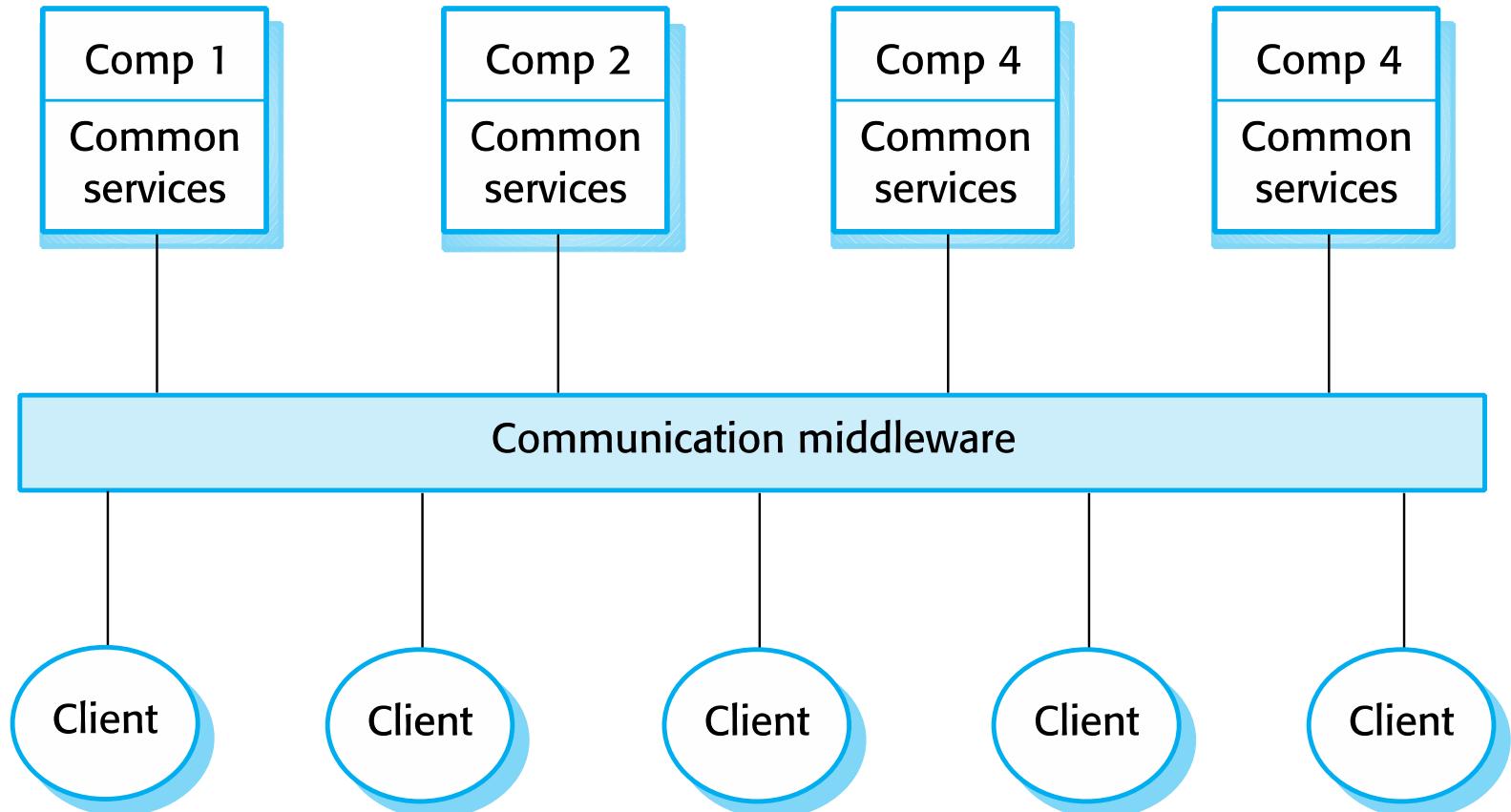
Architecture	Applications
Two-tier client–server architecture with fat clients	<p>Applications where application processing is provided by off-the-shelf software (e.g., Microsoft Excel) on the client.</p> <p>Applications where computationally intensive processing of data (e.g., data visualization) is required.</p> <p>Mobile applications where internet connectivity cannot be guaranteed. Some local processing using cached information from the database is therefore possible.</p>
Multi-tier client–server architecture	<p>Large-scale applications with hundreds or thousands of clients.</p> <p>Applications where both the data and the application are volatile.</p> <p>Applications where data from multiple sources are integrated.</p>

Distributed component architectures



- ✧ There is no distinction in a distributed component architecture between clients and servers.
- ✧ Each distributable entity is a component that provides services to other components and receives services from other components.
- ✧ Component communication is through a middleware system.

A distributed component architecture

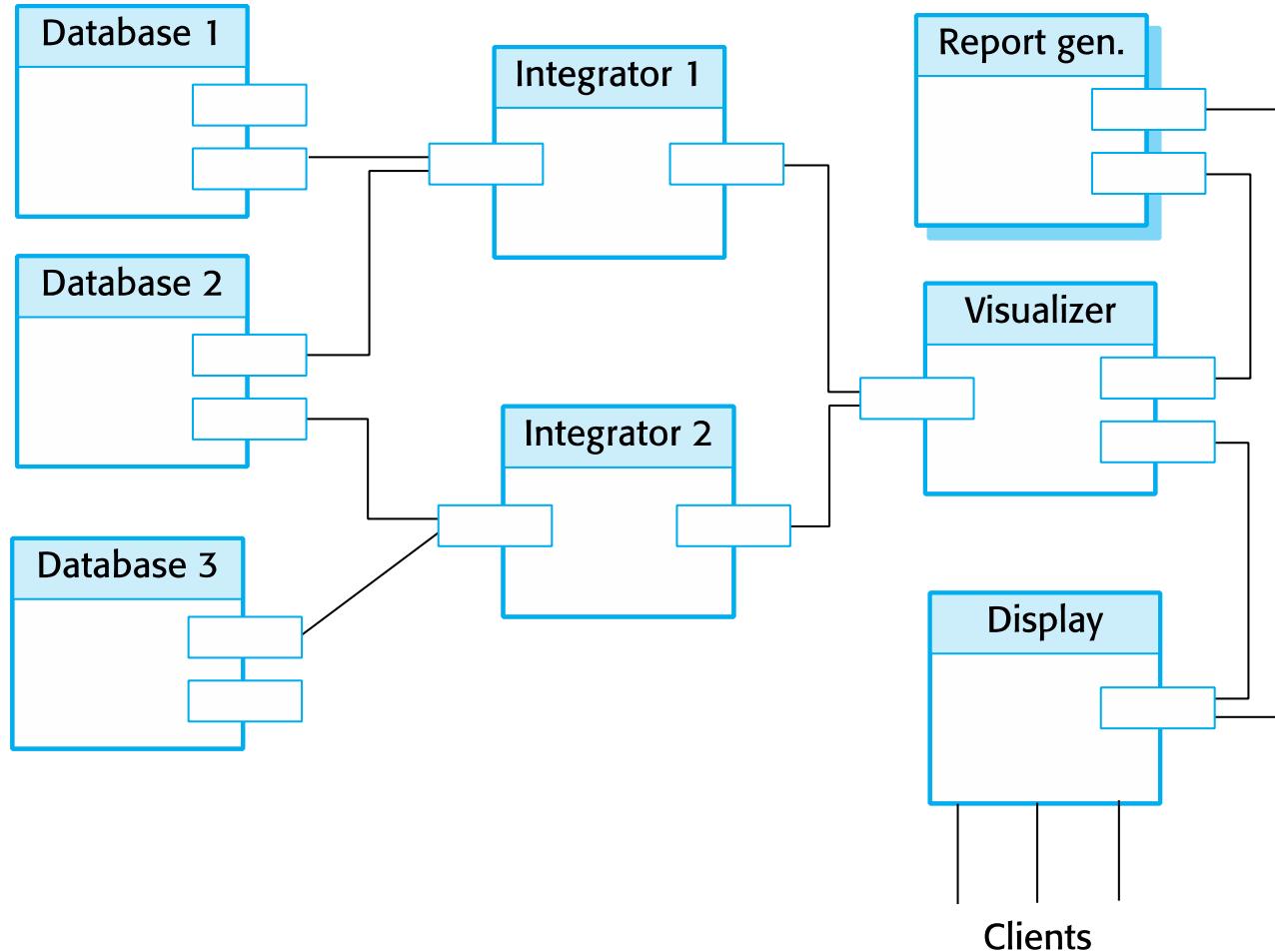
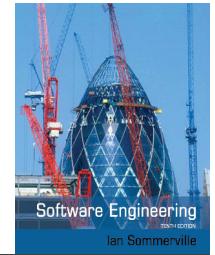


Benefits of distributed component architecture



- ✧ It allows the system designer to delay decisions on where and how services should be provided.
- ✧ It is a very open system architecture that allows new resources to be added as required.
- ✧ The system is flexible and scalable.
- ✧ It is possible to reconfigure the system dynamically with objects migrating across the network as required.

A distributed component architecture for a data mining system

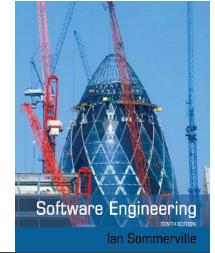


Disadvantages of distributed component architecture



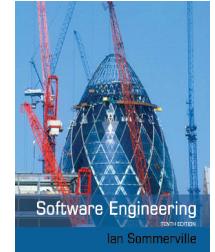
- ✧ Distributed component architectures suffer from two major disadvantages:
 - They are more complex to design than client–server systems. Distributed component architectures are difficult for people to visualize and understand.
 - Standardized middleware for distributed component systems has never been accepted by the community. Different vendors, such as Microsoft and Sun, have developed different, incompatible middleware.
- ✧ As a result of these problems, service-oriented architectures are replacing distributed component architectures in many situations.

Peer-to-peer architectures



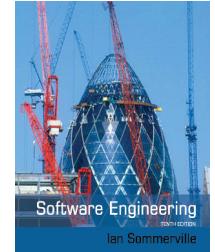
- ✧ Peer to peer (p2p) systems are decentralised systems where computations may be carried out by any node in the network.
- ✧ The overall system is designed to take advantage of the computational power and storage of a large number of networked computers.
- ✧ Most p2p systems have been personal systems but there is increasing business use of this technology.

Peer-to-peer systems



- ✧ File sharing systems based on the BitTorrent protocol
- ✧ Messaging systems such as Jabber
- ✧ Payments systems – Bitcoin
- ✧ Databases – Freenet is a decentralized database
- ✧ Phone systems – Viber
- ✧ Computation systems - SETI@home

P2p architectural models

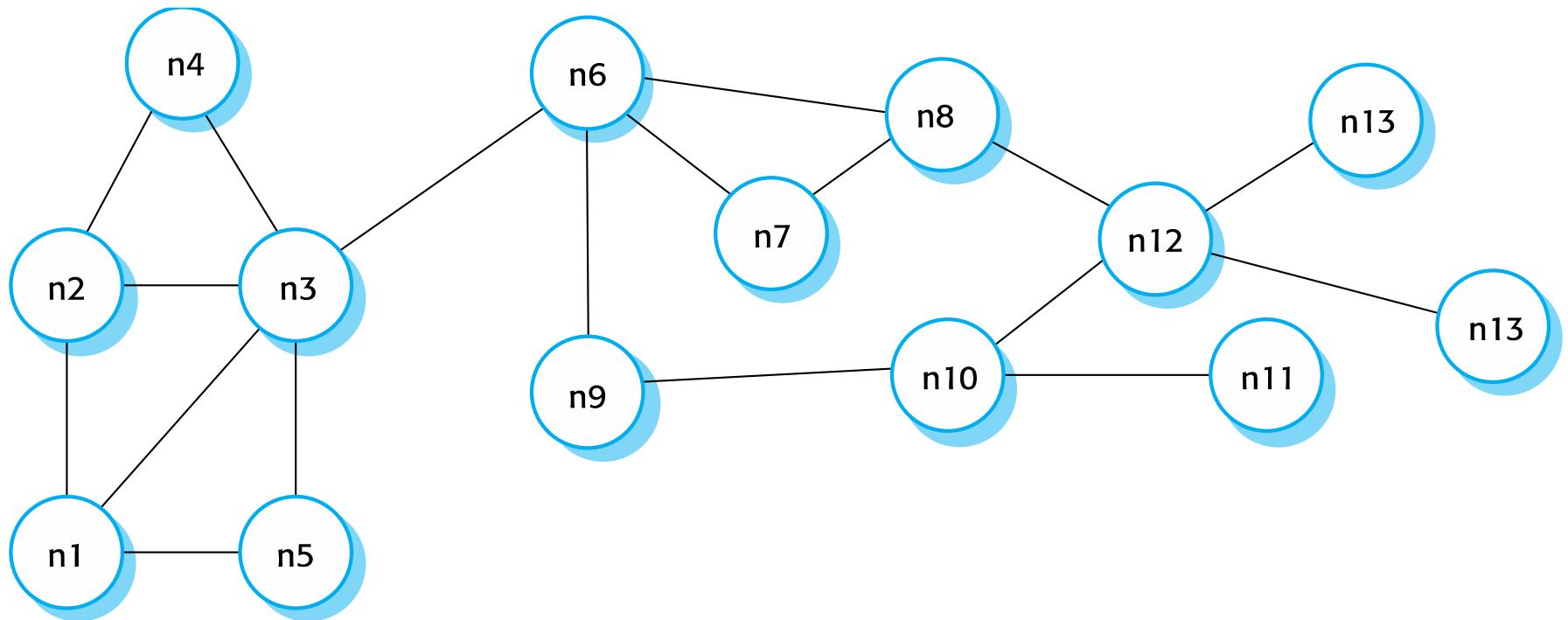


- ✧ The logical network architecture
 - Decentralised architectures;
 - Semi-centralised architectures.
- ✧ Application architecture
 - The generic organisation of components making up a p2p application.
- ✧ Focus here on network architectures.

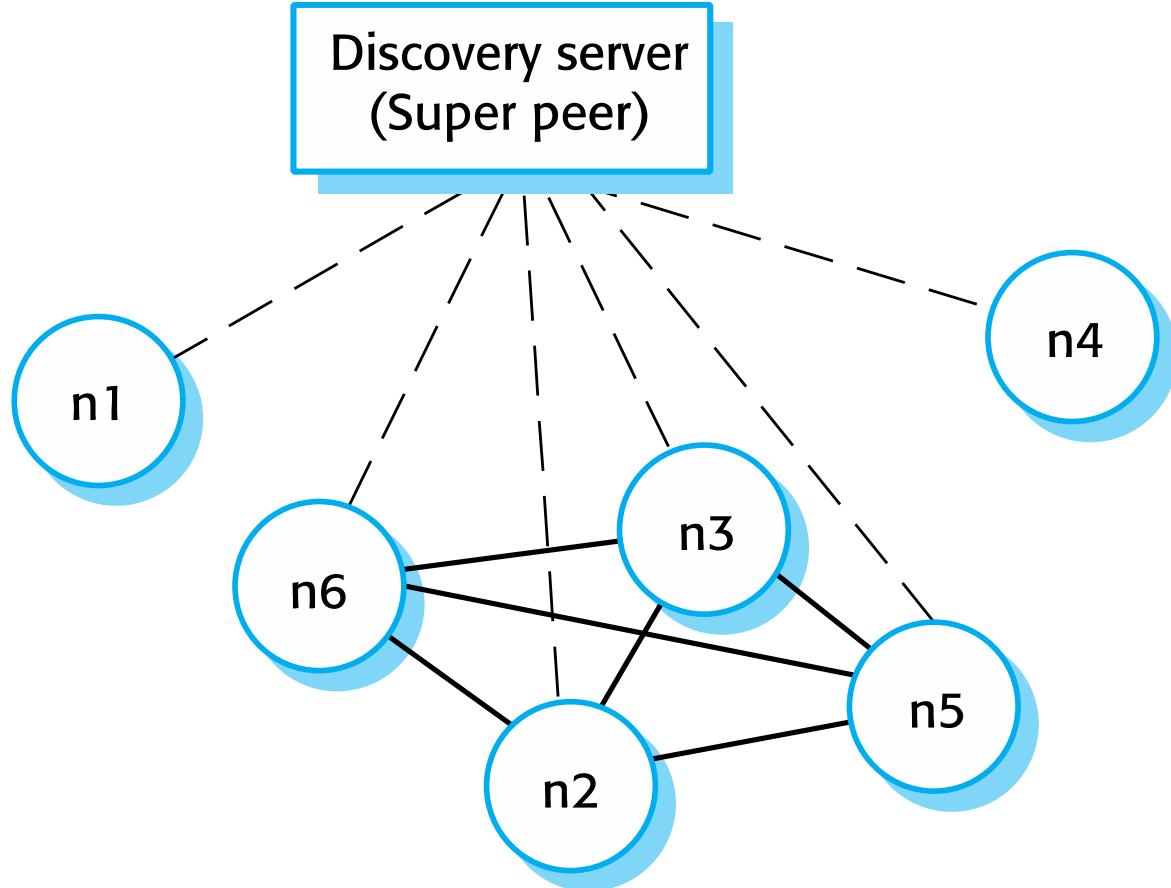
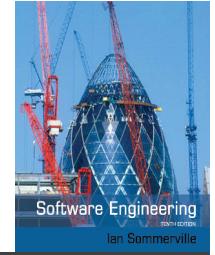
A decentralized p2p architecture



Software Engineering
Ian Sommerville



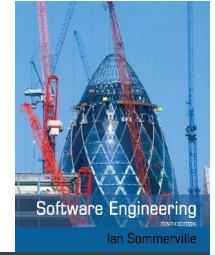
A semicentralized p2p architecture





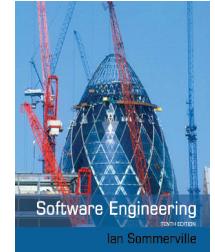
Software as a service

Use of p2p architecture



- ✧ When a system is computationally-intensive and it is possible to separate the processing required into a large number of independent computations.
- ✧ When a system primarily involves the exchange of information between individual computers on a network and there is no need for this information to be centrally-stored or managed.

Security issues in p2p system

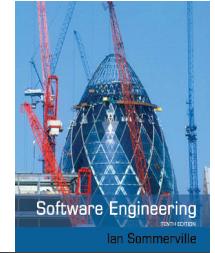


- ✧ Security concerns are the principal reason why p2p architectures are not widely used.
- ✧ The lack of central management means that malicious nodes can be set up to deliver spam and malware to other nodes in the network.
- ✧ P2P communications require careful setup to protect local information and if not done correctly, then this is exposed to other peers.

Software as a service



- ✧ Software as a service (SaaS) involves hosting the software remotely and providing access to it over the Internet.
 - Software is deployed on a server (or more commonly a number of servers) and is accessed through a web browser. It is not deployed on a local PC.
 - The software is owned and managed by a software provider, rather than the organizations using the software.
 - Users may pay for the software according to the amount of use they make of it or through an annual or monthly subscription.



Key elements of SaaS

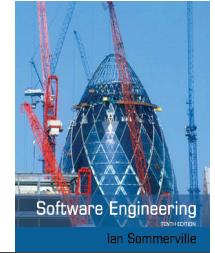
- ✧ Software is deployed on a server (or more commonly a number of servers) and is accessed through a web browser. It is not deployed on a local PC.
- ✧ The software is owned and managed by a software provider, rather than the organizations using the software.
- ✧ Users may pay for the software according to the amount of use they make of it or through an annual or monthly subscription. Sometimes, the software is free for anyone to use but users must then agree to accept advertisements, which fund the software service.

SaaS and SOA



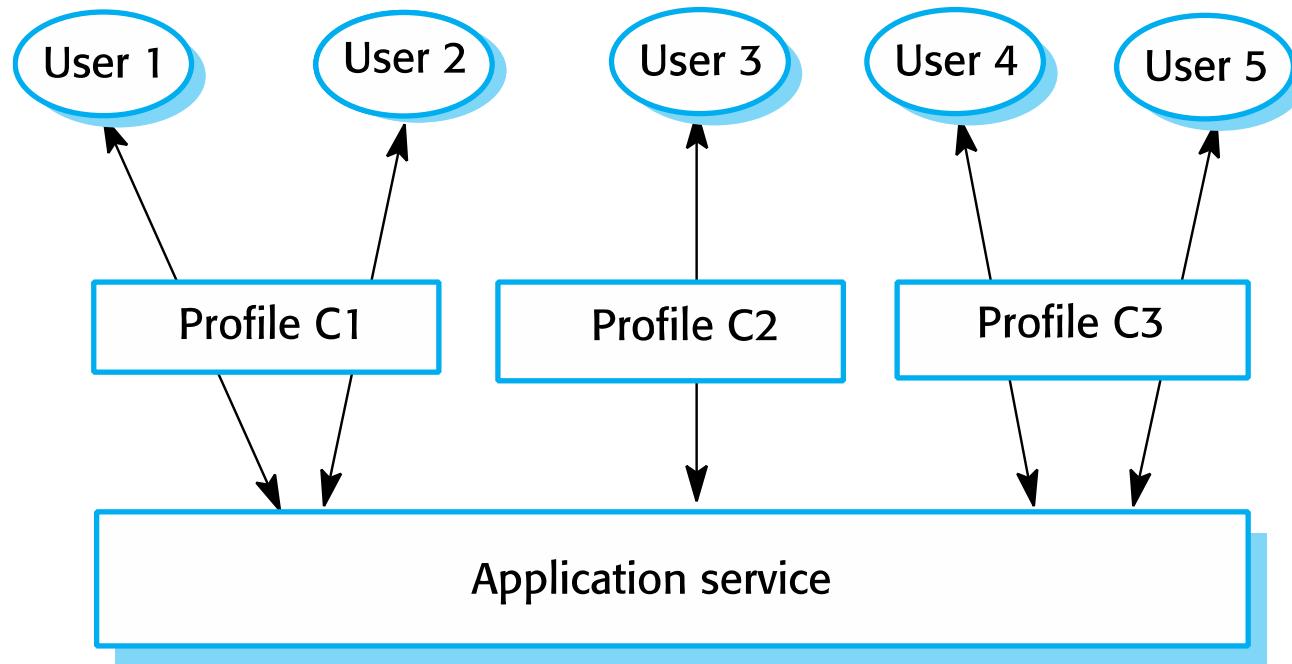
- ✧ Software as a service is a way of providing functionality on a remote server with client access through a web browser. The server maintains the user's data and state during an interaction session. Transactions are usually long transactions e.g. editing a document.
- ✧ Service-oriented architecture is an approach to structuring a software system as a set of separate, stateless services. These may be provided by multiple providers and may be distributed. Typically, transactions are short transactions where a service is called, does something then returns a result.

Implementation factors for SaaS

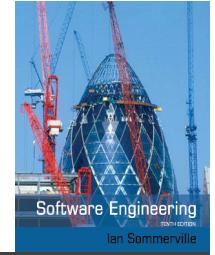


- ✧ *Configurability* How do you configure the software for the specific requirements of each organization?
- ✧ *Multi-tenancy* How do you present each user of the software with the impression that they are working with their own copy of the system while, at the same time, making efficient use of system resources?
- ✧ *Scalability* How do you design the system so that it can be scaled to accommodate an unpredictably large number of users?

Configuration of a software system offered as a service

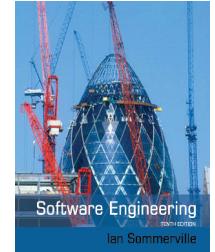


Service configuration



- ✧ Branding, where users from each organization, are presented with an interface that reflects their own organization.
- ✧ Business rules and workflows, where each organization defines its own rules that govern the use of the service and its data.
- ✧ Database extensions, where each organization defines how the generic service data model is extended to meet its specific needs.
- ✧ Access control, where service customers create individual accounts for their staff and define the resources and functions that are accessible to each of their users.

Multi-tenancy



- ✧ Multi-tenancy is a situation in which many different users access the same system and the system architecture is defined to allow the efficient sharing of system resources.
- ✧ It must appear to each user that they have the sole use of the system.
- ✧ Multi-tenancy involves designing the system so that there is an absolute separation between the system functionality and the system data.

A multitenant database



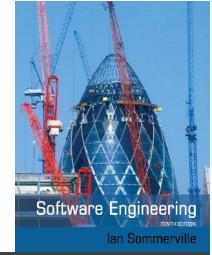
Tenant	Key	Name	Address
234	C100	XYZ Corp	43, Anystreet, Sometown
234	C110	BigCorp	2, Main St, Motown
435	X234	J. Bowie	56, Mill St, Starville
592	PP37	R. Burns	Alloway, Ayrshire

Scalability



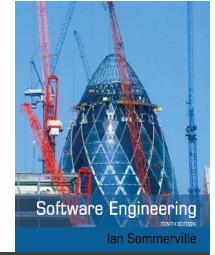
- ✧ Develop applications where each component is implemented as a simple stateless service that may be run on any server.
- ✧ Design the system using asynchronous interaction so that the application does not have to wait for the result of an interaction (such as a read request).
- ✧ Manage resources, such as network and database connections, as a pool so that no single server is likely to run out of resources.
- ✧ Design your database to allow fine-grain locking. That is, do not lock out whole records in the database when only part of a record is in use.

Key points



- ✧ The benefits of distributed systems are that they can be scaled to cope with increasing demand, can continue to provide user services if parts of the system fail, and they enable resources to be shared.
- ✧ Issues to be considered in the design of distributed systems include transparency, openness, scalability, security, quality of service and failure management.
- ✧ Client–server systems are structured into layers, with the presentation layer implemented on a client computer. Servers provide data management, application and database services.
- ✧ Client-server systems may have several tiers, with different layers of the system distributed to different computers.

Key points



- ✧ Architectural patterns for distributed systems include master-slave architectures, two-tier and multi-tier client-server architectures, distributed component architectures and peer-to-peer architectures.
- ✧ Distributed component systems require middleware to handle component communications and to allow components to be added to and removed from the system.
- ✧ Peer-to-peer architectures are decentralized with no distinguished clients and servers. Computations can be distributed over many systems in different organizations.
- ✧ Software as a service is a way of deploying applications as thin client- server systems, where the client is a web browser.



Chapter 18 – Service-oriented Software Engineering

Topics covered



- ✧ Service-oriented architectures
- ✧ RESTful services
- ✧ Service engineering
- ✧ Service composition

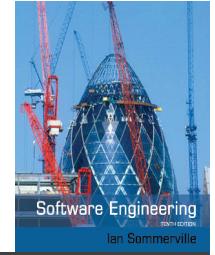
Web services



- ✧ A web service is an instance of a more general notion of a service:

“an act or performance offered by one party to another. Although the process may be tied to a physical product, the performance is essentially intangible and does not normally result in ownership of any of the factors of production”.
- ✧ The essence of a service, therefore, is that the provision of the service is independent of the application using the service.
- ✧ Service providers can develop specialized services and offer these to a range of service users from different organizations.

Reusable services



- ✧ Services are reusable components that are independent (no requires interface) and are loosely coupled.
- ✧ A web service is:
 - *A loosely coupled, reusable software component that encapsulates discrete functionality, which may be distributed and programmatically accessed. A web service is a service that is accessed using standard Internet and XML-based protocols.*
- ✧ Services are platform and implementation-language independent

Benefits of service-oriented approach



- ✧ Services can be offered by any service provider inside or outside of an organisation so organizations can create applications by integrating services from a range of providers.
- ✧ The service provider makes information about the service public so that any authorised user can use the service.
- ✧ Applications can delay the binding of services until they are deployed or until execution. This means that applications can be reactive and adapt their operation to cope with changes to their execution environment.

Benefits of a service-oriented approach



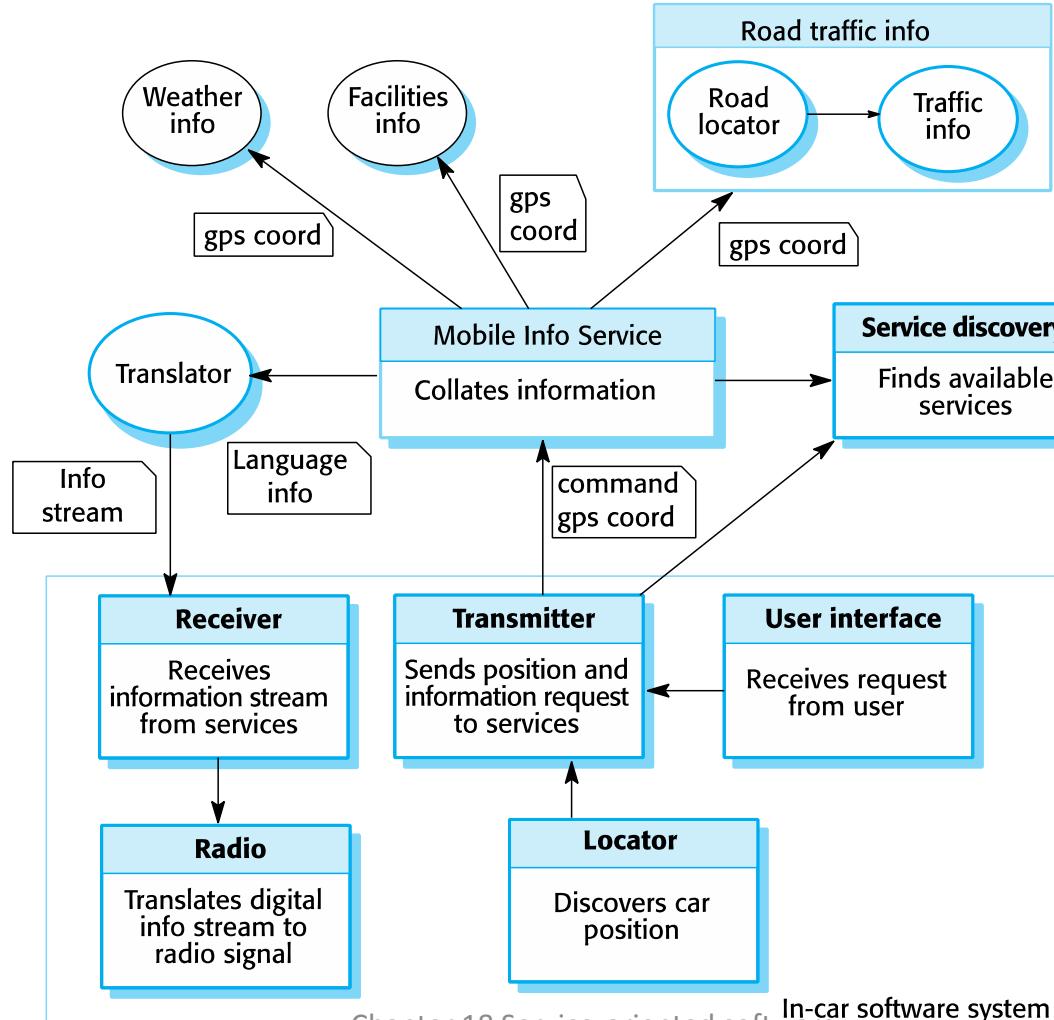
- ✧ Opportunistic construction of new services is possible. A service provider may recognise new services that can be created by linking existing services in innovative ways.
- ✧ Service users can pay for services according to their use rather than their provision. Instead of buying a rarely-used component, the application developers can use an external service that will be paid for only when required.
- ✧ Applications can be made smaller, which is particularly important for mobile devices with limited processing and memory capabilities. Computationally-intensive processing can be offloaded to external services.

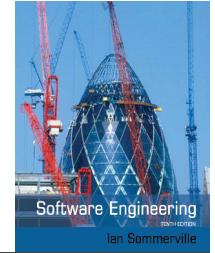
Services scenario



- ✧ An in-car information system provides drivers with information on weather, road traffic conditions, local information etc. This is linked to car audio system so that information is delivered as a signal on a specific channel.
- ✧ The car is equipped with GPS receiver to discover its position and, based on that position, the system accesses a range of information services. Information may be delivered in the driver's specified language.

A service-based, in-car information system





Advantage of SOA for this application

- ✧ It is not necessary to decide when the system is programmed or deployed what service provider should be used or what specific services should be accessed.
 - As the car moves around, the in-car software uses the service discovery service to find the most appropriate information service and binds to that.
 - Because of the use of a translation service, it can move across borders and therefore make local information available to people who don't speak the local language.

Service-oriented software engineering

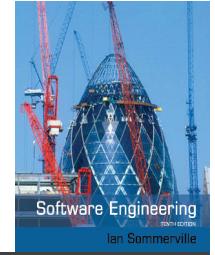


- ✧ As significant a development as object-oriented development.
- ✧ Building applications based on services allows companies and other organizations to cooperate and make use of each other's business functions.
- ✧ Service-based applications may be constructed by linking services from various providers using either a standard programming language or a specialized workflow language.



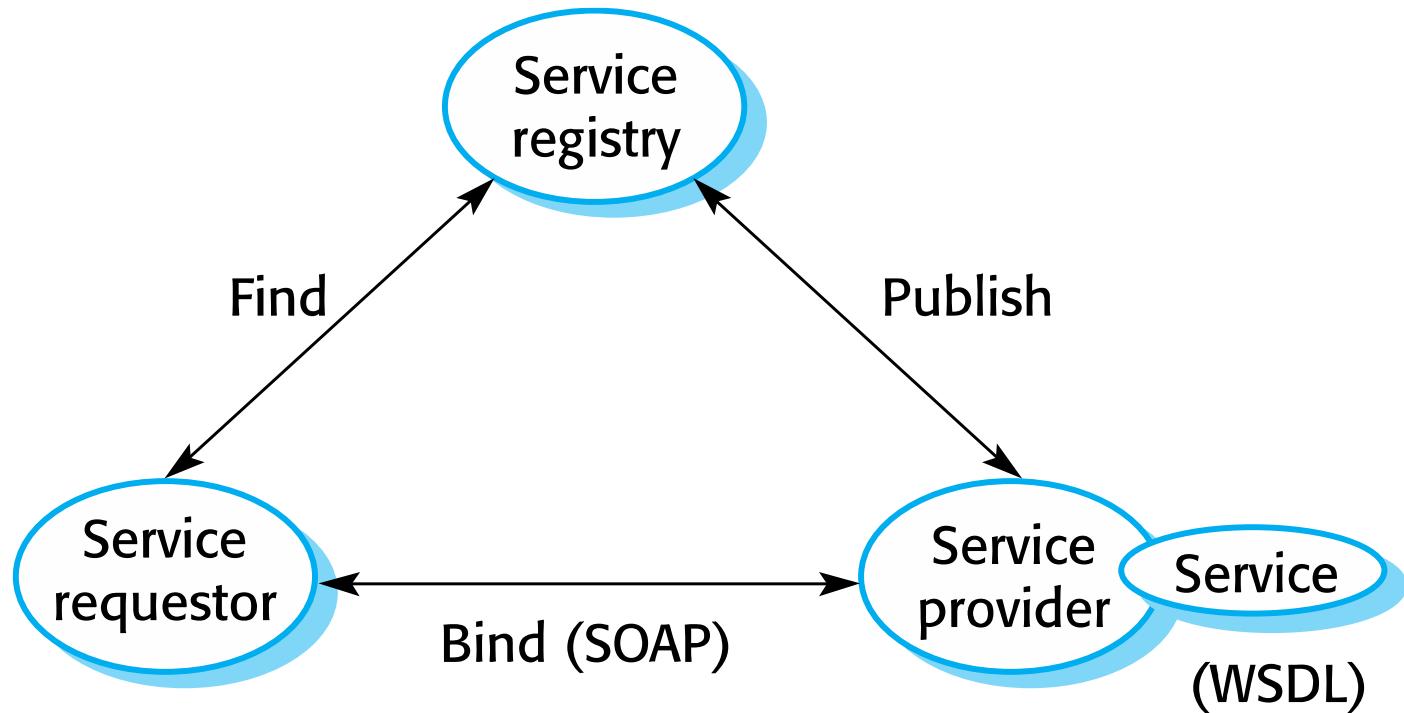
Service-oriented architecture

Service-oriented architectures

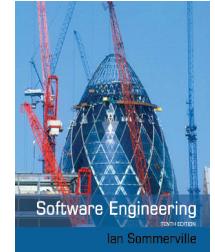


- ✧ A means of developing distributed systems where the components are stand-alone services
- ✧ Services may execute on different computers from different service providers
- ✧ Standard protocols have been developed to support service communication and information exchange

Service-oriented architecture

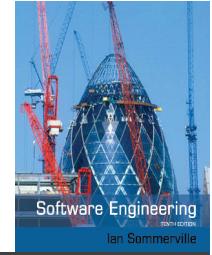


Benefits of SOA



- ✧ Services can be provided locally or outsourced to external providers
- ✧ Services are language-independent
- ✧ Investment in legacy systems can be preserved
- ✧ Inter-organisational computing is facilitated through simplified information exchange

Key standards



✧ SOAP

- A message exchange standard that supports service communication

✧ WSDL (Web Service Definition Language)

- This standard allows a service interface and its bindings to be defined

✧ WS-BPEL

- A standard for workflow languages used to define service composition



Web service standards

XML technologies (XML, XSD, XSLT,)

Support (WS-Security, WS-Addressing, ...)

Process (WS-BPEL)

Service definition (UDDI, WSDL)

Messaging (SOAP)

Transport (HTTP, HTTPS, SMTP, ...)

Service-oriented software engineering



- ✧ Existing approaches to software engineering have to evolve to reflect the service-oriented approach to software development
 - Service engineering. The development of dependable, reusable services
 - Software development for reuse
 - Software development with services. The development of dependable software where services are the fundamental components
 - Software development with reuse

Services as reusable components



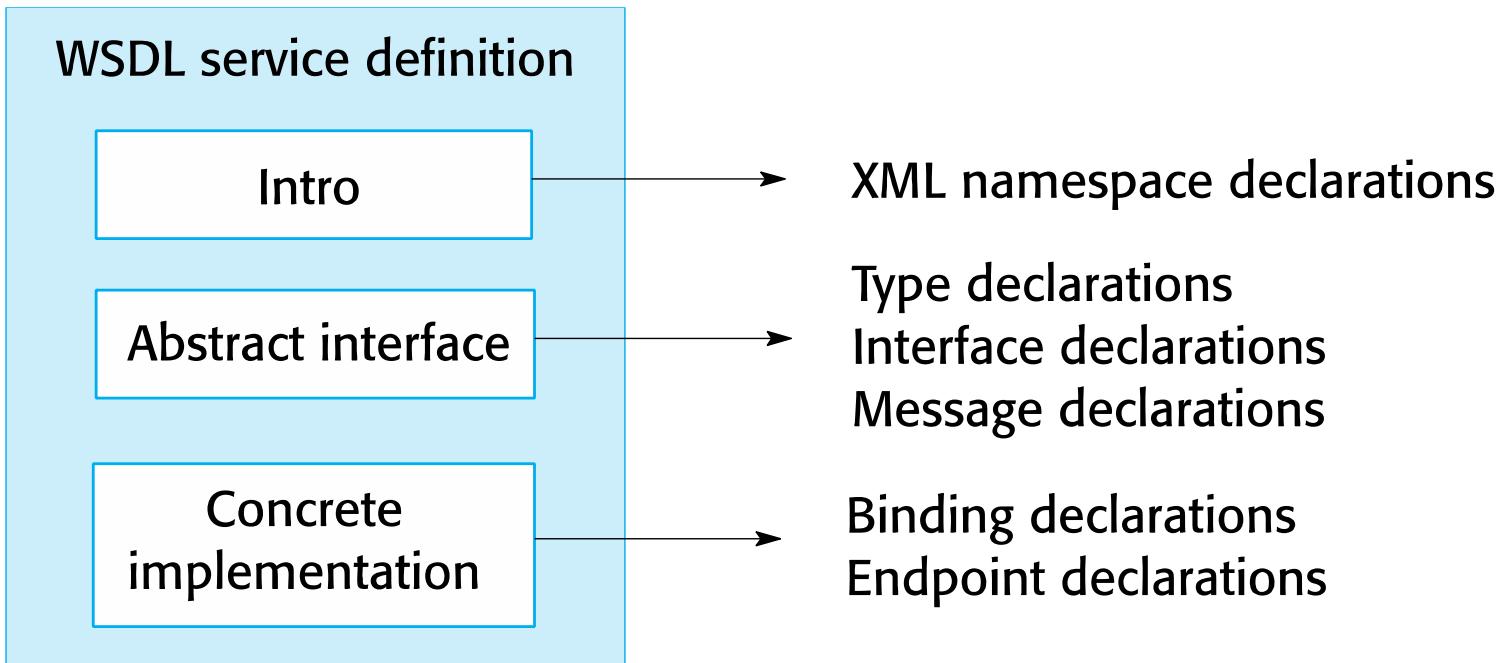
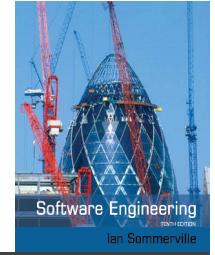
- ✧ A service can be defined as:
 - *A loosely-coupled, reusable software component that encapsulates discrete functionality which may be distributed and programmatically accessed. A web service is a service that is accessed using standard Internet and XML-based protocols*
- ✧ A critical distinction between a service and a component as defined in CBSE is that services are independent
 - Services do not have a 'requires' interface
 - Services rely on message-based communication with messages expressed in XML

Web service description language



- ✧ The service interface is defined in a service description expressed in WSDL (Web Service Description Language).
- ✧ The WSDL specification defines
 - What operations the service supports and the format of the messages that are sent and received by the service
 - How the service is accessed - that is, the binding maps the abstract interface onto a concrete set of protocols
 - Where the service is located. This is usually expressed as a URI (Universal Resource Identifier)

Organization of a WSDL specification





WSDL specification components

- ✧ The ‘what’ part of a WSDL document, called an interface, specifies what operations the service supports, and defines the format of the messages that are sent and received by the service.
- ✧ The ‘how’ part of a WSDL document, called a binding, maps the abstract interface to a concrete set of protocols. The binding specifies the technical details of how to communicate with a Web service.
- ✧ The ‘where’ part of a WSDL document describes the location of a specific Web service implementation (its endpoint).

Part of a WSDL description for a web service



Define some of the types used. Assume that the namespace prefixes ‘ws’ refers to the namespace URI for XML schemas and the namespace prefix associated with this definition is weathns.

```
<types>
  <xs: schema targetNamespace = “http://.../weathns”
    xmlns: weathns = “http://.../weathns” >
    <xs:element name = “PlaceAndDate” type = “pdrec” />
    <xs:element name = “MaxMinTemp” type = “mmtrec” />
    <xs: element name = “InDataFault” type = “errmess” />

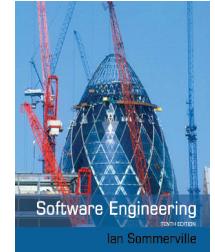
    <xs: complexType name = “pdrec”
      <xs: sequence>
        <xs:element name = “town” type = “xs:string”/>
        <xs:element name = “country” type = “xs:string”/>
        <xs:element name = “day” type = “xs:date” />
      </xs:complexType>
```

Definitions of MaxMinType and InDataFault here

```
  </schema>
```

```
</types>
```

Part of a WSDL description for a web service



Now define the interface and its operations. In this case, there is only a single operation to return maximum and minimum temperatures.

```
<interface name = "weatherInfo" >
  <operation name = "getMaxMinTemps" pattern = "wsdlIns: in-out">
    <input messageLabel = "In" element = "weathns: PlaceAndDate" />
    <output messageLabel = "Out" element = "weathns:MaxMinTemp" />
    <outfault messageLabel = "Out" element = "weathns:InDataFault" />
  </operation>
</interface>
```

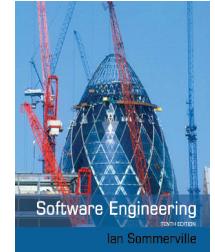


Software Engineering

Ian Sommerville

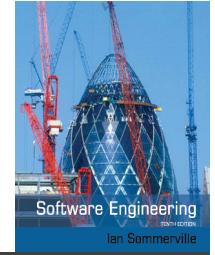
RESTful services

RESTful web services



- ✧ Current web services standards have been criticized as ‘heavyweight’ standards that are over-general and inefficient.
- ✧ REST (REpresentational State Transfer) is an architectural style based on transferring representations of resources from a server to a client.
- ✧ This style underlies the web as a whole and is simpler than SOAP/WSDL for implementing web services.
- ✧ RESTful services involve a lower overhead than so-called ‘big web services’ and are used by many organizations implementing service-based systems.

Resources



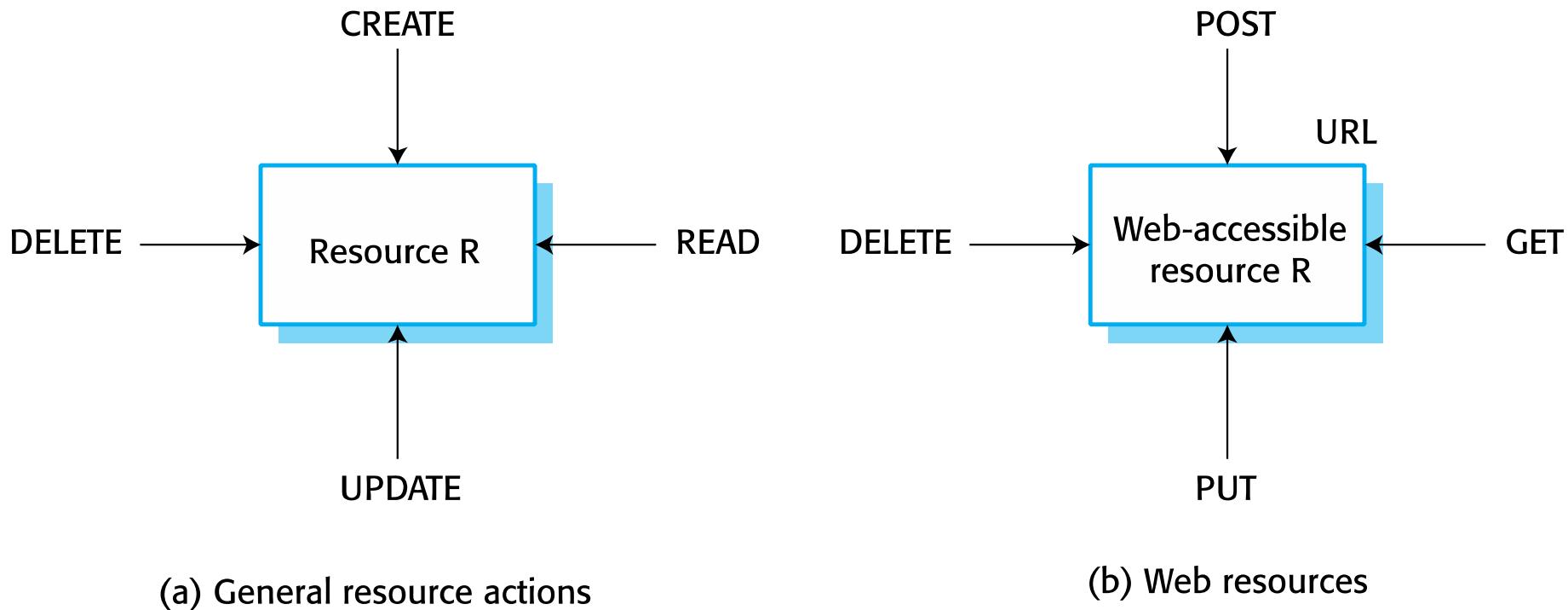
- ✧ The fundamental element in a RESTful architecture is a resource.
- ✧ Essentially, a resource is simply a data element such as a catalog, a medical record, or a document, such as this book chapter.
- ✧ In general, resources may have multiple representations i.e. they can exist in different formats.
 - MS WORD
 - PDF
 - Quark XPress

Resource operations

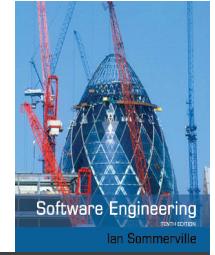


- ✧ Create – bring the resource into existence.
- ✧ Read – return a representation of the resource.
- ✧ Update – change the value of the resource.
- ✧ Delete – make the resource inaccessible.

Resources and actions

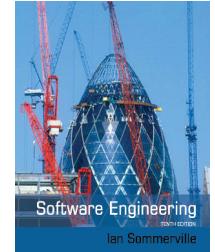


Operation functionality



- ✧ POST is used to create a resource. It has associated data that defines the resource.
- ✧ GET is used to read the value of a resource and return that to the requestor in the specified representation, such as XHTML, that can be rendered in a web browser.
- ✧ PUT is used to update the value of a resource.
- ✧ DELETE is used to delete the resource.

Resource access



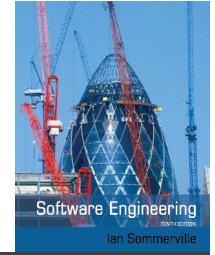
- ✧ When a RESTful approach is used, the data is exposed and is accessed using its URL.
- ✧ Therefore, the weather data for each place in the database, might be accessed using URLs such as:
 - <http://weather-info-example.net/temperatures/boston>
<http://weather-info-example.net/temperatures/edinburgh>
- ✧ Invokes the GET operation and returns a list of maximum and minimum temperatures.
- ✧ To request the temperatures for a specific date, a URL query is used:
 - <http://weather-info-example.net/temperatures/edinburgh?date=20140226>



Query results

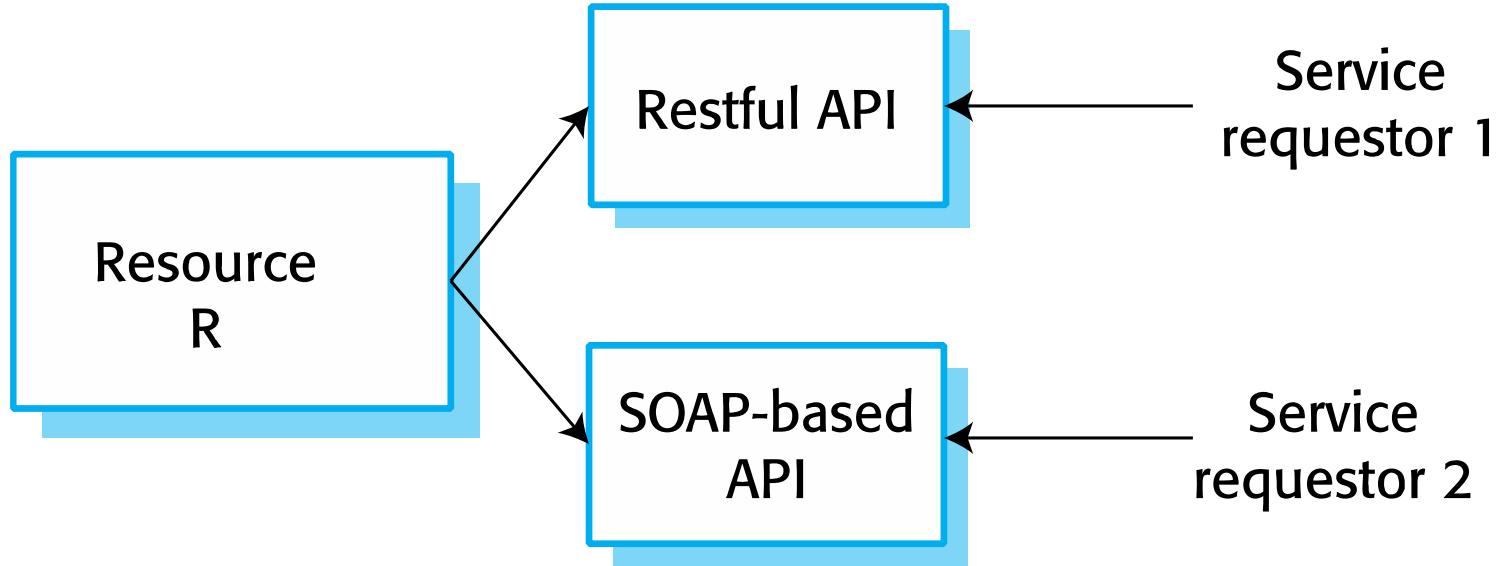
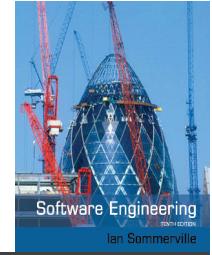
- ✧ The response to a GET request in a RESTful service may include URLs.
- ✧ If the response to a request is a set of resources, then the URL of each of these may be included.
 - <http://weather-info-example.net/temperatures/edinburgh-scotland>
 - <http://weather-info-example.net/temperatures/edinburgh-australia>
 - <http://weather-info-example.net/temperatures/edinburgh-maryland>

Disadvantages of RESTful approach



- ✧ When a service has a complex interface and is not a simple resource, it can be difficult to design a set of RESTful services to represent this.
- ✧ There are no standards for RESTful interface description so service users must rely on informal documentation to understand the interface.
- ✧ When you use RESTful services, you have to implement your own infrastructure for monitoring and managing the quality of service and the service reliability.

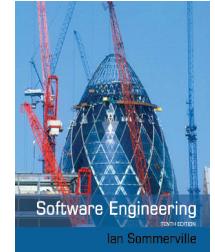
RESTful and SOAP-based APIs





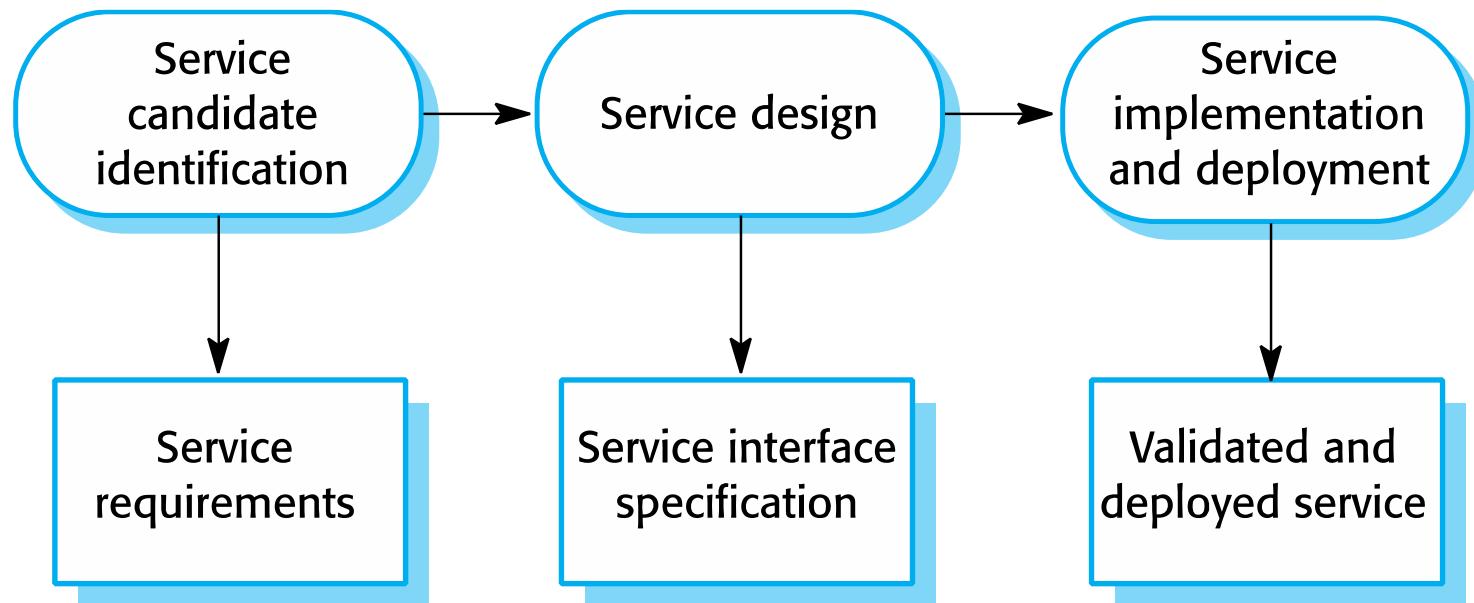
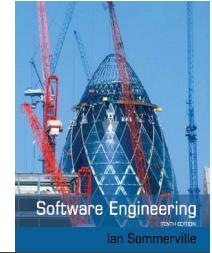
Service engineering

Service engineering

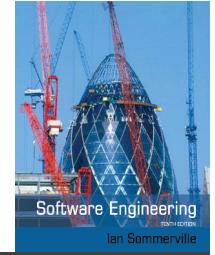


- ✧ The process of developing services for reuse in service-oriented applications
- ✧ The service has to be designed as a reusable abstraction that can be used in different systems.
- ✧ Generally useful functionality associated with that abstraction must be designed and the service must be robust and reliable.
- ✧ The service must be documented so that it can be discovered and understood by potential users.

The service engineering process



Stages of service engineering



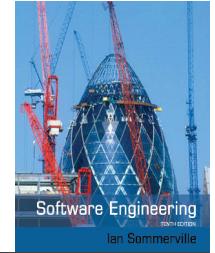
- ✧ Service candidate identification, where you identify possible services that might be implemented and define the service requirements.
- ✧ Service design, where you design the logical service interface and its implementation interfaces (SOAP and/or RESTful)
- ✧ Service implementation and deployment, where you implement and test the service and make it available for use.

Service candidate identification



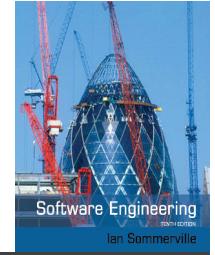
- ✧ Services should support business processes.
- ✧ Service candidate identification involves understanding an organization's business processes to decide which reusable services could support these processes.
- ✧ Three fundamental types of service
 - Utility services that implement general functionality used by different business processes.
 - Business services that are associated with a specific business function e.g., in a university, student registration.
 - Coordination services that support composite processes such as ordering.

Task and entity-oriented services



- ✧ Task-oriented services are those associated with some activity.
- ✧ Entity-oriented services are like objects. They are associated with a business entity such as a job application form.
- ✧ Utility or business services may be entity- or task-oriented, coordination services are always task-oriented.

Service classification



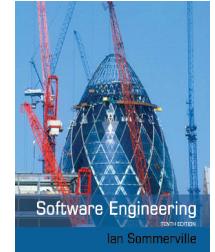
	Utility	Business	Coordination
Task	Currency converter Employee locator	Validate claim form Check credit rating	Process expense claim Pay external supplier
Entity	Document style checker Web form to XML converter	Expenses form Student application form	

Service identification



- ✧ Is the service associated with a single logical entity used in different business processes?
- ✧ Is the task one that is carried out by different people in the organisation? Can this fit with a RESTful model?
- ✧ Is the service independent?
- ✧ Does the service have to maintain state? Is a database required?
- ✧ Could the service be used by clients outside the organisation?
- ✧ Are different users of the service likely to have different non-functional requirements?

Service identification example



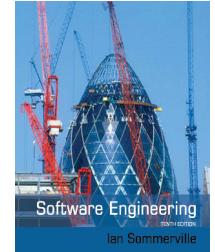
- ✧ A large company, which sells computer equipment, has arranged special prices for approved configurations for some customers.
- ✧ To facilitate automated ordering, the company wishes to produce a catalog service that will allow customers to select the equipment that they need.
- ✧ Unlike a consumer catalog, orders are not placed directly through a catalog interface. Instead, goods are ordered through the web-based procurement system of each company that accesses the catalog as a web service.
- ✧ Most companies have their own budgeting and approval procedures for orders and their own ordering process must be followed when an order is placed.



Catalog services

- ✧ Created by a supplier to show which good can be ordered from them by other companies
- ✧ Service requirements
 - Specific version of catalogue should be created for each client
 - Catalogue shall be downloadable
 - The specification and prices of up to 6 items may be compared
 - Browsing and searching facilities shall be provided
 - A function shall be provided that allows the delivery date for ordered items to be predicted
 - Virtual orders shall be supported which reserve the goods for 48 hours to allow a company order to be placed

Catalogue: Non-functional requirements



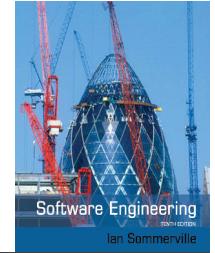
- ✧ Access shall be restricted to employees of accredited organisations
- ✧ Prices and configurations offered to each organisation shall be confidential
- ✧ The catalogue shall be available from 0700 to 1100
- ✧ The catalogue shall be able to process up to 10 requests per second

Functional descriptions of catalog service operations



Operation	Description
MakeCatalog	Creates a version of the catalog tailored for a specific customer. Includes an optional parameter to create a downloadable PDF version of the catalog.
Lookup	Displays all of the data associated with a specified catalog item.
Search	This operation takes a logical expression and searches the catalog according to that expression. It displays a list of all items that match the search expression.

Functional descriptions of catalog service operations



Operation	Description
Compare	Provides a comparison of up to six characteristics (e.g., price, dimensions, processor speed, etc.) of up to four catalog items.
CheckDelivery	Returns the predicted delivery date for an item if ordered that day.
MakeVirtualOrder	Reserves the number of items to be ordered by a customer and provides item information for the customer's own procurement system.

Service interface design



- ✧ Involves thinking about the operations associated with the service and the messages exchanged
- ✧ The number of messages exchanged to complete a service request should normally be minimised.
- ✧ Service state information may have to be included in messages



Interface design stages

Software Engineering
Ian Sommerville

✧ Logical interface design

- Starts with the service requirements and defines the operation names and parameters associated with the service. Exceptions should also be defined

✧ Message design (SOAP)

- For SOAP-based services, design the structure and organisation of the input and output messages. Notations such as the UML are a more abstract representation than XML
- The logical specification is converted to a WSDL description

✧ Interface design (REST)

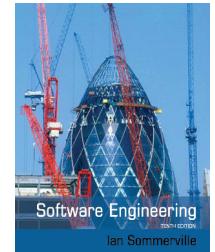
- Design how the required operations map onto REST operations and what resources are required.

Catalog interface design

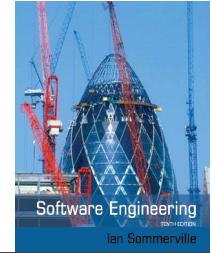


Operation	Inputs	Outputs	Exceptions
MakeCatalog	$mcln$ Company id PDF-flag	$mcOut$ URL of the catalog for that company	$mcFault$ Invalid company id
Lookup	$lookIn$ Catalog URL Catalog number	$lookOut$ URL of page with the item information	$lookFault$ Invalid catalog number
Search	$searchIn$ Catalog URL Search string	$searchOut$ URL of web page with search results	$searchFault$ Badly formed search string

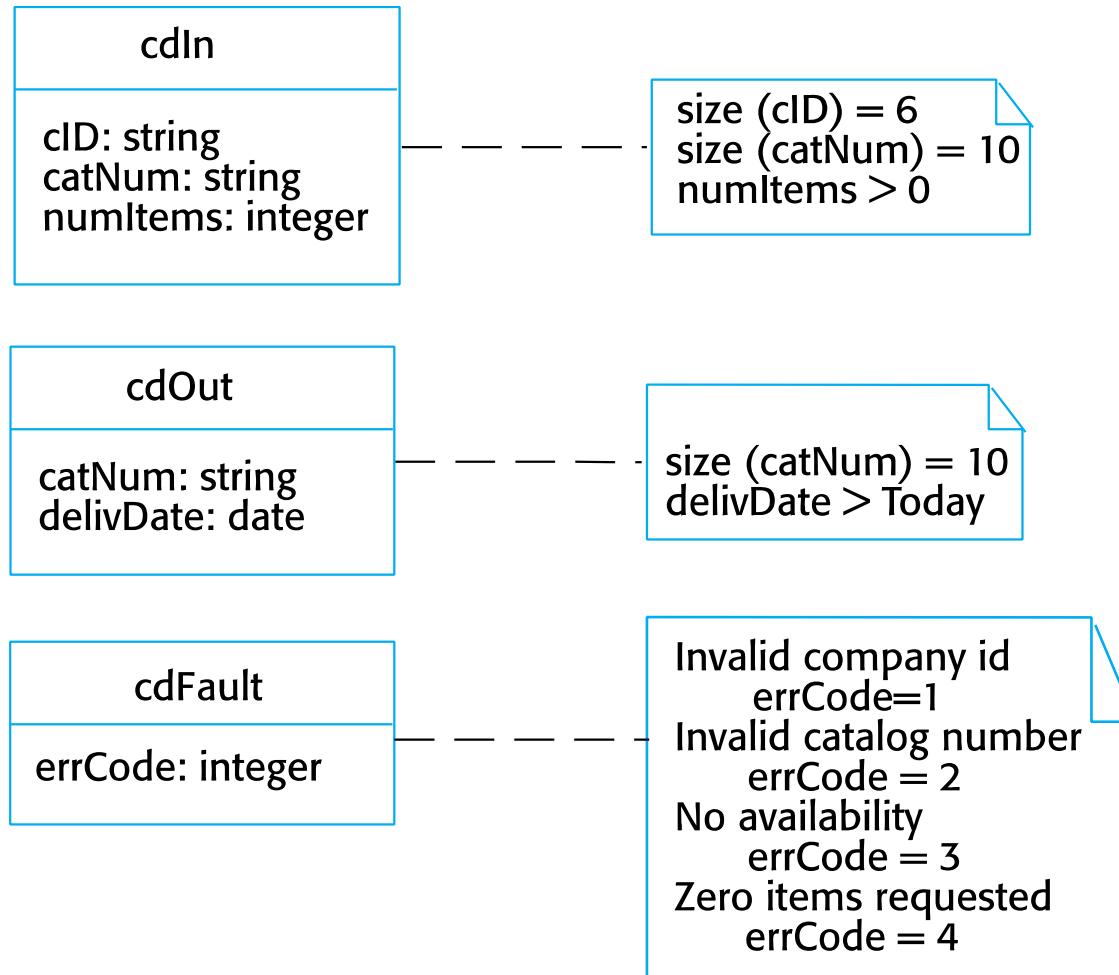
Catalog interface design



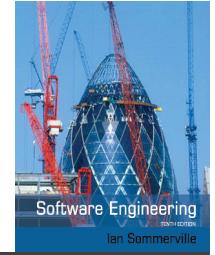
Operation	Inputs	Outputs	Exceptions
Compare	$compIn$ Catalog URL Entry attribute (up to 6) Catalog number (up to 4)	$compOut$ URL of page showing comparison table	$compFault$ Invalid company id Invalid catalog number Unknown attribute
CheckDelivery	$cdIn$ Company id Catalog number Number of items required	$cdOut$ Catalog number Expected delivery date	$cdFault$ Invalid company id No availability Zero items requested
MakeVirtualOrder	$poIn$ Company id Number of items required Catalog number	$poOut$ Catalog number Number of items required Predicted delivery date Unit price estimate Total price estimate	$poFault$ Invalid company id Invalid catalog number Zero items requested



UML definition of input and output messages



RESTful interface



- ✧ There should be a resource representing a company-specific catalog. This should have a URL of the form <base catalog>/<company name> and should be created using a POST operation.
- ✧ Each catalog item should have its own URL of the form:
 - <base catalog>/<company name>/<item identifier>.
- ✧ The GET operation is used to retrieve items.
 - **Lookup** is implemented by using the URL of an item in a catalog as the GET parameter.
 - **Search** is implemented by using GET with the company catalog as the URL and the search string as a query parameter. This GET operation returns a list of URLs of the items matching the search.

RESTful interface



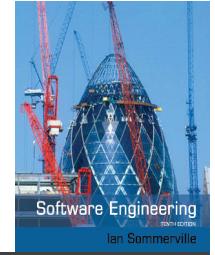
- ✧ The **Compare** operation can be implemented as a sequence of GET operations, to retrieve the individual items, followed by a POST operation to create the comparison table and a final GET operation to return this to the user.
- ✧ The **CheckDelivery** and **MakeVirtualOrder** operations require an additional resource, representing a virtual order.
 - A POST operation is used to create this resource with the number of items required. The company id is used to automatically fill in the order form and the delivery date is calculated. This can then be retrieved using a GET operation.

Service implementation and deployment



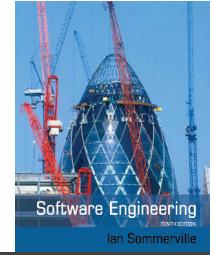
- ✧ Programming services using a standard programming language or a workflow language
- ✧ Services then have to be tested by creating input messages and checking that the output messages produced are as expected
- ✧ Deployment involves publicising the service and installing it on a web server. Current servers provide support for service installation

Legacy system services



- ✧ Services can be implemented by implementing a service interface to existing legacy systems
- ✧ Legacy systems offer extensive functionality and this can reduce the cost of service implementation
- ✧ External applications can access this functionality through the service interfaces

Service descriptions

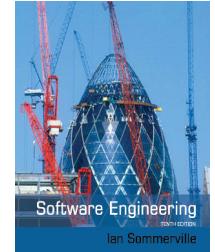


- ✧ Information about your business, contact details, etc.
This is important for trust reasons. Users of a service have to be confident that it will not behave maliciously.
- ✧ An informal description of the functionality provided by the service. This helps potential users to decide if the service is what they want.
- ✧ A description of how to use the services SOAP-based and RESTful.
- ✧ Subscription information that allows users to register for information about updates to the service.



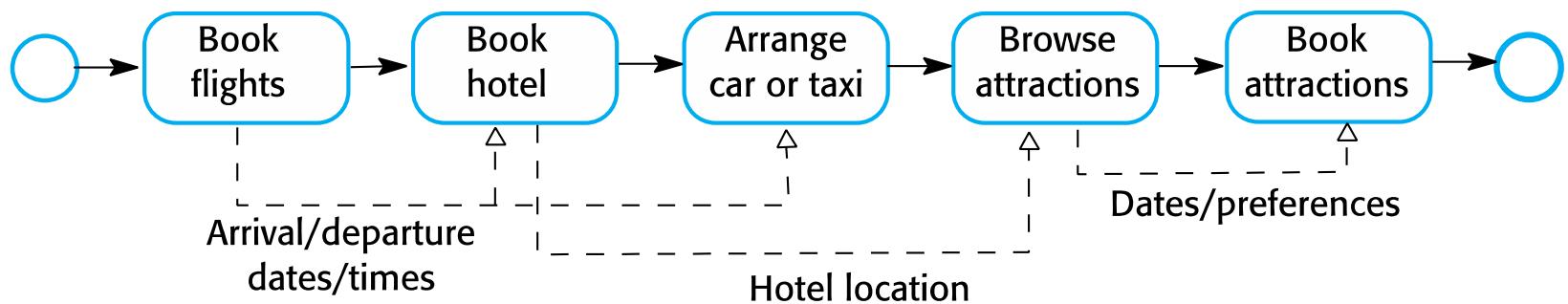
Service composition

Software development with services



- ✧ Existing services are composed and configured to create new composite services and applications
- ✧ The basis for service composition is often a workflow
 - Workflows are logical sequences of activities that, together, model a coherent business process
 - For example, provide a travel reservation services which allows flights, car hire and hotel bookings to be coordinated

Vacation package workflow

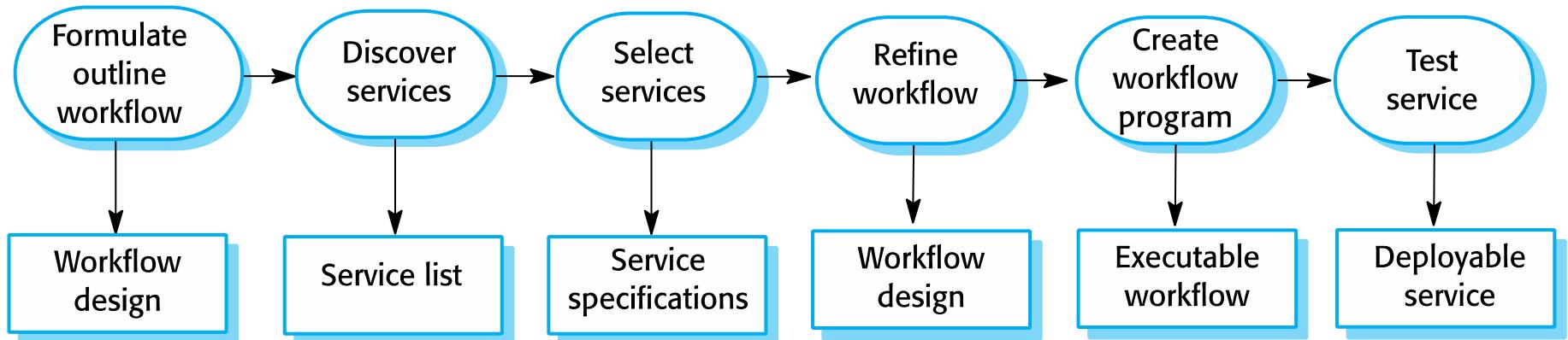


Service construction by composition



Software Engineering

Ian Sommerville



Construction by composition



✧ *Formulate outline workflow*

- In this initial stage of service design, you use the requirements for the composite service as a basis for creating an 'ideal' service design.

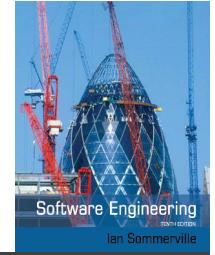
✧ *Discover services*

- During this stage of the process, you search service registries or catalogs to discover what services exist, who provides these services and the details of the service provision.

✧ *Select possible services*

- Your selection criteria will obviously include the functionality of the services offered. They may also include the cost of the services and the quality of service (responsiveness, availability, etc.) offered.

Construction by composition



✧ *Refine workflow.*

- This involves adding detail to the abstract description and perhaps adding or removing workflow activities.

✧ *Create workflow program*

- During this stage, the abstract workflow design is transformed to an executable program and the service interface is defined. You can use a conventional programming language, such as Java or a workflow language, such as WS-BPEL.

✧ *Test completed service or application*

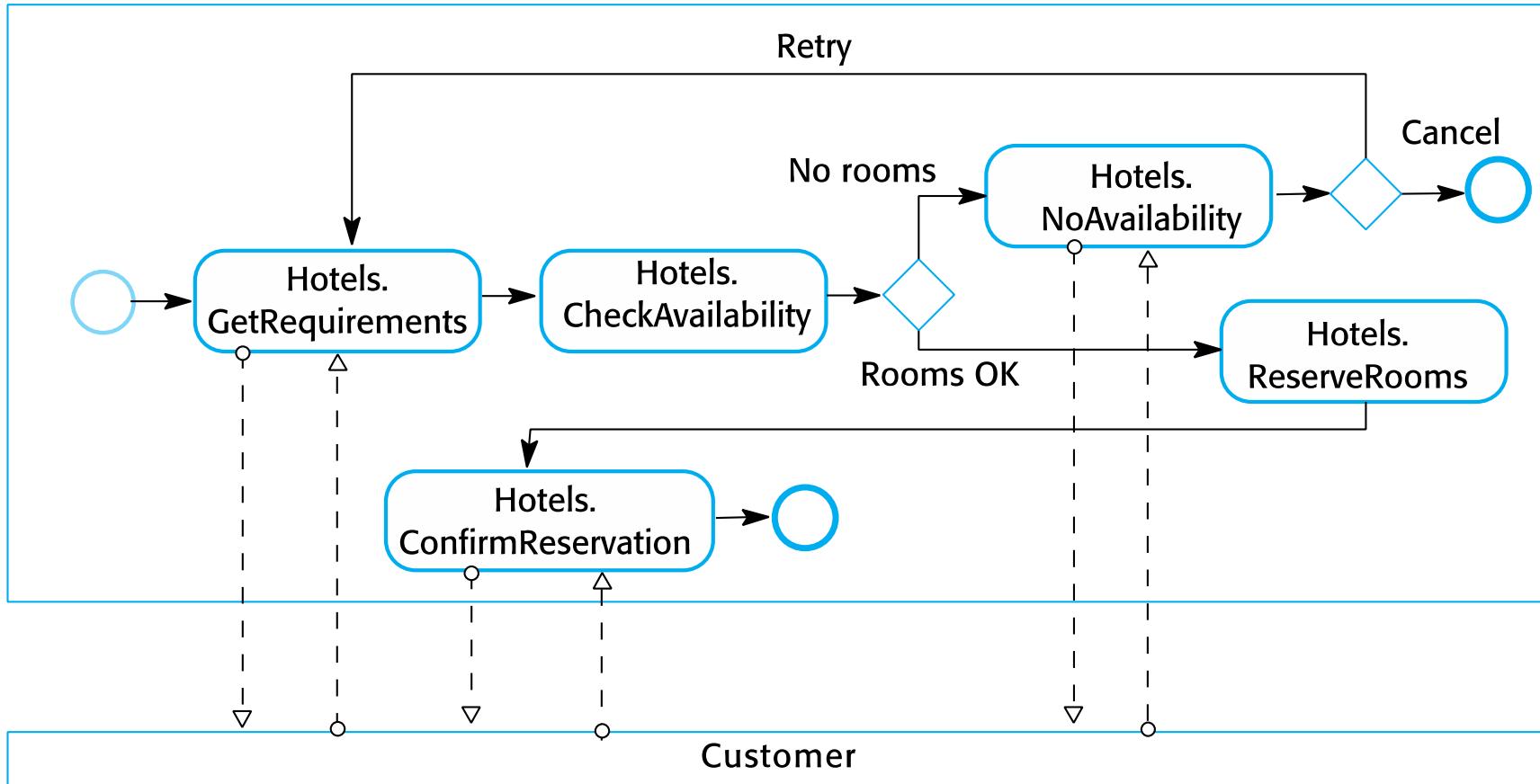
- The process of testing the completed, composite service is more complex than component testing in situations where external services are used.

Workflow design and implementation

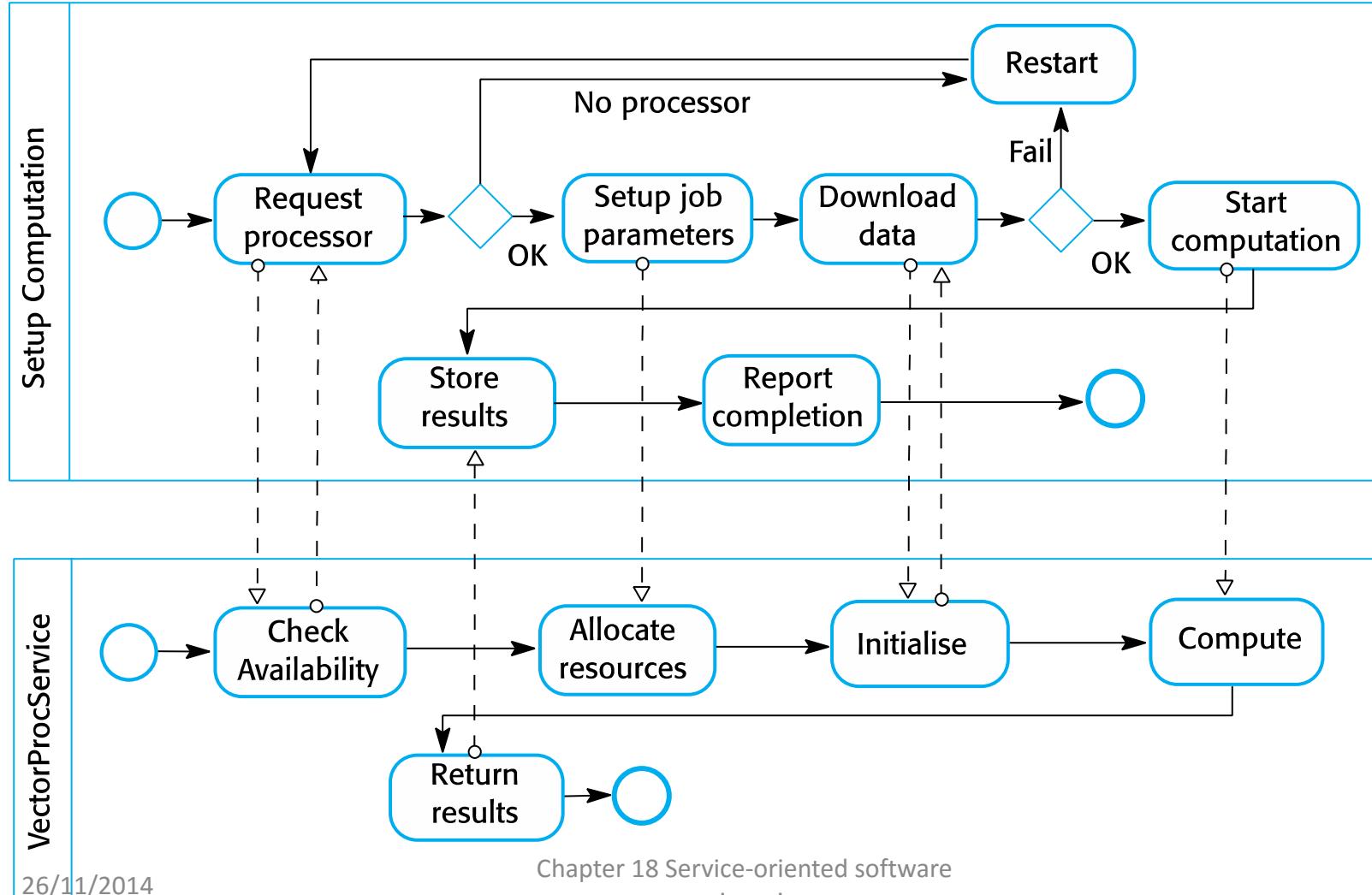
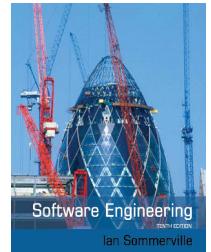


- ✧ WS-BPEL is an XML-standard for workflow specification. However, WS-BPEL descriptions are long and unreadable
- ✧ Graphical workflow notations, such as BPMN, are more readable and WS-BPEL can be generated from them
- ✧ In inter-organisational systems, separate workflows are created for each organisation and linked through message exchange.
- ✧ Workflows can be used with both SOAP-based and RESTful services.

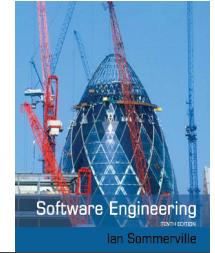
A fragment of a hotel booking workflow



Interacting workflows

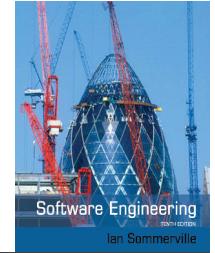


Testing service compositions

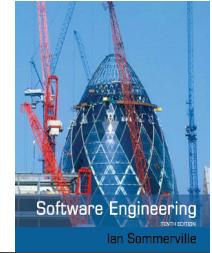


- ✧ Testing is intended to find defects and demonstrate that a system meets its functional and non-functional requirements.
- ✧ Service testing is difficult as (external) services are ‘black-boxes’. Testing techniques that rely on the program source code cannot be used.

Service testing problems



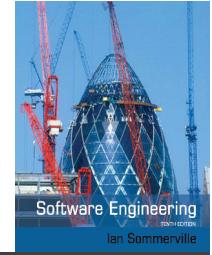
- ✧ External services may be modified by the service provider thus invalidating tests which have been completed.
- ✧ Dynamic binding means that the service used in an application may vary - the application tests are not, therefore, reliable.
- ✧ The non-functional behaviour of the service is unpredictable because it depends on load.
- ✧ If services have to be paid for as used, testing a service may be expensive.
- ✧ It may be difficult to invoke compensating actions in external services as these may rely on the failure of other services which cannot be simulated.



Key points

- ✧ Service-oriented architecture is an approach to software engineering where reusable, standardized services are the basic building blocks for application systems.
- ✧ Services may be implemented within a service-oriented architecture using a set of XML-based web service standards. These include standards for service communication, interface definition and service enactment in workflows.
- ✧ Alternatively, a RESTful architecture may be used which is based on resources and standard operations on these resources.
- ✧ A RESTful approach uses the http and https protocols for service communication and maps operations on the standard http verbs POST, GET, PUT and DELETE.

Key points

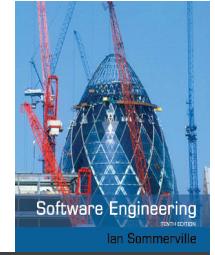


- ✧ Utility services provide general-purpose functionality; business services implement part of a business process; coordination services coordinate service execution.
- ✧ Service engineering involves identifying candidate services for implementation, defining service interfaces and implementing, testing and deploying services.
- ✧ The development of software using services involves composing and configuring services to create new composite services and systems.
- ✧ Graphical workflow languages, such as BPMN, may be used to describe a business process and the services used in that process.



Chapter 19 – Systems Engineering

Topics covered

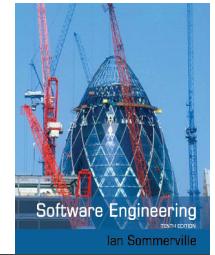


- ✧ Sociotechnical systems
- ✧ Conceptual design
- ✧ Systems procurement
- ✧ System development
- ✧ System operation and evolution

Systems



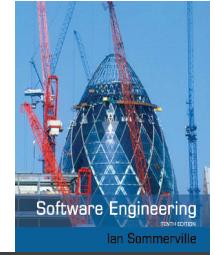
- ✧ Software engineering is not an isolated activity but is part of a broader systems engineering process.
- ✧ Software systems are therefore not isolated systems but are essential components of broader systems that have a human, social or organizational purpose.
- ✧ Example
 - Wilderness weather system is part of broader weather recording and forecasting systems
 - These include hardware and software, forecasting processes, system users, the organizations that depend on weather forecasts, etc.



Types of system

- ✧ Technical computer-based systems
 - Include hardware and software but not humans or organizational processes.
 - Off the shelf applications, control systems, etc.
- ✧ Sociotechnical systems
 - Include technical systems plus people who use and manage these systems and the organizations that own the systems and set policies for their use.
 - Business systems, command and control systems, etc.

Systems engineering



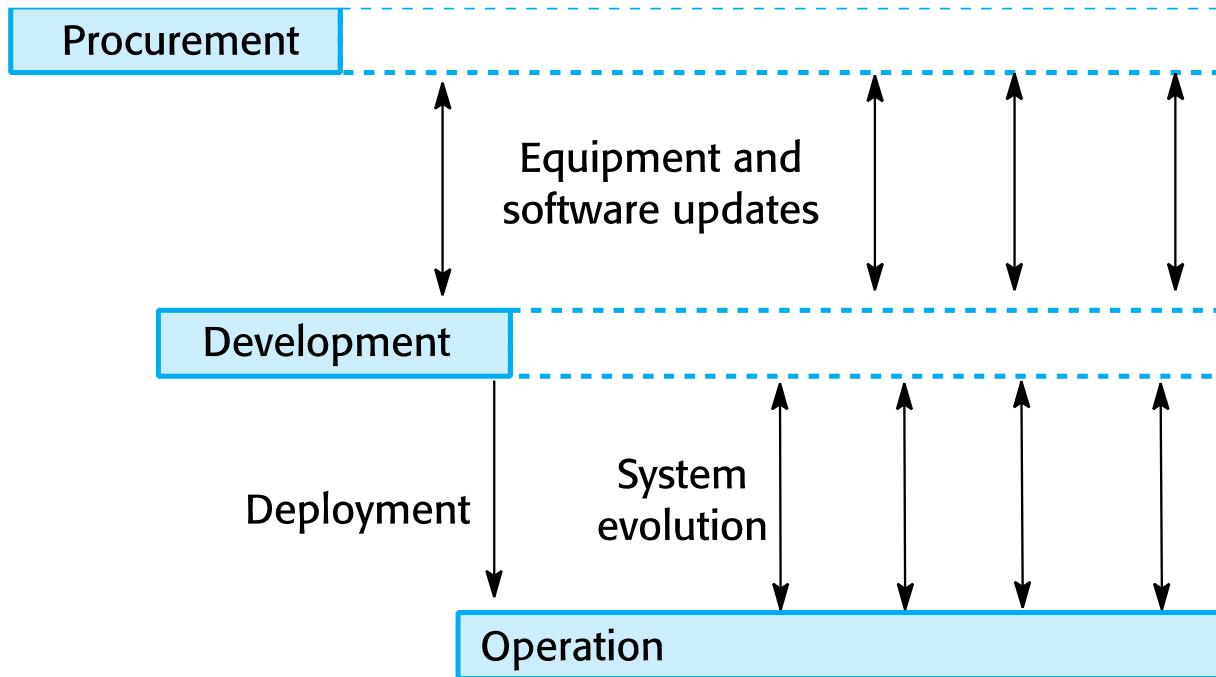
- ✧ Procuring, specifying, designing, implementing, validating, deploying and maintaining sociotechnical systems.
- ✧ Concerned with the services provided by the system, constraints on its construction and operation and the ways in which it is used to fulfil its purpose or purposes.

Systems and software engineering

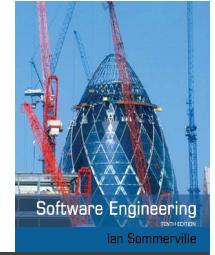


- ✧ Software is now the dominant element in all enterprise systems. Software engineers have to play a more active part in high-level systems decision making if the system software is to be dependable and developed on time and to budget.
- ✧ As a software engineer, it helps if you have a broader awareness of how software interacts with other hardware and software systems, and the human, social and organizational factors that affect the ways in which software is used.

Stages of systems engineering



Systems engineering stages



✧ Conceptual design

- Sets out the purpose of the system, why it is needed and the high-level features that users might expect to see in the system

✧ Procurement or acquisition

- The conceptual design is developed so that decisions about the contract for the system development can be made.

✧ Development

- Hardware and software is engineered and operational processes defined.

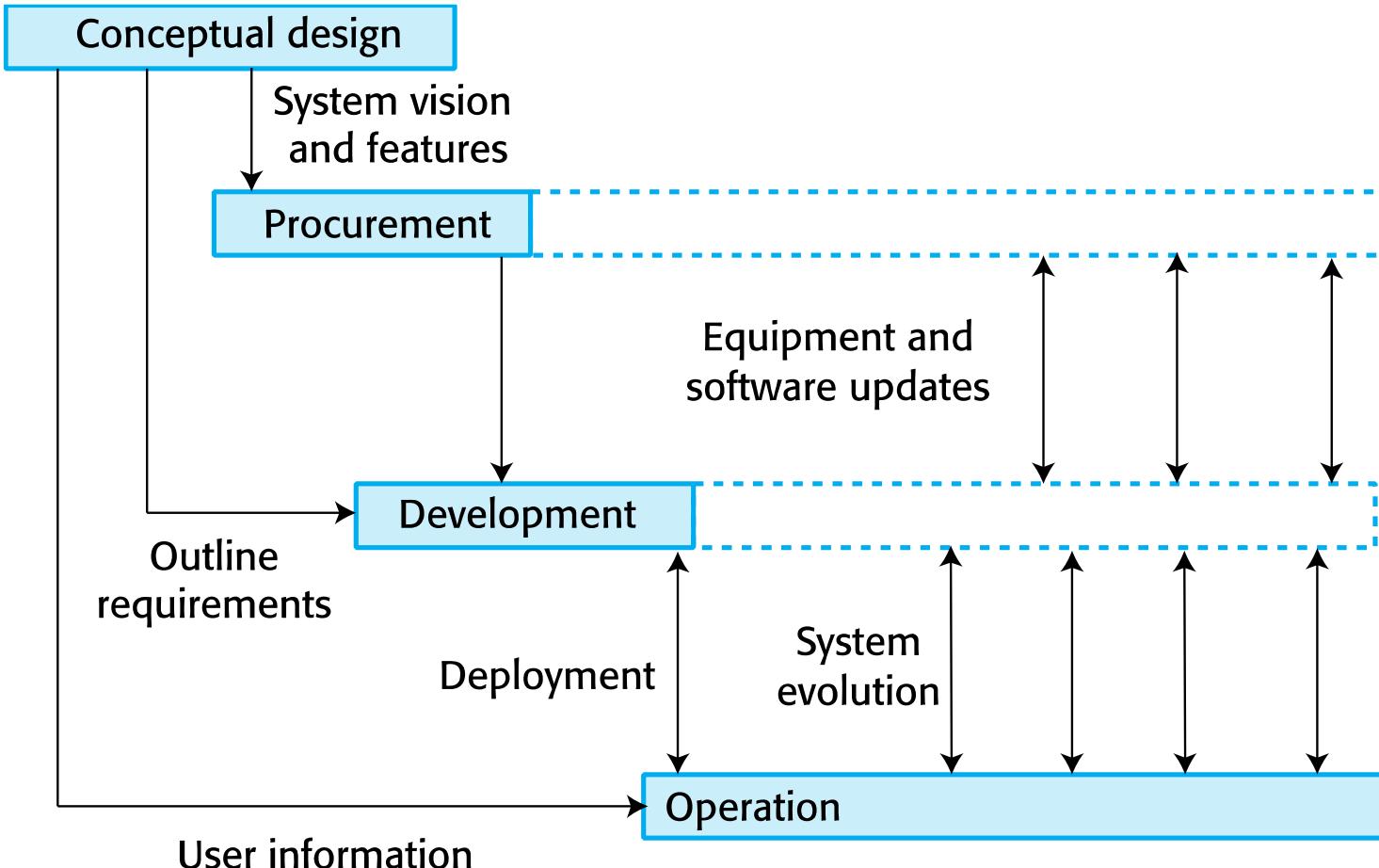
✧ Operation

- The system is deployed and used for its intended purpose.

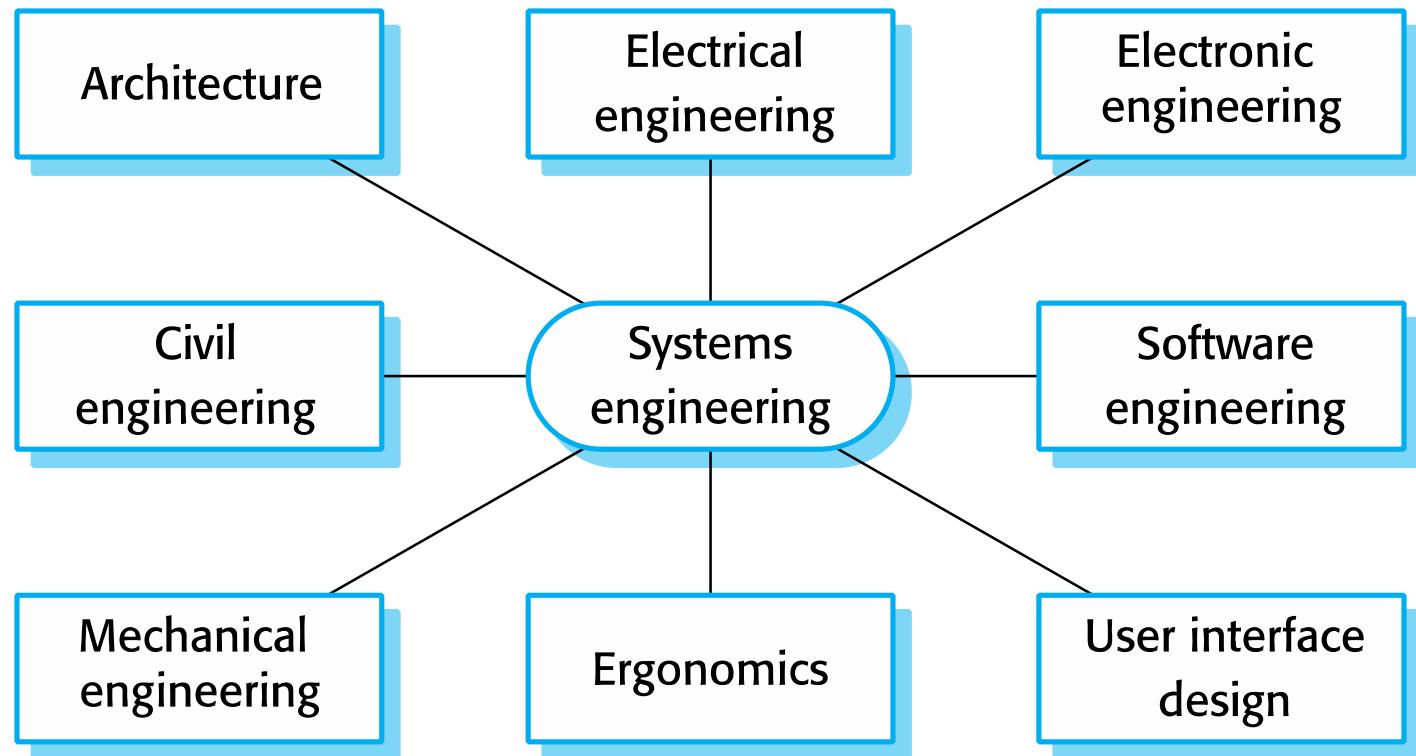
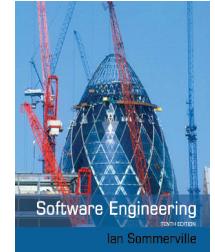
Stages of systems engineering



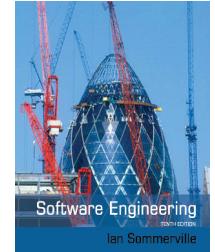
Software Engineering
Ian Sommerville



Professional disciplines involved



Inter-disciplinary working



✧ Communication difficulties

- Different disciplines use the same terminology to mean different things. This can lead to misunderstandings about what will be implemented.

✧ Differing assumptions

- Each discipline makes assumptions about what can and can't be done by other disciplines.

✧ Professional boundaries

- Each discipline tries to protect their professional boundaries and expertise and this affects their judgments on the system.

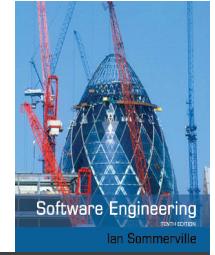


Software Engineering

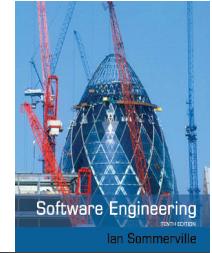
Ian Sommerville

Sociotechnical systems

Sociotechnical systems

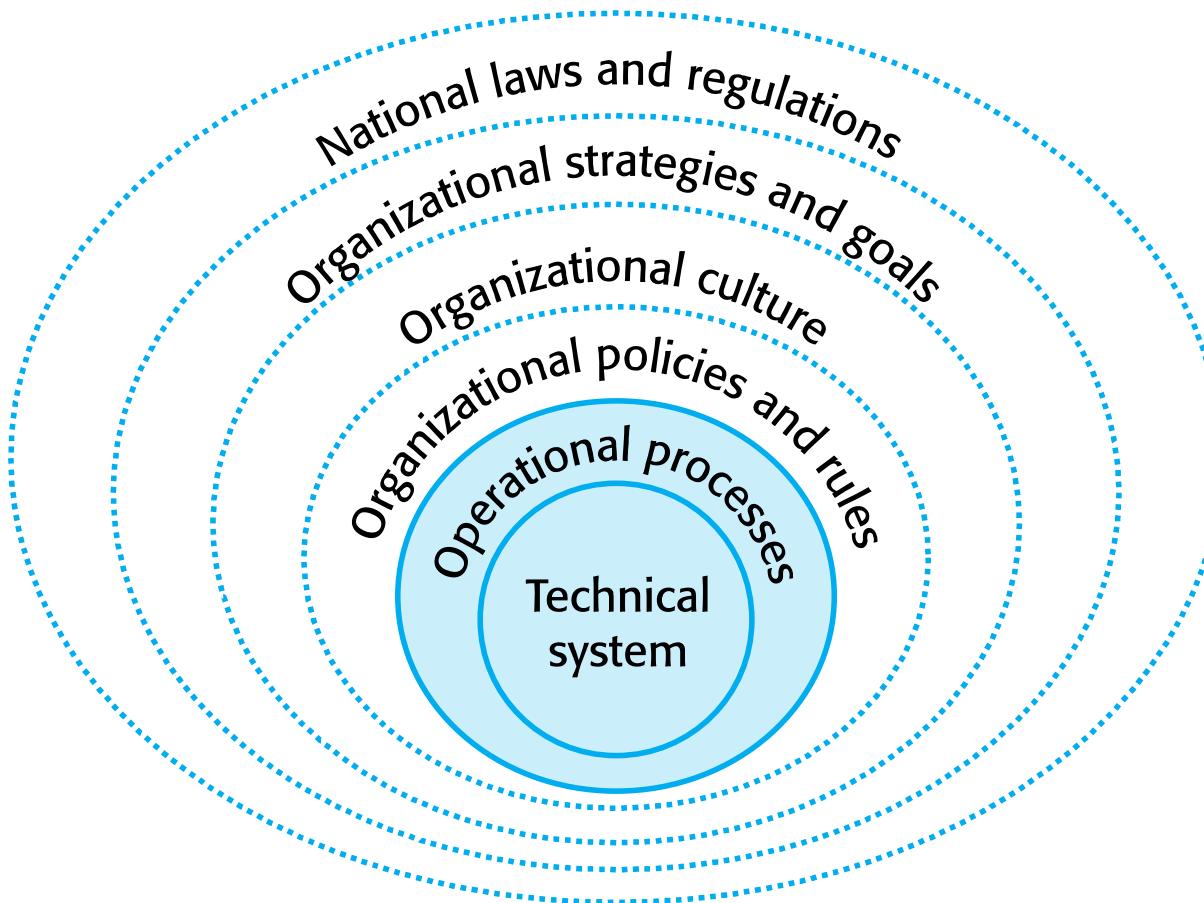


- ✧ Large-scale systems that do not just include software and hardware but also people, processes and organizational policies.
- ✧ Sociotechnical systems are often ‘systems of systems’ i.e. are made up of a number of independent systems.
 - Systems of systems are covered in Chapter 20
- ✧ The boundaries of sociotechnical system are subjective rather than objective
 - Different people see the system in different ways

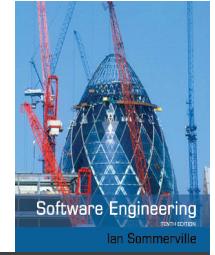


Layered structure of sociotechnical systems

Software Engineering
Ian Sommerville

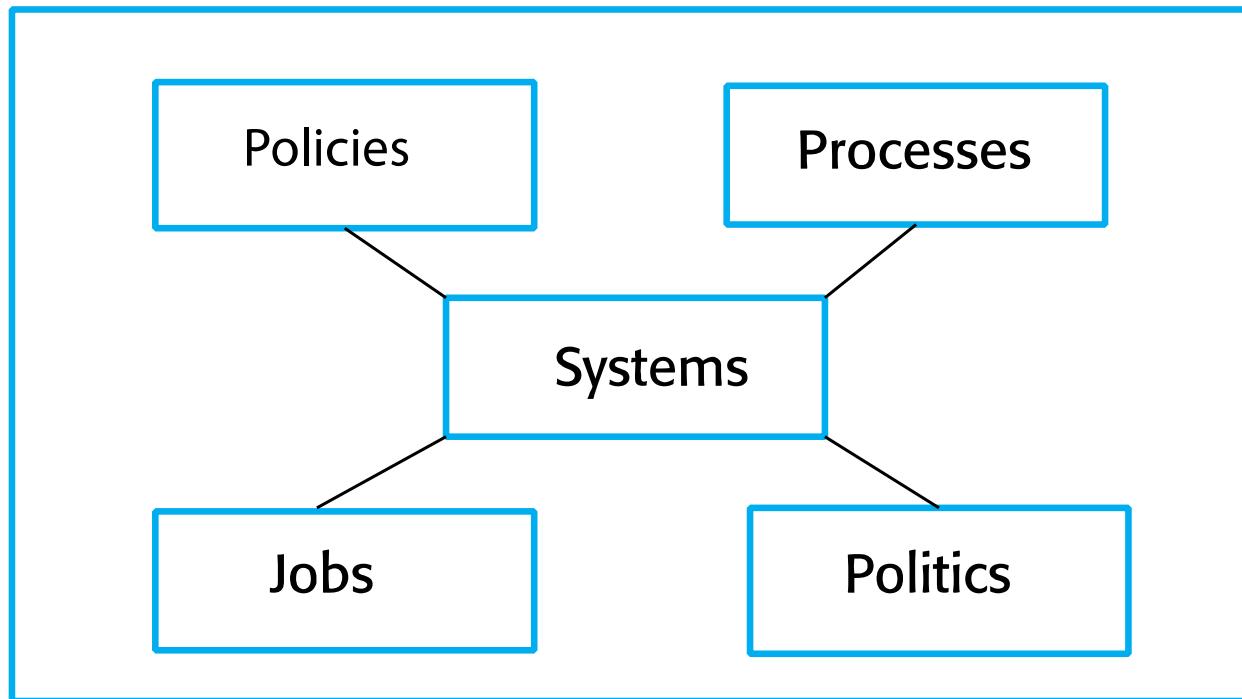


Systems and organizations



- ✧ Sociotechnical systems are used within organizations and are therefore profoundly affected by the organizational environment in which they are used.
- ✧ Failure to take this environment into account when designing the system is likely to lead to user dissatisfaction and system rejection.

Organisational elements



Organizational affects



✧ Process changes

- Systems may require changes to business processes so training may be required. Significant changes may be resisted by users.

✧ Job changes

- Systems may de-skill users or cause changes to the way they work. The status of individuals may be affected by a new system.

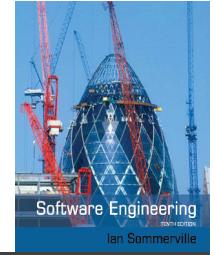
✧ Organizational policies

- The proposed system may not be consistent with current organizational policies.

✧ Organizational politics

- Systems may change the political power structure in an organization. Those that control the system have more power.

Complex systems



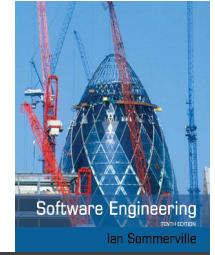
- ✧ A system may include software, mechanical, electrical and electronic hardware and be operated by people.
- ✧ System components are dependent on other system components.
- ✧ The properties and behaviour of system components are inextricably inter-mingled. This leads to complexity.
- ✧ Complexity is the reason why sociotechnical systems have emergent properties, are non-deterministic and have subjective success criteria.

Socio-technical system characteristics



- ✧ Emergent properties
 - Properties of the system of a whole that depend on the system components and their relationships.
- ✧ Non-deterministic
 - They do not always produce the same output when presented with the same input because the system's behaviour is partially dependent on human operators.
- ✧ Complex relationships with organisational objectives
 - The extent to which the system supports organisational objectives does not just depend on the system itself.

Emergent properties



- ✧ Properties of the system as a whole rather than properties that can be derived from the properties of components of a system
- ✧ Emergent properties are a consequence of the relationships between system components
- ✧ They can therefore only be assessed and measured once the components have been integrated into a system

Examples of emergent properties



Software Engineering
Ian Sommerville

Property	Description
Reliability	System reliability depends on component reliability but unexpected interactions can cause new types of failures and therefore affect the reliability of the system.
Repairability	This property reflects how easy it is to fix a problem with the system once it has been discovered. It depends on being able to diagnose the problem, access the components that are faulty, and modify or replace these components.
Security	The security of the system (its ability to resist attack) is a complex property that cannot be easily measured. Attacks may be devised that were not anticipated by the system designers and so may defeat built-in safeguards.
Usability	This property reflects how easy it is to use the system. It depends on the technical system components, its operators, and its operating environment.
Volume	The volume of a system (the total space occupied) varies depending on how the component assemblies are arranged and connected.



Types of emergent property

✧ Functional properties

- These appear when all the parts of a system work together to achieve some objective. For example, a bicycle has the functional property of being a transportation device once it has been assembled from its components.

✧ Non-functional emergent properties

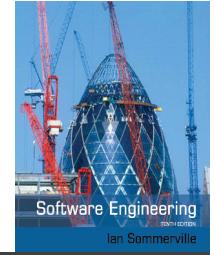
- Examples are reliability, performance, safety, and security. These relate to the behaviour of the system in its operational environment. They are often critical for computer-based systems as failure to achieve some minimal defined level in these properties may make the system unusable.

Reliability as an emergent property



- ✧ Because of component inter-dependencies, faults can be propagated through the system.
- ✧ System failures often occur because of unforeseen inter-relationships between components.
- ✧ It is practically impossible to anticipate all possible component relationships.
- ✧ Software reliability measures may give a false picture of the overall system reliability.

Influences on reliability



✧ *Hardware reliability*

- What is the probability of a hardware component failing and how long does it take to repair that component?

✧ *Software reliability*

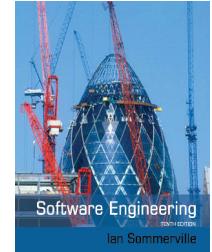
- How likely is it that a software component will produce an incorrect output. Software failure is usually distinct from hardware failure in that software does not wear out.

✧ *Operator reliability*

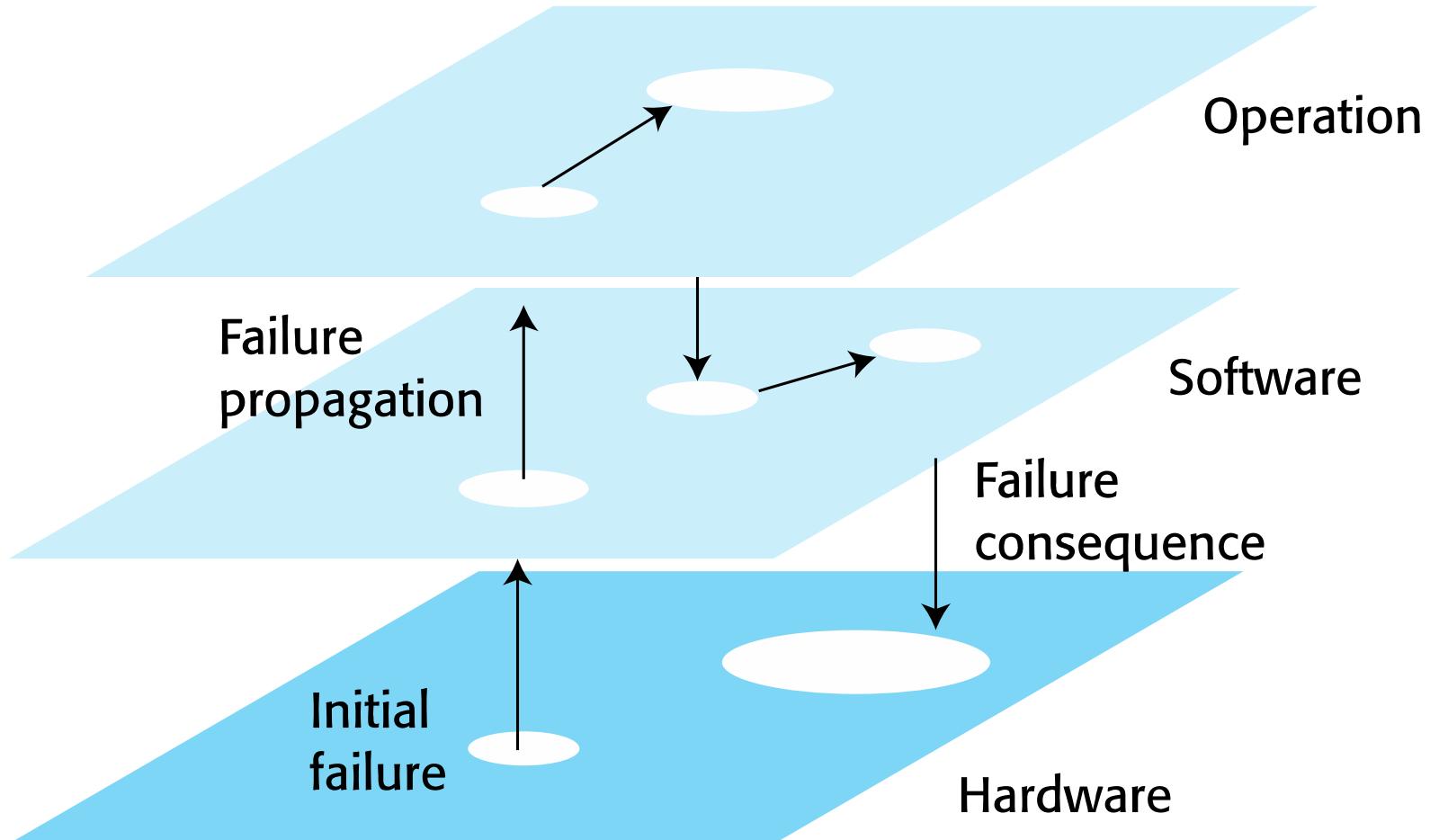
- How likely is it that the operator of a system will make an error?

✧ Failures are not independent and they propagate from one level to another.

Failure propagation



Software Engineering
Ian Sommerville

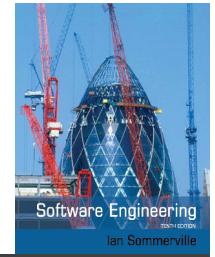


Reliability and system context



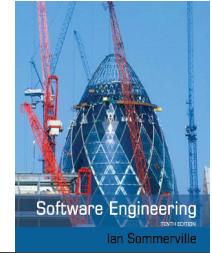
- ✧ System reliability depends on the context where the system is used.
- ✧ A system that is reliable in one environment may be less reliable in a different environment because the physical conditions (e.g. the temperature) and the mode of operation is different.

Non-determinism



Software Engineering
Ian Sommerville

- ✧ A deterministic system is one where a given sequence of inputs will always produce the same sequence of outputs.
- ✧ Software systems are deterministic; systems that include humans are non-deterministic
 - A socio-technical system will not always produce the same sequence of outputs from the same input sequence
 - Human elements
 - People do not always behave in the same way
 - System changes
 - System behaviour is unpredictable because of frequent changes to hardware, software and data.



Success criteria

- ✧ Complex systems are developed to address ‘wicked problems’ – problems where there cannot be a complete specification.
- ✧ Different stakeholders see the problem in different ways and each has a partial understanding of the issues affecting the system.
- ✧ Consequently, different stakeholders have their own views about whether or not a system is ‘successful’
 - Success is a judgment and cannot be objectively measured.
 - Success is judged using the effectiveness of the system when deployed rather than judged against the original reasons for procurement.

Conflicting views of success



- ✧ The Mentcare system is designed to support multiple, conflicting goals
 - Improve quality of care.
 - Provide better information and care costs and so increase revenue.
- ✧ Fundamental conflict
 - Doctors and nurses had to provide additional information over and above that required for clinical purposes.
 - They had less time to interact with patients, so quality of care reduced. System was not a success.
- ✧ However, managers had better reports
 - System was a success from a managerial perspective.



Software Engineering

Ian Sommerville

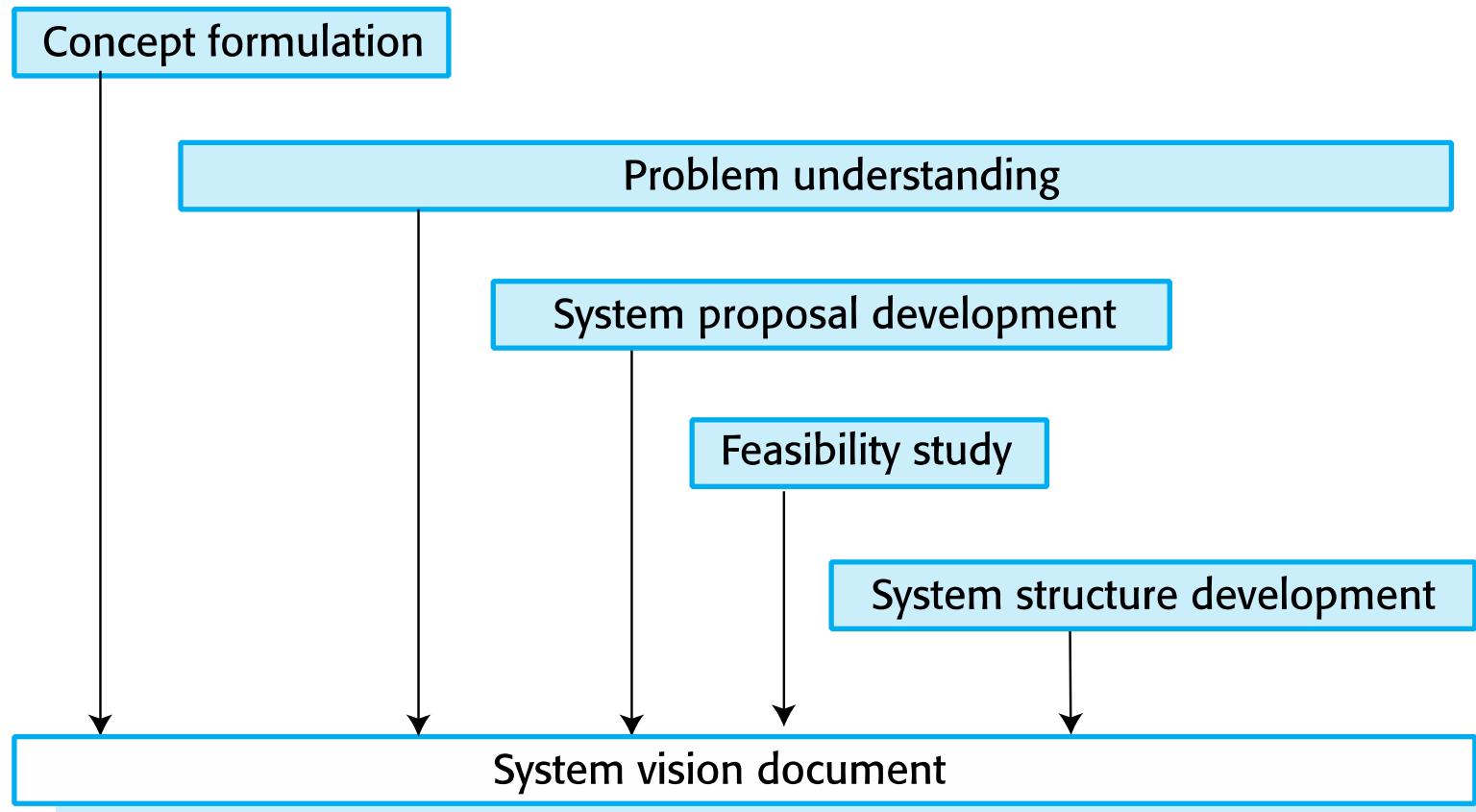
Conceptual design

Conceptual design



- ✧ Investigate the feasibility of an idea and develop that idea to create an overall vision of a system.
- ✧ Conceptual design precedes and overlaps with requirements engineering
 - May involve discussions with users and other stakeholders and the identification of critical requirements
- ✧ The aim of conceptual design is to create a high-level system description that communicates the system purpose to non-technical decision makers.

Conceptual design activities





✧ Concept formulation

- Refine an initial statement of needs and work out what type of system is most likely to meet the needs of system stakeholders

✧ Problem understanding

- Discuss with stakeholders how they do their work, what is and isn't important to them, what they like and don't like about existing systems

✧ System proposal development

- Set out ideas for possible systems (maybe more than one)



✧ Feasibility study

- Look at comparable systems that have been developed elsewhere (if any) and assess whether or not the proposed system could be implemented using current hardware and software technologies

✧ System structure development

- Develop an outline architecture for the system, identifying (where appropriate) other systems that may be reused

✧ System vision document

- Document the results of the conceptual design in a readable, non-technical way. Should include a short summary and more detailed appendices.

User stories for presentation of system vision



Digital art

Jill is an S2 pupil at a secondary school in Dundee. She has a smart phone of her own and the family has a shared Samsung tablet and a Dell laptop computer. At school, Jill signs on to the school computer and is presented with a personalized Glow+ environment, which includes a range of services, some chosen by her teachers and some she has chosen herself from the Glow app library.

She is working on a Celtic art project and she uses Google to research a range of art sites. She sketches out some designs on paper then uses the camera on her phone to photograph what she has done and uploads this using the school wifi to her personal Glow+ space. Her homework is to complete the design and write a short commentary on her ideas.

User stories (2)



At home, she uses the family tablet to sign on to Glow+ and she then uses an artwork 'app' to process her photograph and to extend the work, add colour, etc.

She finishes this and to complete the work she moves to her home laptop to type up her commentary. She uploads the finished work to Glow+ and sends a message to her art teacher that it is available for review. Her teacher looks at this in a free period before Jill's next art class using a school tablet and, in class, discusses the work with Jill.

After the discussion, the teacher and Jill decide that the work should be shared and they publish it to the school web pages that show examples of students' work. In addition, the work is included in Jill's e-portfolio – her record of schoolwork from age 3 to 18.

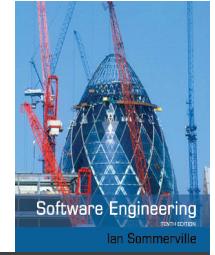


Software Engineering

Ian Sommerville

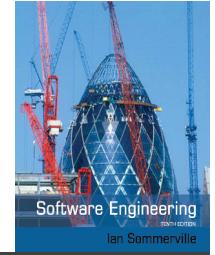
System procurement

System procurement



- ✧ Acquiring a system (or systems) to meet some identified organizational need.
- ✧ Before procurement, decisions are made on:
 - Scope of the system
 - System budgets and timescales
 - High-level system requirements
- ✧ Based on this information, decisions are made on whether to procure a system, the type of system and the potential system suppliers.

Decision drivers



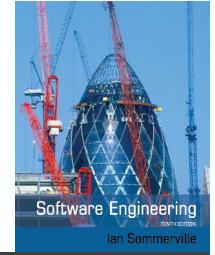
- ✧ The state of other organizational systems and whether or not they need to be replaced
- ✧ The need to comply with external regulations
- ✧ External competition
- ✧ Business re-organization
- ✧ Available budget

Procurement and development



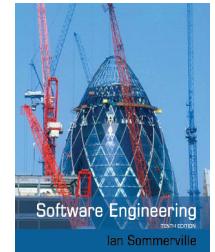
- ✧ It is usually necessary to develop a conceptual design document and high-level requirements before procurement
 - You need a specification to let a contract for system development
 - The specification may allow you to buy a commercial off-the-shelf (COTS) system. Almost always cheaper than developing a system from scratch
- ✧ Large complex systems usually consist of a mix of off the shelf and specially designed components. The procurement processes for these different types of component are usually different.

Types of system

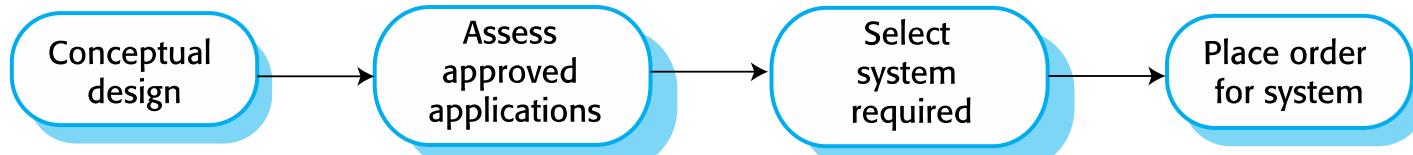


- ✧ Off-the-shelf applications that may be used without change and which need only minimal configuration for use.
- ✧ Configurable application or ERP systems that have to be modified or adapted for use either by modifying the code or by using inbuilt configuration features, such as process definitions and rules.
- ✧ Custom systems that have to be designed and implemented specially for use.

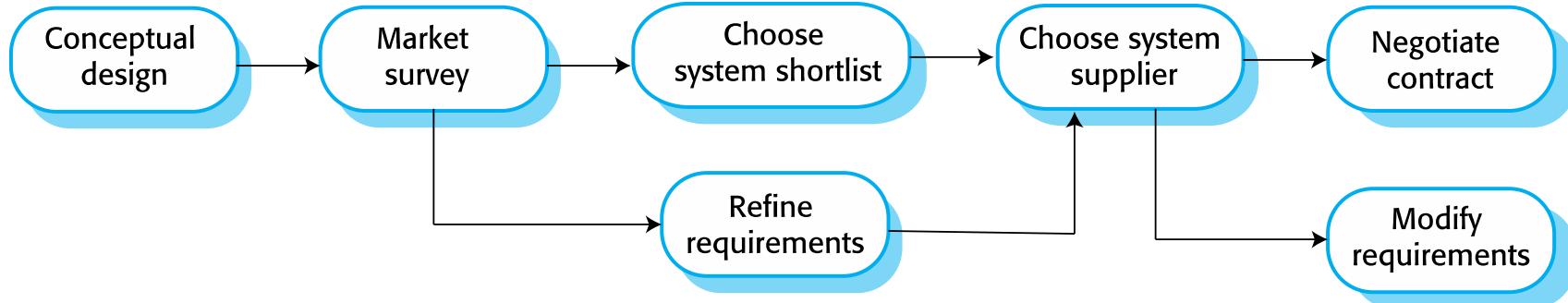
System procurement processes



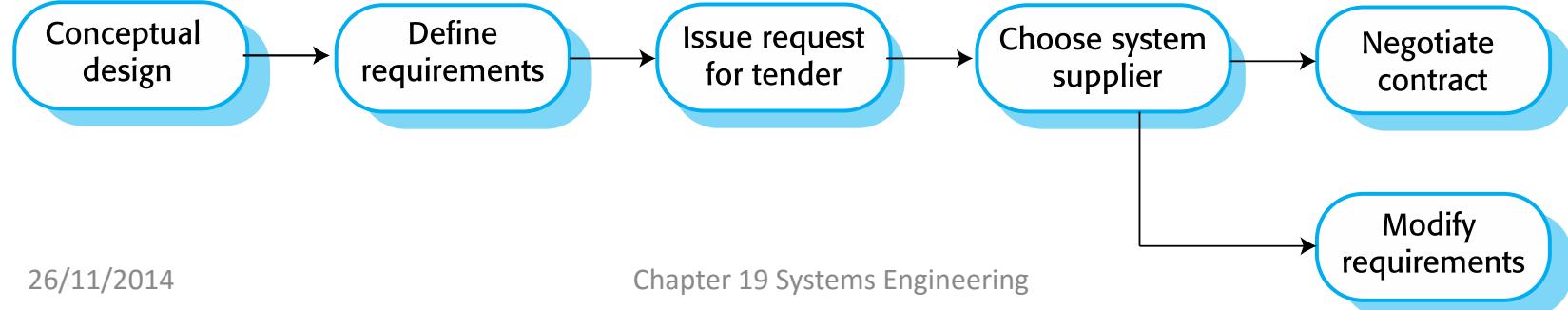
Off-the-shelf systems

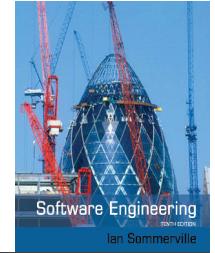


Configurable systems



Custom systems

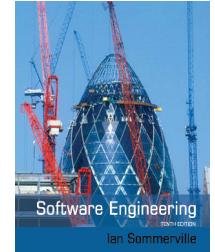




Procurement issues

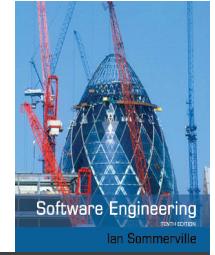
- ✧ Organizations often have an approved and recommended set of application software that has been checked by the IT department.
 - It is usually possible to buy or acquire open source software from this set directly without the need for detailed justification.
 - There are no detailed requirements and the users adapt to the features of the chosen application.
- ✧ Off-the-shelf components do not usually match requirements exactly.
 - Choosing a system means that you have to find the closest match between the system requirements and the facilities offered by off-the-shelf systems.

Procurement issues (2)



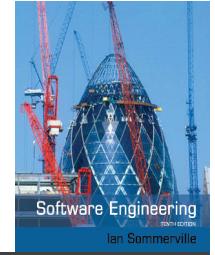
- ✧ When a system is to be built specially, the specification of requirements is part of the contract for the system being acquired.
 - It is therefore a legal as well as a technical document.
 - The requirements document is critical and procurement processes of this type usually take a considerable amount of time.
- ✧ For public sector systems especially, there are detailed rules and regulations that affect the procurement of systems.
 - These force the development of detailed requirements and make agile development difficult

Procurement issues (3)



- ✧ For application systems that require change or for custom systems there is usually a contract negotiation period where the customer and supplier negotiate the terms and conditions for the development of the system.
 - During this process, requirements changes may be agreed to reduce the overall costs and avoid some development problems.

Procurement decisions



- ✧ Decisions made at the procurement stage of the systems engineering process are critical for later stages in that process.
 - Poor procurement decisions often lead to problems such as late delivery of a system and the development of systems that are unsuited to their operational environment.
 - If the wrong system or the wrong supplier is chosen then the technical processes of system and software engineering become more complex.

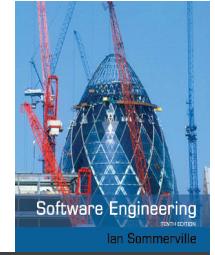


Software Engineering

Ian Sommerville

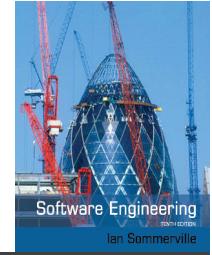
System development

System development



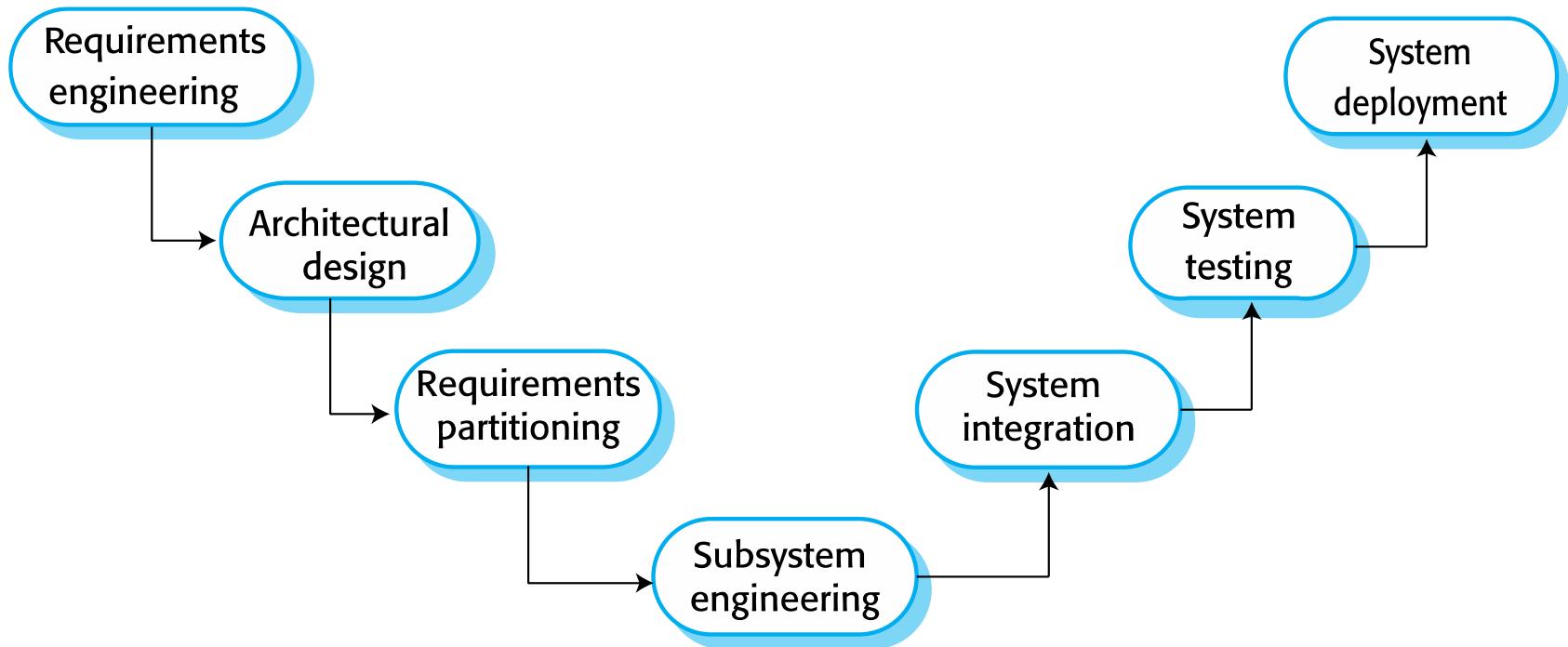
- ✧ Usually follows a plan-driven approach because of the need for parallel development of different parts of the system
 - Little scope for iteration between phases because hardware changes are very expensive. Software may have to compensate for hardware problems.
- ✧ Inevitably involves engineers from different disciplines who must work together
 - Much scope for misunderstanding here.
 - As explained, different disciplines use a different vocabulary and much negotiation is required. Engineers may have personal agendas to fulfil.

Systems development



Software Engineering

Ian Sommerville



The system development process



✧ Requirements engineering

- The process of refining, analysing and documenting the high-level and business requirements identified in the conceptual design

✧ Architectural design

- Establishing the overall architecture of the system, identifying components and their relationships

✧ Requirements partitioning

- Deciding which subsystems (identified in the system architecture) are responsible for implementing the system requirements

The system development process (2)



✧ Subsystem engineering

- Developing the software components of the system, configuring off-the-shelf hardware and software, defining the operational processes for the system and re-designing business processes

✧ System integration

- Putting together system elements to create a new system

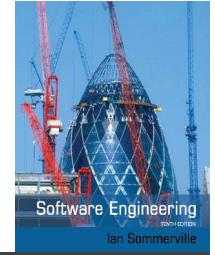
✧ System testing

- The whole system is tested to discover problems

✧ System deployment

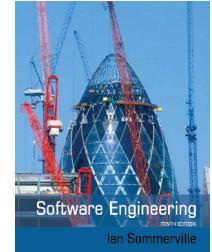
- the process of making the system available to its users, transferring data from existing systems and establishing communications with other systems in the environment

Requirements and design

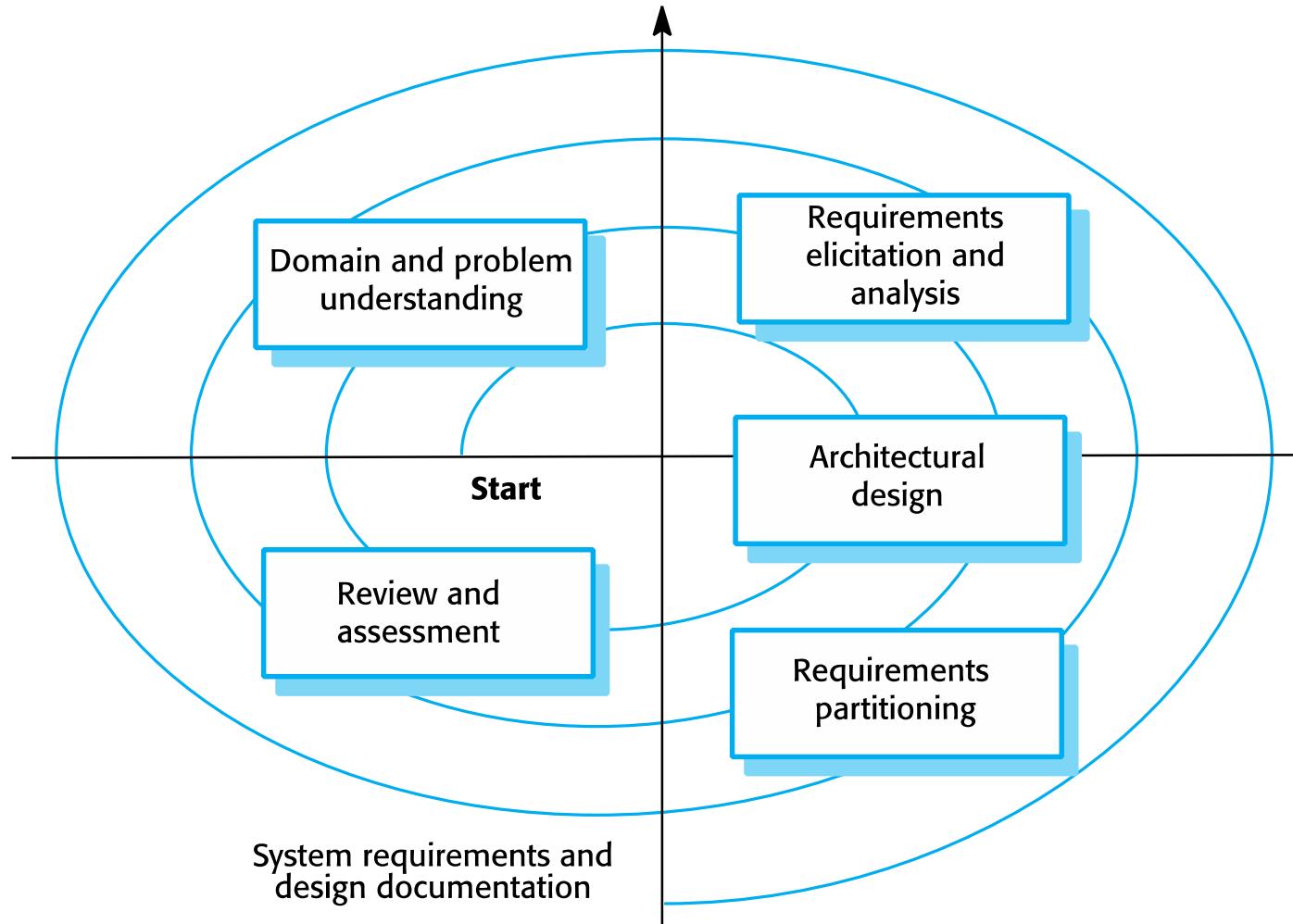


- ✧ Requirements engineering and system design are inextricably linked.
- ✧ Constraints posed by the system's environment and other systems limit design choices so the actual design to be used may be a requirement.
- ✧ Initial design may be necessary to structure the requirements.
- ✧ As you do design, you learn more about the requirements.

Requirements and design spiral



Software Engineering
Ian Sommerville



Subsystem engineering



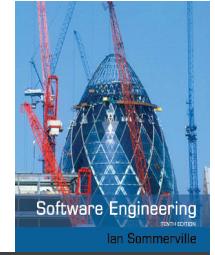
- ✧ Typically parallel projects developing the hardware, software and communications.
- ✧ May involve some application systems procurement.
- ✧ Lack of communication across implementation teams can cause problems.
- ✧ There may be a bureaucratic and slow mechanism for proposing system changes, which means that the development schedule may be extended because of the need for rework.

System integration



- ✧ The process of putting hardware, software and people together to make a system.
- ✧ Should ideally be tackled incrementally so that sub-systems are integrated one at a time.
- ✧ The system is tested as it is integrated.
- ✧ Interface problems between sub-systems are usually found at this stage.
- ✧ May be problems with uncoordinated deliveries of system components.

System delivery and deployment



- ✧ After completion, the system has to be installed in the customer's environment
 - Environmental assumptions may be incorrect;
 - May be human resistance to the introduction of a new system;
 - System may have to coexist with alternative systems for some time;
 - May be physical installation problems (e.g. cabling problems);
 - Data cleanup may be required;
 - Operator training has to be identified.



Software Engineering

Ian Sommerville

System operation and evolution

System operation



- ✧ Operational processes are the processes involved in using the system for its defined purpose.
- ✧ For new systems, these processes may have to be designed and tested and operators trained in the use of the system.
- ✧ Operational processes should be flexible to allow operators to cope with problems and periods of fluctuating workload.

Problems with operation automation



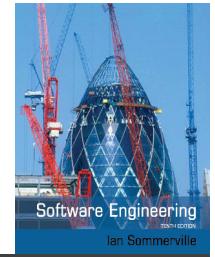
- ✧ It is likely to increase the technical complexity of the system because it has to be designed to cope with all anticipated failure modes. This increases the costs and time required to build the system.
- ✧ Automated systems are inflexible. People are adaptable and can cope with problems and unexpected situations. This means that you do not have to anticipate everything that could possibly go wrong when you are specifying and designing the system

System evolution



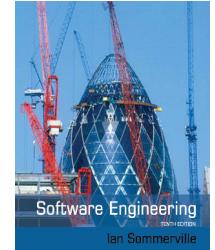
- ✧ Large systems have a long lifetime. They must evolve to meet changing requirements.
- ✧ Evolution is inherently costly
 - Changes must be analysed from a technical and business perspective;
 - Sub-systems interact so unanticipated problems can arise;
 - There is rarely a rationale for original design decisions;
 - System structure is corrupted as changes are made to it.
- ✧ Existing systems which must be maintained are sometimes called legacy systems.

Factors that affect system lifetimes



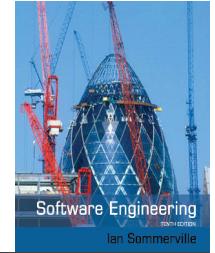
Factor	Rationale
Investment cost	The costs of a systems engineering project may be tens or even hundreds of millions of dollars. These costs can only be justified if the system can deliver value to an organization for many years.
Loss of expertise	As businesses change and restructure to focus on their core activities, they often lose engineering expertise. This may mean that they lack the ability to specify the requirements for a new system.
Replacement cost	The cost of replacing a large system is very high. Replacing an existing system can only be justified if this leads to significant cost savings over the existing system.

Factors that affect system lifetimes



Factor	Rationale
Return on investment	If a fixed budget is available for systems engineering, spending this on new systems in some other area of the business may lead to a higher return on investment than replacing an existing system.
Risks of change	Systems are an inherent part of business operations and the risks of replacing existing systems with new systems cannot be justified. The danger with a new system is that things can go wrong in the hardware, software and operational processes. The potential costs of these problems for the business may be so high that they cannot take the risk of system replacement.
System dependencies	Other systems may depend on a system and making changes to these other systems to accommodate a replacement system may be impractical.

Cost factors in system evolution



- ✧ Proposed changes have to be analyzed very carefully from a business and a technical perspective.
- ✧ Subsystems are never completely independent so changes to a subsystem may have side-effects that adversely affect other subsystems.
- ✧ Reasons for original design decisions are often unrecorded. Those responsible for the system evolution have to work out why these decisions were made.
- ✧ As systems age, their structure becomes corrupted by change so the costs of making further changes increases.

Key points



- ✧ Systems engineering is concerned with all aspects of specifying, buying, designing and testing complex sociotechnical systems.
- ✧ Sociotechnical systems include computer hardware, software and people, and are situated within an organization. They are designed to support organizational or business goals and objectives.
- ✧ The emergent properties of a system are characteristics of the system as a whole rather than of its component parts. They include properties such as performance, reliability, usability, safety and security.

Key points



- ✧ The fundamental systems engineering processes are conceptual systems design, system procurement, system development and system operation.
- ✧ Conceptual systems design is a key activity where high level system requirements and a vision of the operational system is developed.
- ✧ System procurement covers all of the activities involved in deciding what system to buy and who should supply that system. Different procurement processes are used for off-the-shelf application systems, configurable COTS systems and custom systems.

Key points



- ✧ System development processes include requirements specification, design, construction, integration and testing.
- ✧ When a system is put into use, the operational processes and the system itself inevitably change to reflect changes to the business requirements and the system's environment.



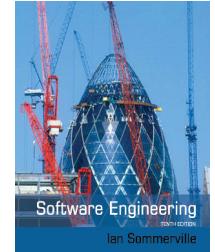
Chapter 20 – Systems of Systems

Topics covered



- ✧ System complexity
- ✧ System of systems classification
- ✧ Reductionism and complex systems
- ✧ Systems of systems engineering
- ✧ Systems of systems architecture

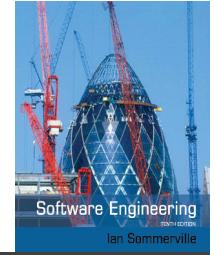
Systems of systems



Software Engineering
Ian Sommerville

- ✧ More and more systems are being constructed by integrated existing, independent systems
- ✧ *A system of systems is a system that contains two or more independently managed elements.*
- ✧ There is no single manager for all of the parts of the system of systems and that different parts of a system are subject to different management and control policies and rules.

Examples of systems of systems



- ✧ A cloud management system that handles local private cloud management and management of servers on public clouds such as Amazon and Microsoft.
- ✧ An online banking system that handles loan requests and which connects to a credit reference system provided by credit reference agency to check the credit of applicants.
- ✧ An emergency information system that integrates information from police, ambulance, fire and coastguard services about the assets available to deal with civil emergencies such as flooding and large-scale accidents.

Essential characteristics of SoS



- ✧ Operational independence of system elements
- ✧ Managerial independence of system elements
- ✧ Evolutionary development
- ✧ Emergence of system characteristics
- ✧ Geographic distribution of system elements
- ✧ Data intensive (data >> code)
- ✧ Heterogeneity

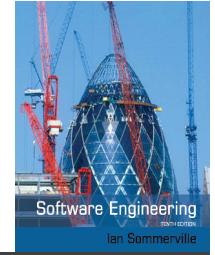


Software Engineering

Ian Sommerville

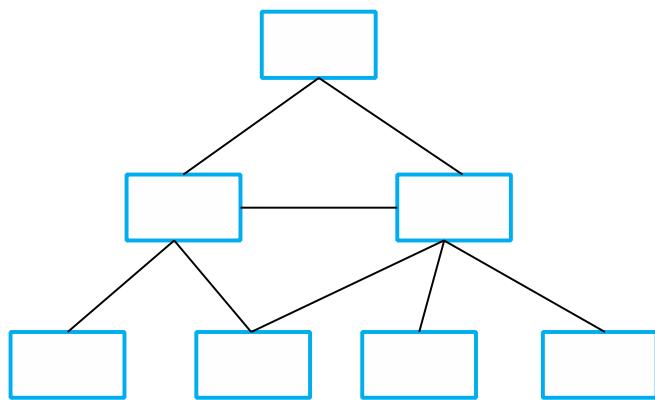
System complexity

Complexity

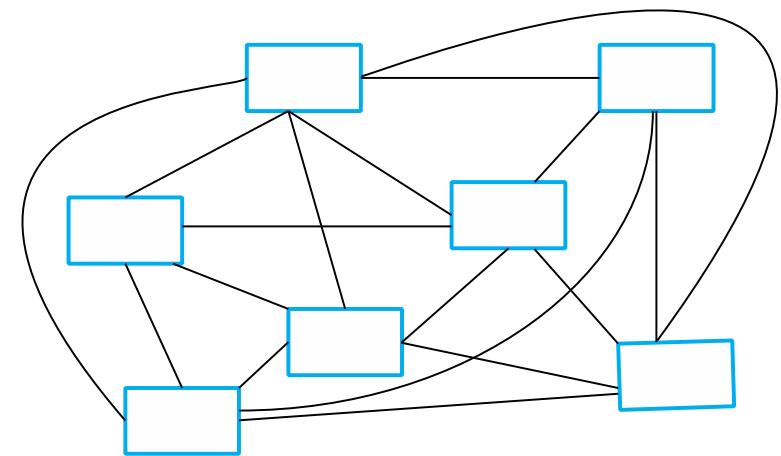


- ✧ All systems are composed of parts (elements) with relationships between these elements of the system.
 - For example, the parts of a program may be objects and the parts of each object may be constants, variables and methods.
 - Examples of relationships include ‘calls’ (method A calls method B), ‘inherits-from’ (object X inherits the methods and attributes of object Y) and ‘part of’ (method A is part of object X).
- ✧ The complexity of any system depends on the number and the types of relationships between system elements.
- ✧ The type of relationship (static or dynamic) also influences the overall complexity of a system.

Simple and complex systems



System (a)



System (b)

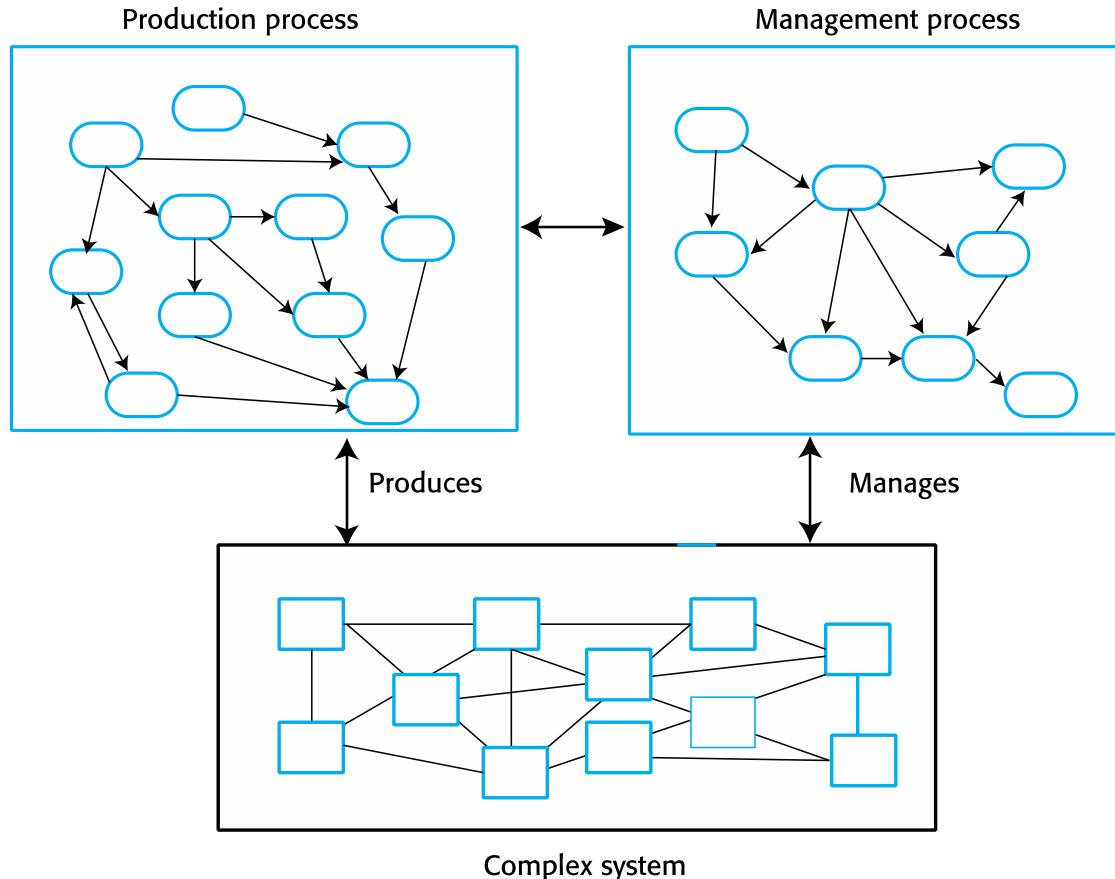
Process complexity



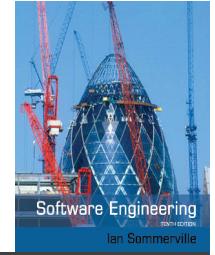
Software Engineering
Ian Sommerville

- ✧ As systems grow in size, they need more complex production and management processes.
- ✧ Complex processes are themselves complex systems.
 - They are difficult to understand and may have undesirable emergent properties. They are more time consuming than simpler processes and they require more documentation and coordination between the people and the organizations involved in the system development.
- ✧ The complexity of the production process is one of the main reasons why projects go wrong, with software delivered late and over-budget.

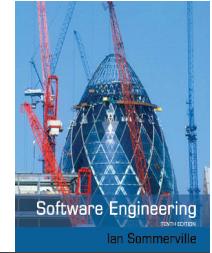
System production and management processes



Complexity and software engineering



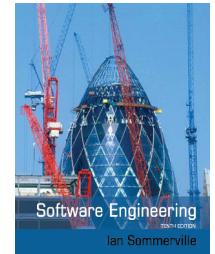
- ✧ Complexity is important for software engineering because it is the main influence on the understandability and the changeability of a system.
- ✧ The more complex a system, the more difficult it is to understand and analyze.
- ✧ As complexity increases, there are more and more relationships between elements of the system and an increased likelihood that changing one part of a system will have undesirable effects elsewhere.



Types of complexity

- ✧ *Technical complexity* is derived from the relationships between the different components of the system itself.
- ✧ *Managerial complexity* is derived from the complexity of the relationships between the system and its managers and the relationships between the managers of different parts of the system.
- ✧ *Governance complexity* of a system depends on the relationships between the laws, regulations and policies that affect the system and the relationships between the decision-making processes in the organizations responsible for the system.

System characteristics and complexity



SoS characteristic	Technical complexity	Managerial complexity	Governance complexity
Operational independence		X	X
Managerial independence	X	X	
Evolutionary development	X		
Emergence	X		
Geographical distribution	X	X	X
Data-intensive	X		X
Heterogeneity	X		

Complexity and project failure



- ✧ Large-scale systems of systems are now unimaginably complex entities that cannot be understood or analyzed as a whole.
- ✧ The large number of interactions between the parts and the dynamic nature of these interactions means that conventional engineering approaches do not work well for complex systems.
- ✧ It is complexity that is the root cause of problems in projects to develop large software-intensive systems, not poor management or technical failings.

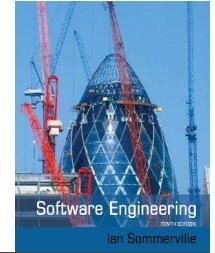


Software Engineering

Ian Sommerville

Systems of systems classification

Maier's classification of systems of systems



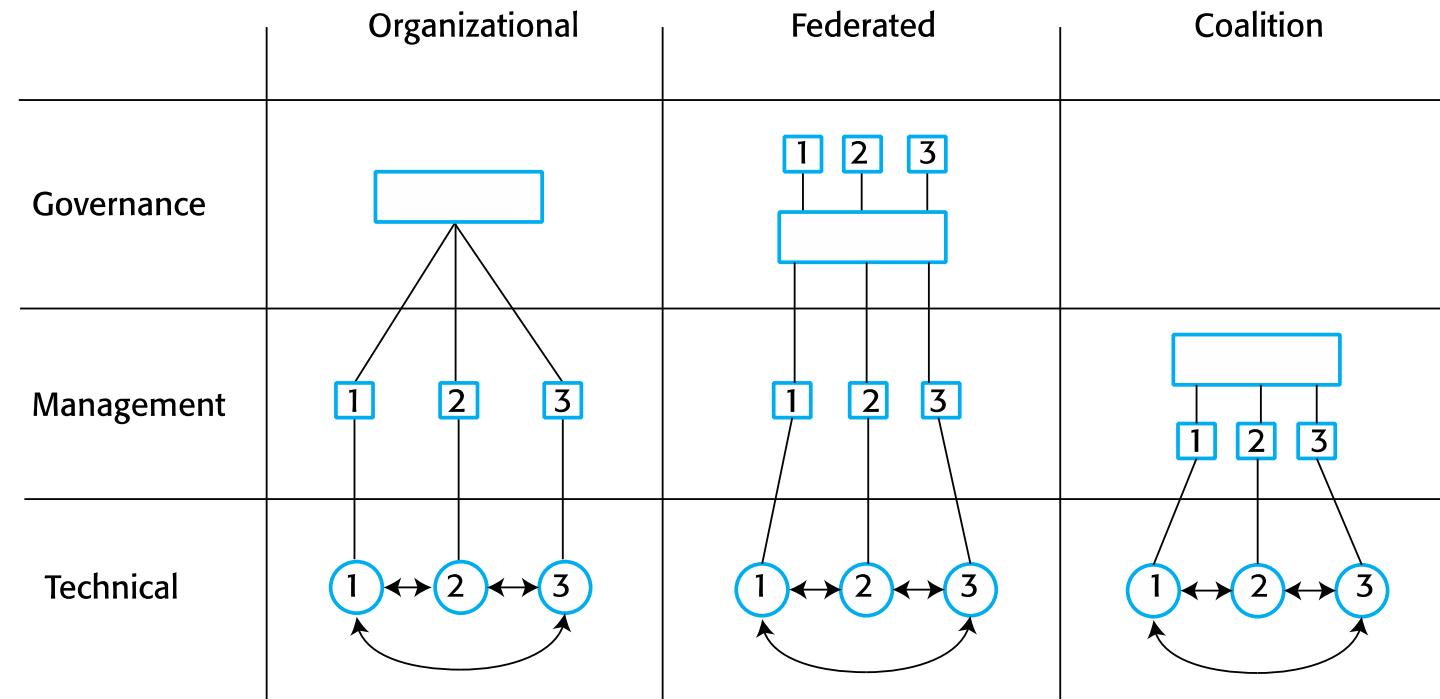
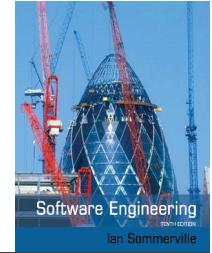
- ✧ **Directed SoS** are owned by a single organization and are developed by integrating systems that are also owned by that organization. The system elements may be independently managed by parts of the organization.
- ✧ **Collaborative SoS** are systems where there is no central authority to set management priorities and resolve disputes. Typically, elements of the system are owned and governed by different organizations.
- ✧ **Virtual systems** have no central governance and the participants may not agree on the overall purpose of the system. Participant systems may enter or leave the SoS.

More intuitive classification terms

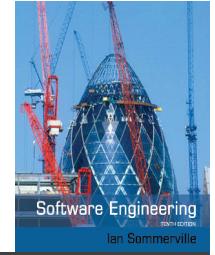


- ✧ *Organizational systems of systems* are SoS where the governance and management of the system lies within the same organization or company.
- ✧ *Federated systems* are SoS where the governance of the SoS depends on a voluntary participative body in which all of the system owners are represented.
- ✧ *System of system coalitions* are SoS where there are no formal governance mechanisms but where the organizations involved informally collaborate and manage their own systems to maintain the system as a whole.

System of systems classification



iLearn as a SoS



- ✧ iLearn is a relatively simple technical system but it has a high level of governance complexity.
- ✧ The development of a digital learning system is a national initiative but to create a digital learning environment, it has to be integrated with network management and school administration systems.
- ✧ There is no common governance process across authorities so, according to the classification scheme, this is a coalition of systems.



Software Engineering

Ian Sommerville

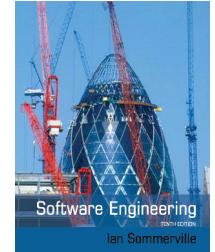
Reductionism and complex systems

Complexity management in engineering



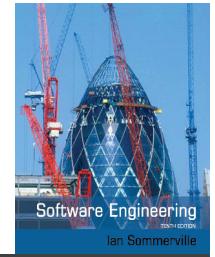
- ✧ The approach that has been the basis of complexity management in software engineering is called *reductionism*.
- ✧ Reductionism is based on the assumption that any system is made up of parts or subsystems.
 - It assumes that the behaviour and properties of the system as a whole can be understood and predicted by understanding the individual parts and the relationships between these parts.
- ✧ To design a system, the parts making up that system are identified, constructed separately and then assembled into the complete system.

Software engineering methods



- ✧ A reductionist approach has been the basis of software engineering for almost 50 years.
- ✧ Top-down design, where you start with a very high-level model of a system and break this down to its components is a reductionist approach.
 - This is the basis of all software design methods, such as object-oriented design. Programming languages include abstractions, such as procedures and objects that directly reflect reductionist system decomposition.
 - Agile methods are also reductionist. The difference between agile methods and top-down design is that system decomposition is incremental when an agile approach is used.

Reductionist methods



- ✧ Reductionist methods are successful when there are relatively few relationships between the parts of a system and it is possible to model these relationships.
- ✧ Software engineering methods attempt to limit complexity by controlling the relationships between parts of the system.
- ✧ Reductionism does not work well when there are many relationships in a system and when these relationships are difficult to understand and analyze.
 - The fundamental assumptions that are inherent to reductionism are inapplicable for large and complex systems

Reductionist assumptions



✧ *System ownership and control*

- Reductionism assumes that there is a controlling authority for a system that can resolve disputes and make high-level technical decisions that will apply across the system.

✧ *Rational decision making*

- Reductionism assumes that interactions between components can be objectively assessed by, for example, mathematical modelling.

✧ *Defined system boundaries*

- Reductionism assumes that the boundaries of a system can be agreed and defined.

System of systems reality



Software Engineering
Ian Sommerville

Control

Owners of a system control its development

Decision making

Decisions are made rationally, driven by technical criteria

Problem definition

There is a definable problem and clear system boundaries

Reductionist assumptions

There is no single system owner or controller

Decision-making driven by political motives

Wicked problem with constantly renegotiated system boundaries

Systems of systems reality

Reductionism and software SoS

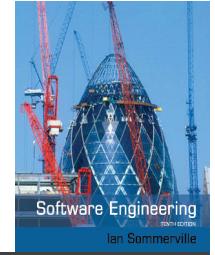


- ✧ Relationships in software systems are not governed by physical laws.
 - Political factors are usually the driver of decision making for large and complex software systems.
- ✧ Software has no physical limitations hence there are no limits on where the boundaries of a system are drawn.
 - The boundaries and the scope of a system are likely to change during its development.
- ✧ Linking software systems from different owners is relatively easy hence we are more likely to try and create a SoS where there is no single governing body.



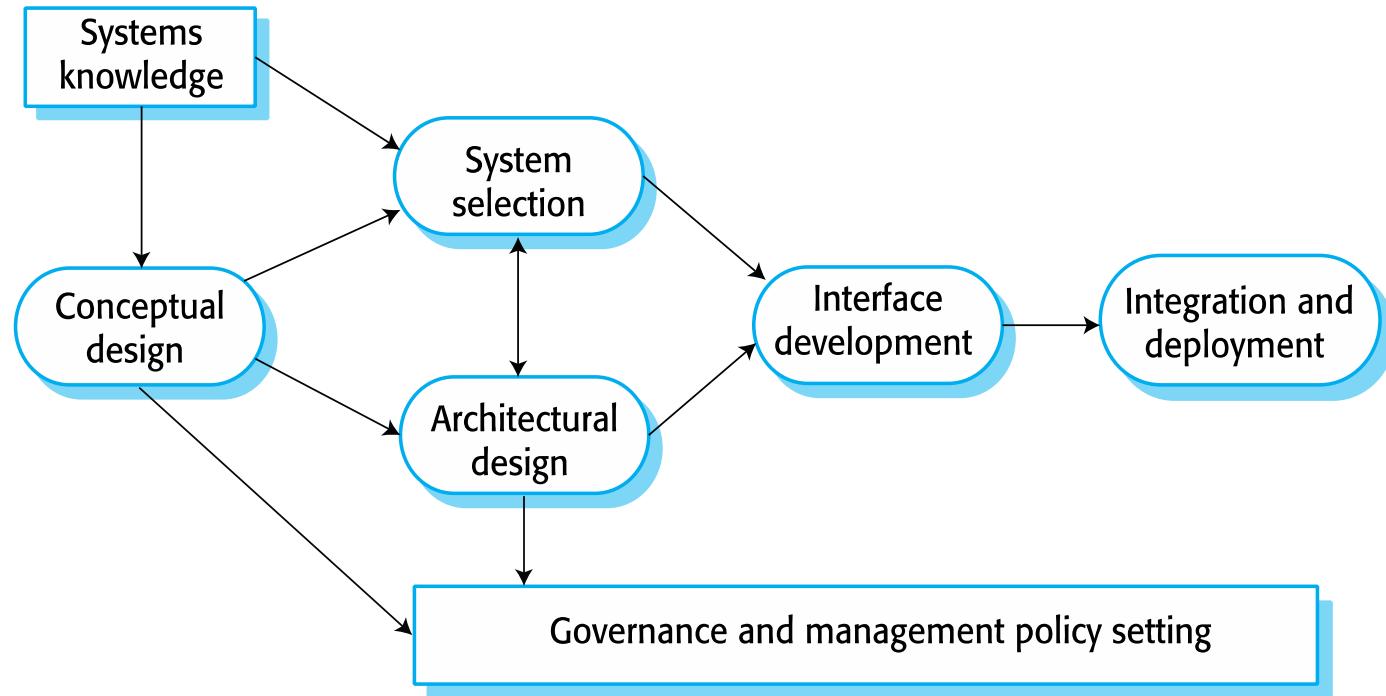
Systems of systems engineering

SoS engineering problems



- ✧ Lack of control over system functionality and performance.
- ✧ Differing and incompatible assumptions made by the developers of the different systems.
- ✧ Different evolution strategies and timetables for the different systems.
- ✧ Lack of support from system owners when problems arise.

Systems of systems engineering



SoS development processes



- ✧ *Conceptual design* is the activity of creating a high-level vision for a system, defining essential requirements and identifying constraints on the overall system.
- ✧ *System selection*, where a set of systems for inclusion in the SoS is chosen.
 - Political imperatives and issues of system governance and management are often the key factors that influence what systems are included in a SoS.
- ✧ *Architectural design* where an overall architecture for the SoS is developed.

SoS development processes



- ✧ *Interface development* – the development of system interfaces so that the constituent systems can interoperate.
- ✧ *Integration and deployment* – making the different systems involved in the SoS work together and interoperate through the developed interfaces.
- ✧ System deployment means putting the system into place in the organizations concerned and making it operational.

Interface development



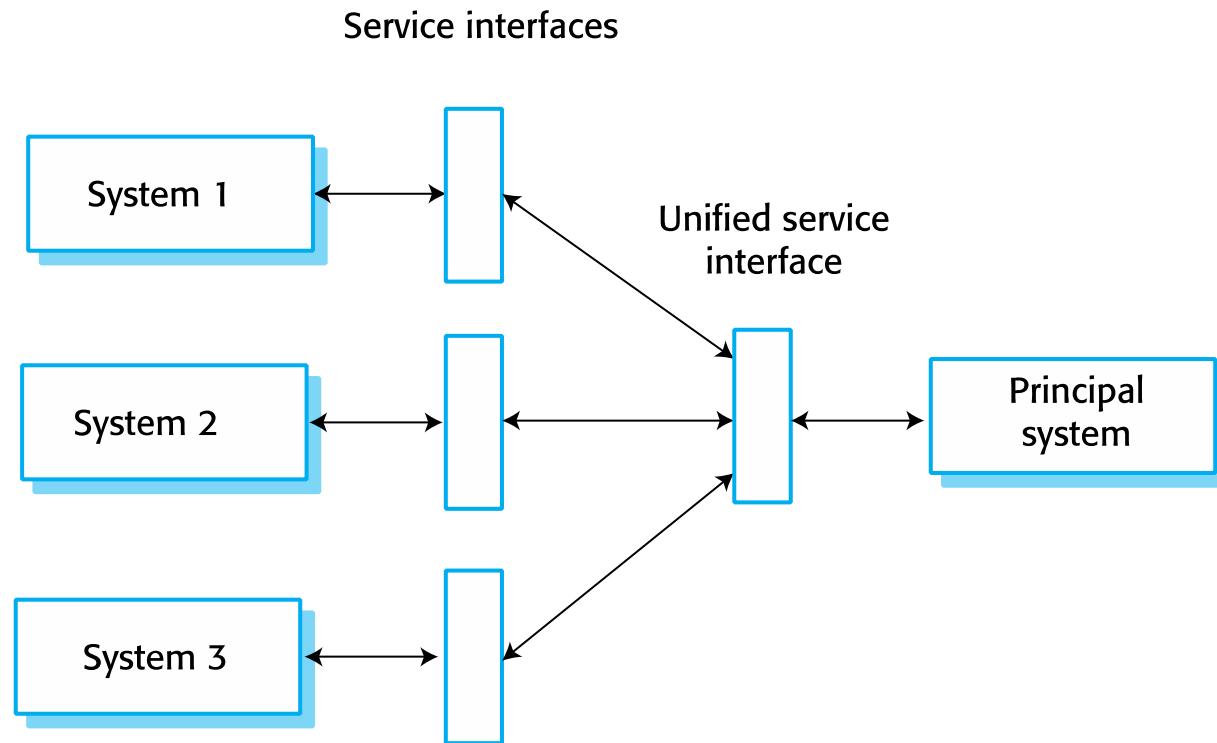
- ✧ In general, the aim in SoS development is for systems to be able to communicate directly with each other without user intervention.
- ✧ Service interfaces
 - If systems in a SoS have service interfaces, they can communicate directly via these interfaces
- ✧ The constituent systems in a SoS often have their own specialized API or only allow their functionality to be accessed through their user interfaces.
 - You therefore have to develop software that reconciles the differences between these interfaces.

Service interface development



- ✧ To develop service-based interfaces, you have to examine the functionality of existing systems and define a set of services to reflect that functionality.
- ✧ The services are implemented either by calls to the underlying system API or by mimicking user interaction with the system.
- ✧ A principal system acts as a service broker, directing service calls between the different systems in the SoS.
- ✧ Each system therefore does not need to know which other system is providing a called service.

Service interfaces



Unified user interfaces



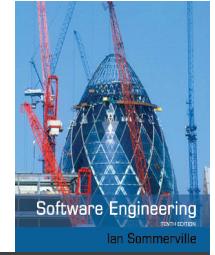
- ✧ User interfaces for each system in a SoS are likely to be different.
- ✧ A principal system must have some overall user interfaces that handles user authentication and provides access to the features of the underlying system.
- ✧ It is usually expensive and time-consuming to implement a unified user interface to replace the individual interfaces of the underlying systems.

Cost-effectiveness of UI development



- ✧ The interaction assumptions of the systems in the SoS
 - If systems have different interaction models, unifying these in a single UI is very difficult
- ✧ The mode of use of the SoS
 - A unified UI slows down interaction if most of the interaction is with a principal system in the SoS
- ✧ The 'openness' of the SoS
 - If the SoS is open, so that new systems may be added to it when it is in use, then unified UI development is impractical.

Integration and deployment



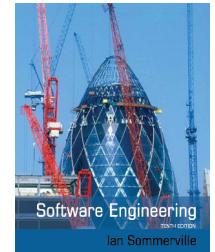
- ✧ For SoS, it makes sense to consider integration and deployment to be part of the same process.
- ✧ Separate integration may be difficult as some of the systems in the SoS may already be in use
- ✧ The integration process should begin with systems that are already deployed, with new systems added to the SoS to provide coherent additions to the functionality of the overall system.

Staged deployment of the iLearn system

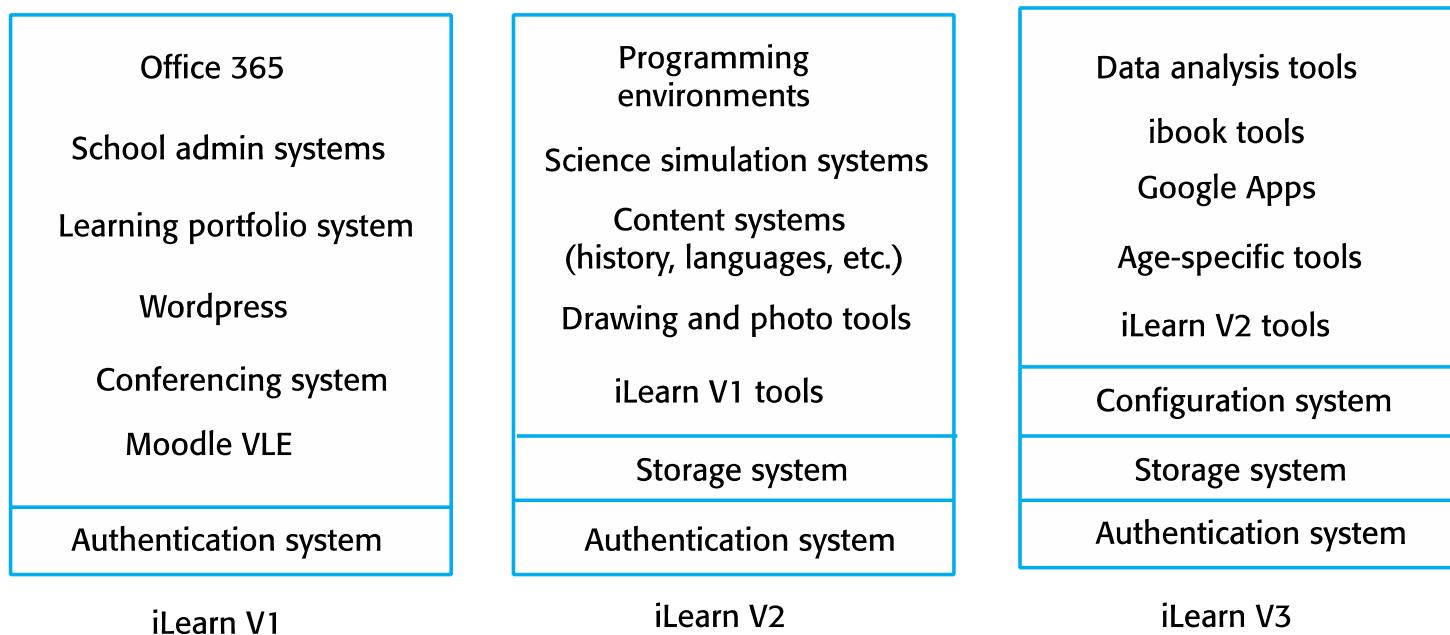


- ✧ The initial deployment provides authentication, basic learning functionality and integration with school administration systems.
- ✧ Stage 2 adds an integrated storage system and a set of more specialized tools to support subject-specific learning.
- ✧ Stage 3 adds features for user configuration and the ability for users to add new systems to the iLearn environment.

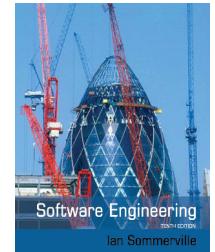
iLearn releases



Release timeline

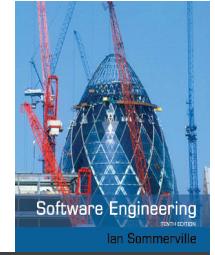


SoS testing



- ✧ There are three reasons why testing systems of systems is difficult and expensive:
 - There may not be a detailed requirements specification that can be used as a basis for system testing. It may not be cost effective to develop a SoS requirements document – the details of the system functionality are defined by the systems included.
 - The constituent systems may change in the course of the testing process so tests may not be repeatable.
 - If problems are discovered, it may not be possible to fix the problems by requiring one or more of the constituent systems to be changed. Intermediate software may have to be introduced to solve the problem.

SoS testing and agile testing



- ✧ Agile methods do not rely on having a complete system specification for system acceptance testing.
- ✧ Stakeholders are engaged with the testing process and to decide when the overall system is acceptable.
- ✧ For SoS, a range of stakeholders should be involved in the testing process if possible and they can comment on whether or not the system is ready for deployment.
- ✧ Agile methods make extensive use of automated testing. This makes it much easier to rerun tests to discover if unexpected system changes have caused problems for the SoS as a whole.

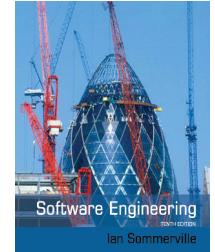


Software Engineering

Ian Sommerville

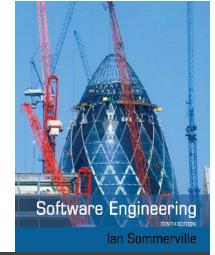
Systems of systems architecture

General principles for architecting SoS



- ✧ Design systems so that they can deliver value if they are incomplete.
- ✧ Be realistic about what can be controlled.
- ✧ Focus on the system interfaces.
- ✧ Provide collaboration incentives.
- ✧ Design a SoS as node and web architecture.
- ✧ Specify behaviour as services exchanged between nodes.
- ✧ Understand and manage system vulnerabilities.

Architectural frameworks



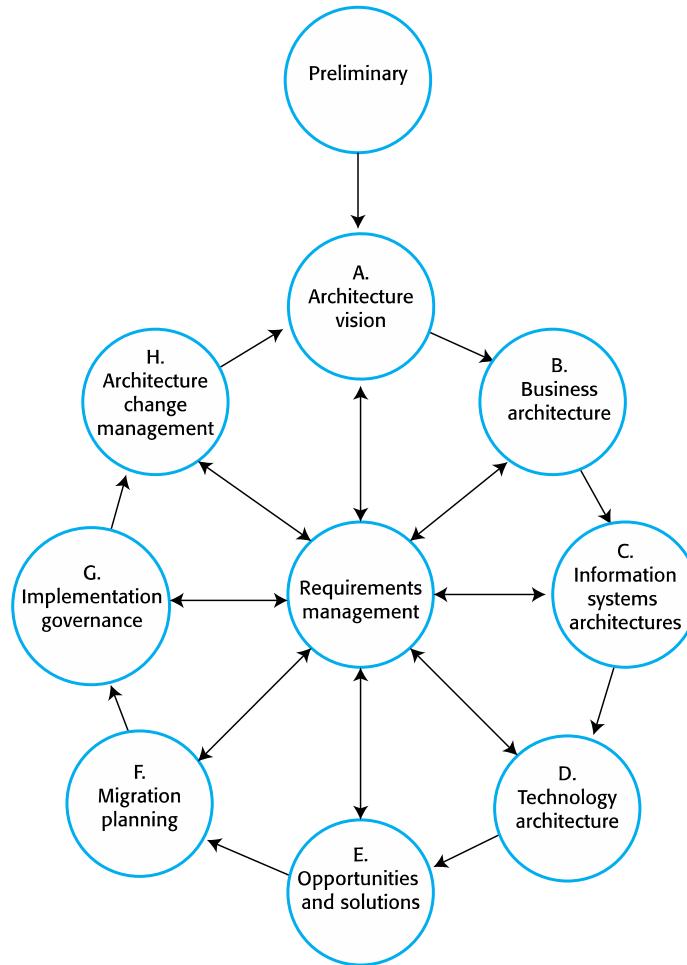
- ✧ Architectural frameworks such as MODAF and TOGAF have been suggested as a means to support the architectural design of systems of systems.
- ✧ An architecture framework recognises that a single model of an architecture does not present all of the information needed for architectural and business analysis.
- ✧ Frameworks propose a number of architectural views that should be created and maintained to describe and document enterprise systems.

TOGAF

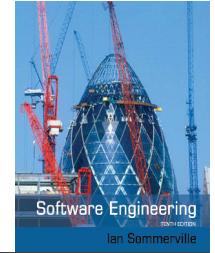


- ✧ The TOGAF framework has been developed by the Open Group as an open standard and is intended to support the design of a business architecture, a data architecture, an application architecture and a technology architecture for an enterprise.
- ✧ At its heart is the Architecture Development Method (ADM), which consists of a number of discrete phases.

TOGAF – Architecture Development Method

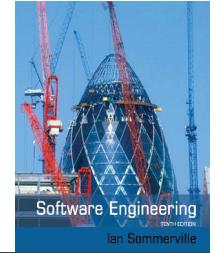


Architectural model management



- ✧ Initial model development takes a long time and involves extensive negotiations between system stakeholders. This slows the development of the overall system.
- ✧ It is time-consuming and expensive to maintain model consistency as changes are made to the organization and the constituent systems in a SoS.

Architectural patterns for SoS



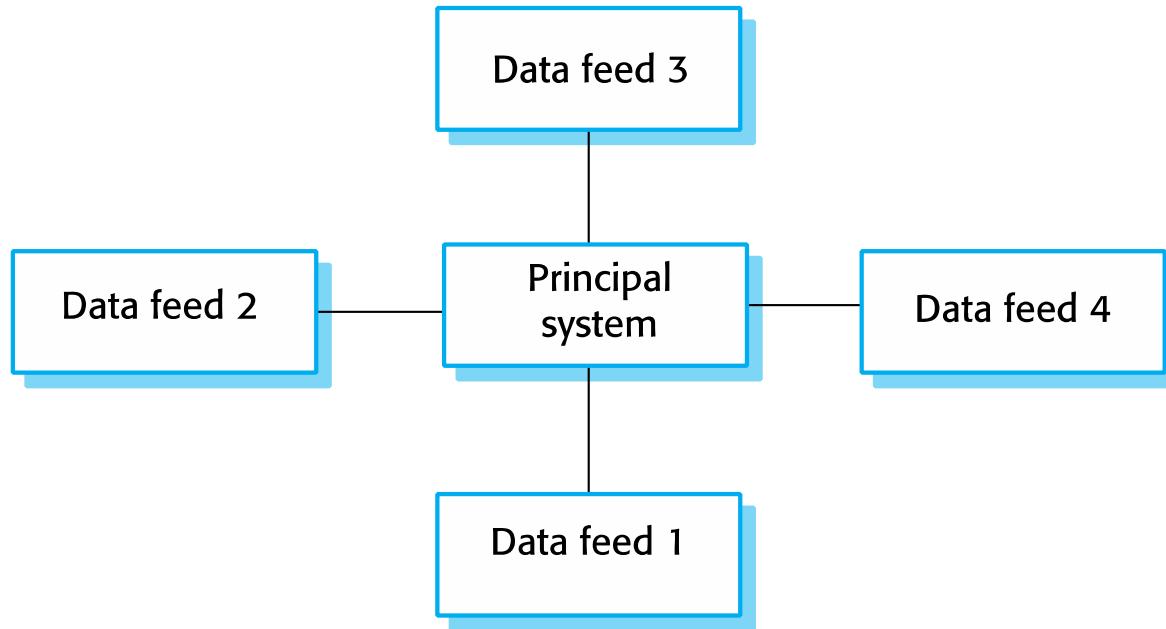
- ✧ An architectural pattern is a stylized architecture that can be recognized across a range of different systems.
- ✧ Architectural patterns are a useful way of stimulating discussions about the most appropriate architecture for a system and for documenting and explaining the architectures used.

Systems as data feeds

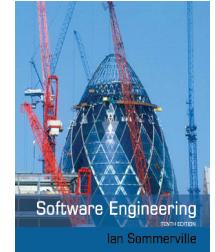


- ✧ There is a principal system that requires data of different types.
- ✧ This data is available from other systems and the principal system queries these systems to get the data required.
- ✧ Generally, the systems that provide data do not interact with each other.
- ✧ This pattern is often observed in organizational or federated systems where some governance mechanisms are in place.

Systems as data feeds

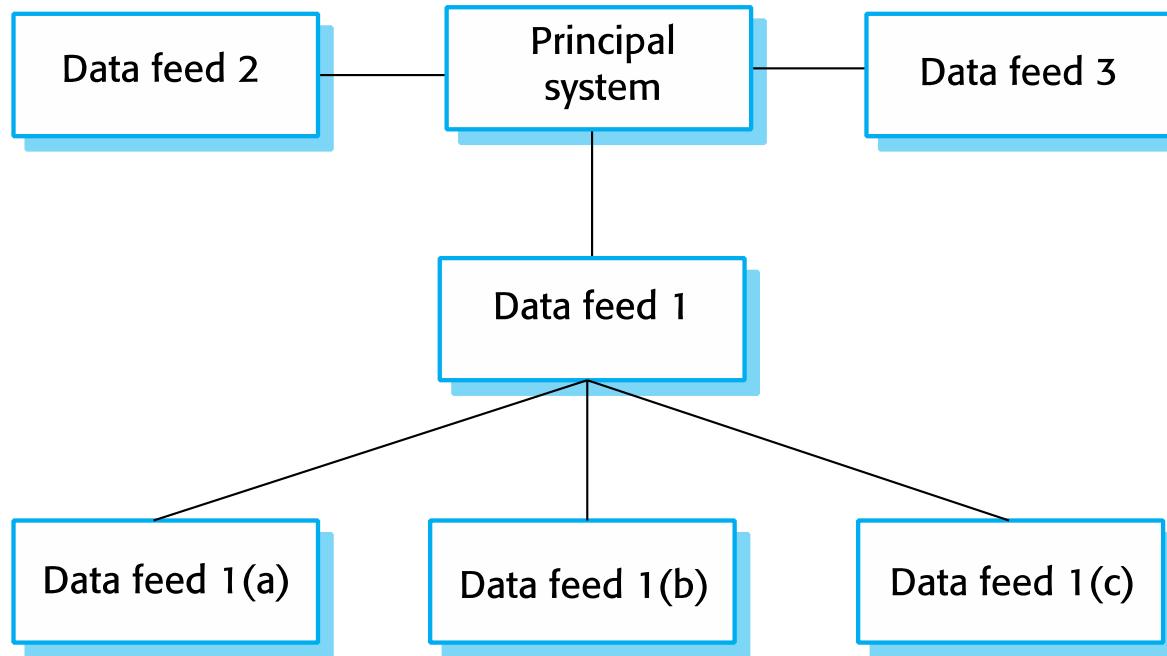


Systems as data feeds

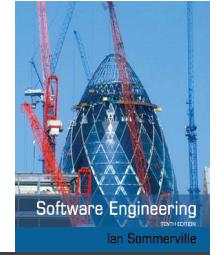


- ✧ The ‘systems as data feeds’ architecture is an appropriate architecture to use when it is possible to identify entities in a unique way and create relatively simple queries about these entities.
- ✧ A variant of the ‘systems as data feeds’ architecture arises when there are a number of systems involved which provide similar data but which are not identical.
- ✧ The architecture has to include an intermediate layer to translate the general query from the principal system into the specific query required by the individual information system.

Systems as data feeds with unifying interface

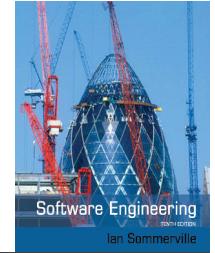


Systems in a container

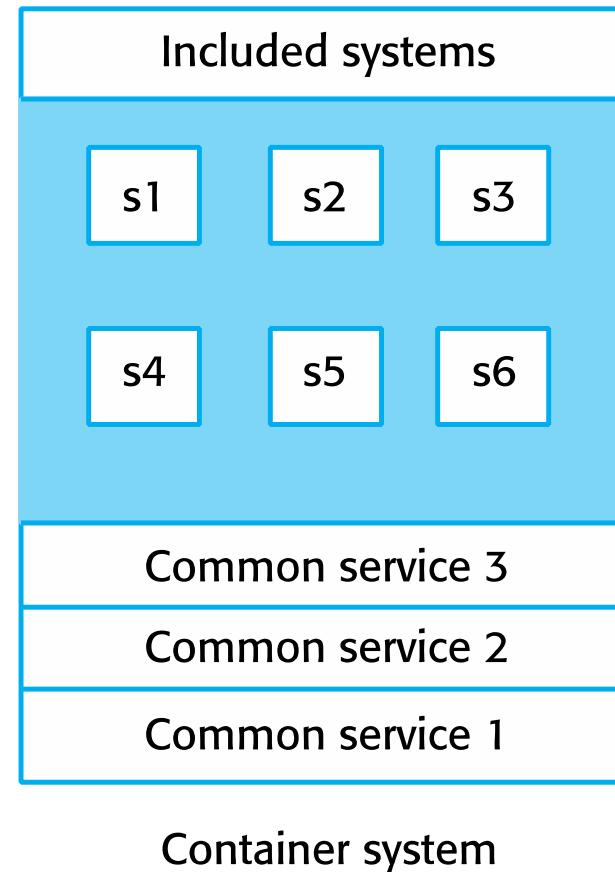


- ✧ Systems in a container are systems of systems where one of the systems acts as a virtual container and provides a set of common services such as an authentication and a storage service.
- ✧ Conceptually, other systems are then placed into this container to make their functionality accessible to system users.
- ✧ You don't place systems into a real container to implement these systems of systems. Rather, for each approved system, there is a separate interface that allows it to be integrated with the common services.

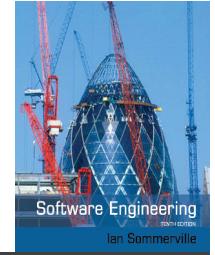
Container systems



Software Engineering
Ian Sommerville



ILearn container: common services

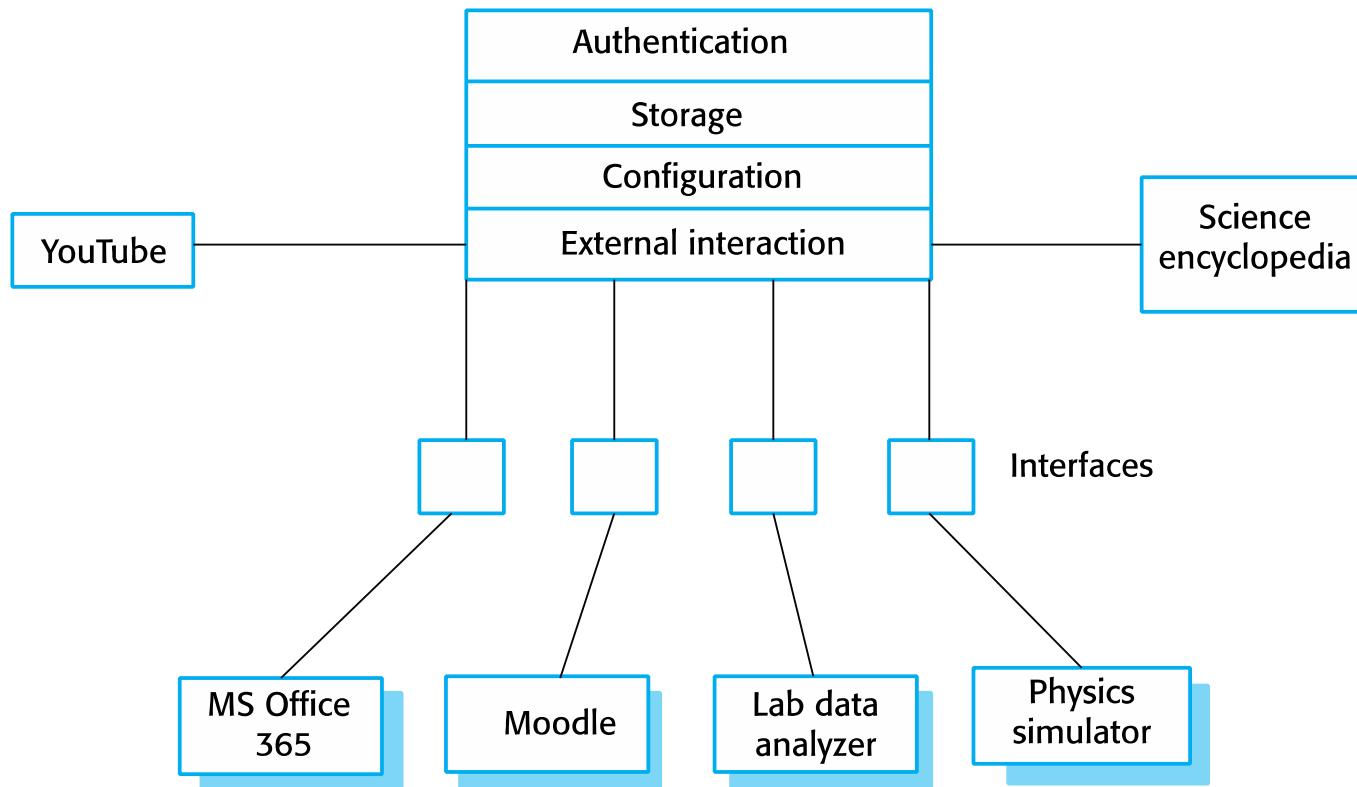


- ✧ An authentication service that provides a single sign-in to all approved systems. Users do not have to maintain separate credentials for these.
- ✧ A storage service for user data. This can be seamlessly transferred to and from approved systems.
- ✧ A configuration service that is used to include or remove systems from the container.

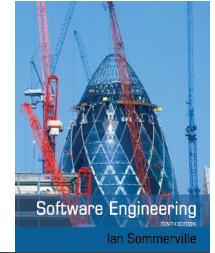
iLearn as a container



The Digital Learning Environment



Container architecture problems



- ✧ A separate interface must be developed for each approved system so that common services can be used with these systems.
- ✧ This means that only a relatively small number of approved systems can be supported.
- ✧ The owners of the container system have no influence on the functionality and behaviour of the included systems. Systems may stop working or may be withdrawn at any time.

Trading systems

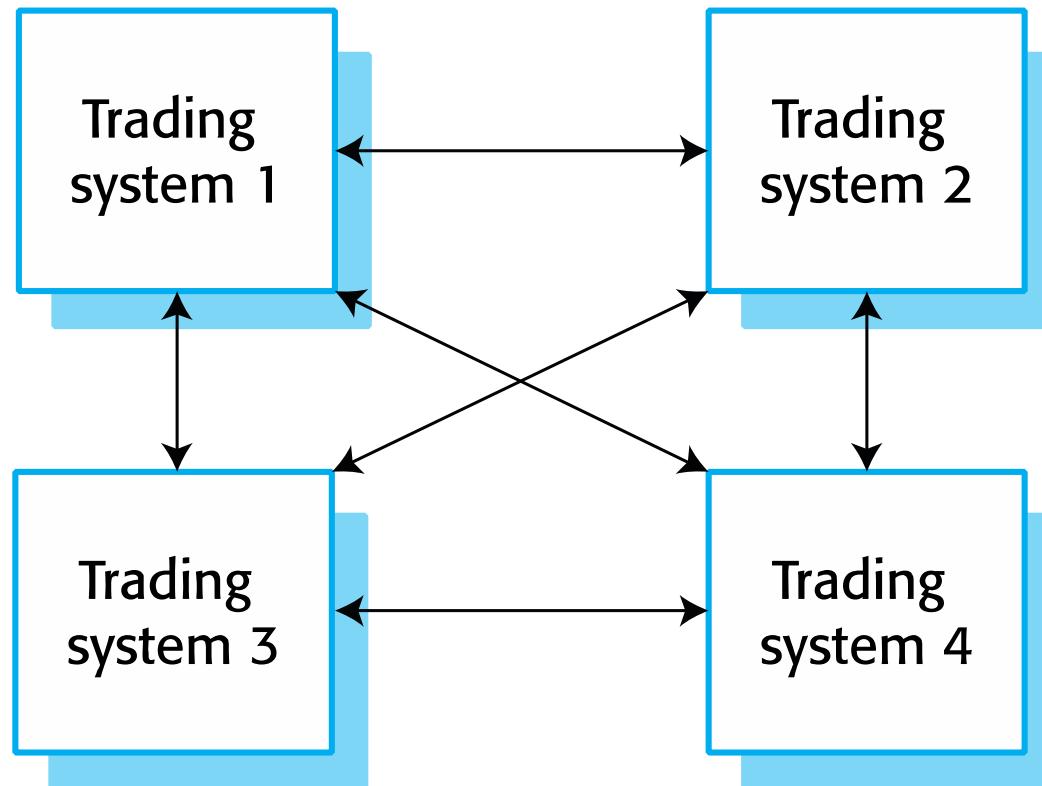


- ✧ Trading systems are systems of systems where there is no single principal system but processing may take place in any of the constituent systems.
- ✧ The systems involved trade information amongst themselves. There may be one-to-one or one-to-many interactions between these systems.
- ✧ Each system publishes its own interface but there may not be any interface standards that are followed by all systems.

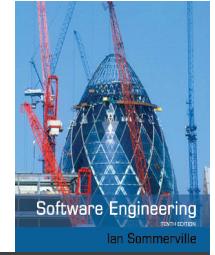
Trading systems



Software Engineering
Ian Sommerville

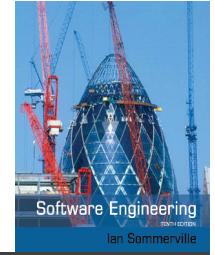


Trading SoS



- ✧ Trading systems may be developed for any type of marketplace with the information exchanged being information about the goods being traded and their prices.
- ✧ While trading systems are systems in their own right and could conceivably be used for individual trading, they are most useful in an automated trading context where the systems negotiate directly with each other.
- ✧ The major problem with this type of system is that there is no governance mechanism so any of the systems involved may change at any time.

Key points



- ✧ Systems of systems are systems where two or more of the constituent systems are independently managed and governed.
- ✧ There are three types of complexity that are important for systems of systems – technical complexity, managerial complexity and governance complexity.
- ✧ System governance can be used as the basis for a classification scheme for SoS. This leads to three classes of SoS namely organizational systems, federated systems and system coalitions.

Key points

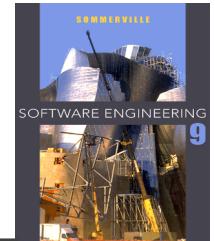


- ✧ Reductionism as an engineering method breaks down because of the inherent complexity of systems of systems.
- ✧ Reductionism assumes clear system boundaries, rational decision making and well-defined problems. None of these are true for systems of systems.
- ✧ The key stages of the SoS development process are conceptual design, system selection, architectural design, interface development and integration and deployment. Governance and management policies must be designed in parallel with these activities.

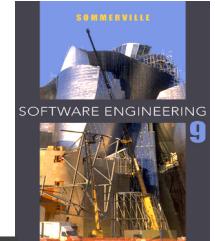
Key points



- ✧ Architectural patterns for systems of systems are a means of describing and discussing typical architectures for SoS.
- ✧ Important patterns are systems as data feeds, systems in a container and trading systems.

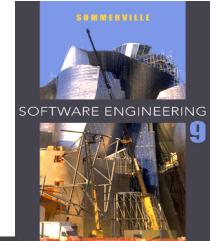


Chapter 21– Real-time Software Engineering



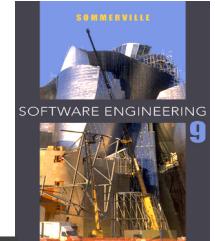
Topics covered

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



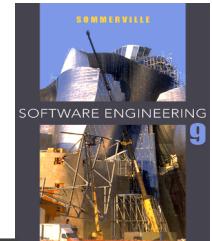
Embedded software

- ✧ 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.



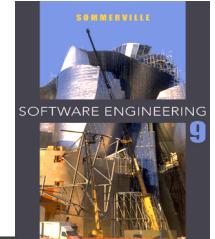
Responsiveness

- ✧ 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.



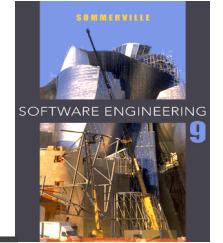
Definition

- ✧ 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.

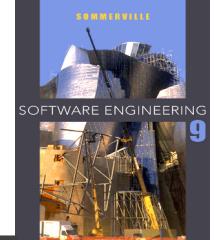


Characteristics of embedded systems

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

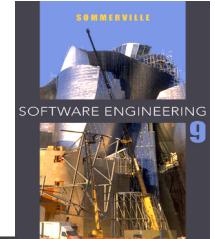


Embedded system design



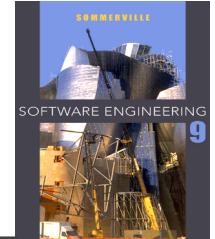
Embedded system design

- ✧ 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.



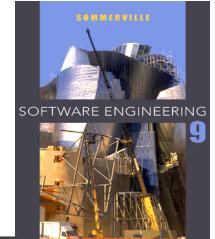
Reactive systems

- ✧ 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
 - 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.



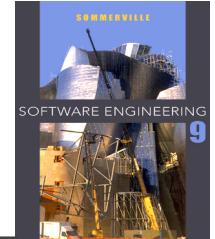
Stimuli and responses for a burglar alarm system

Stimulus	Response
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.

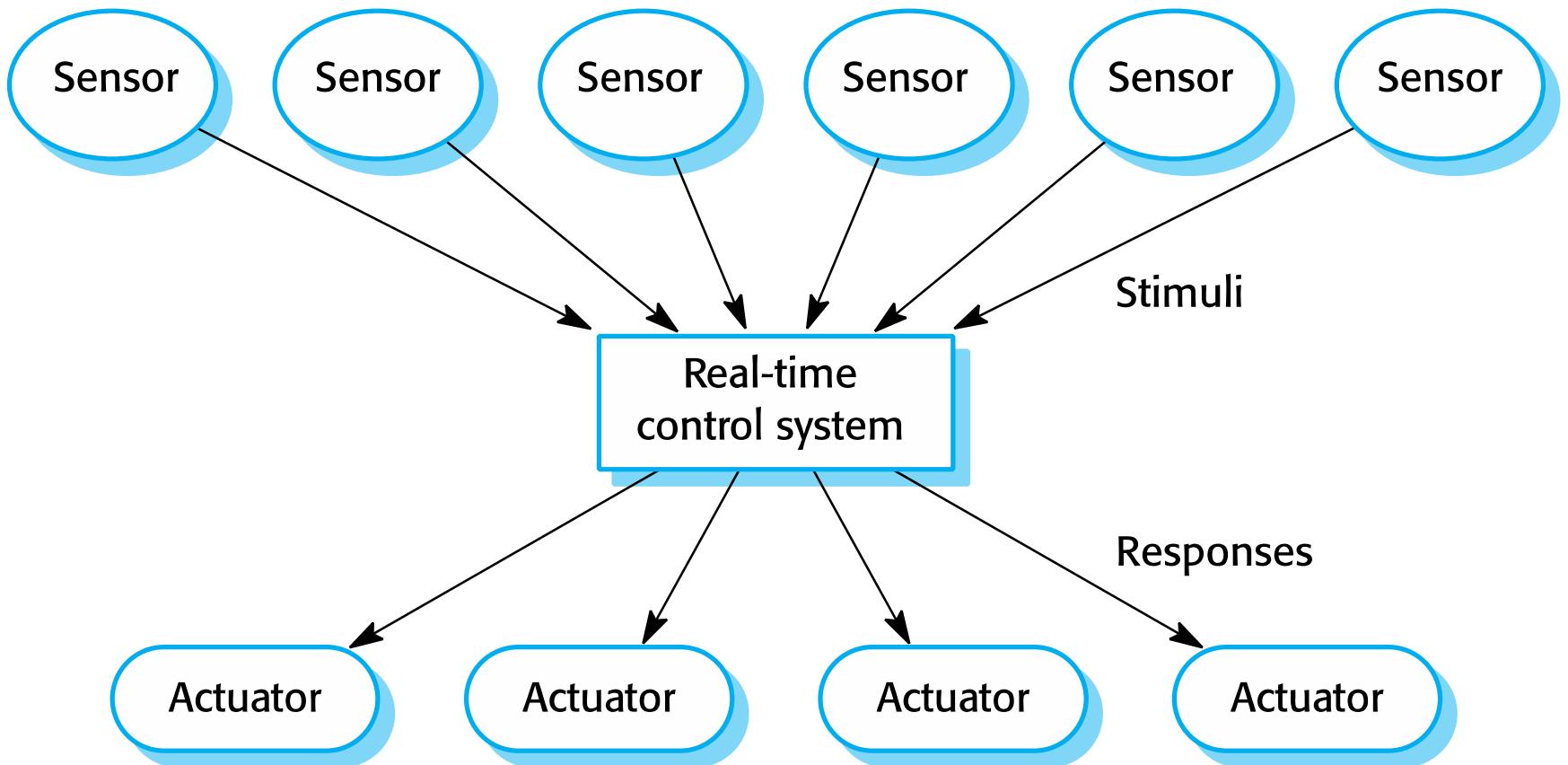


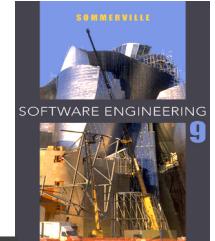
Types of stimuli

- ✧ Stimuli come from sensors in the systems environment and from actuators controlled by the system
 - *Periodic stimuli* These occur at predictable time intervals.
 - For example, the system may examine a sensor every 50 milliseconds and take action (respond) depending on that sensor value (the stimulus).
 - *Aperiodic stimuli* These occur irregularly and unpredictably and are may be signalled using the computer's interrupt mechanism.
 - An example of such a stimulus would be an interrupt indicating that an I/O transfer was complete and that data was available in a buffer.



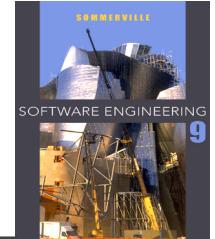
A general model of an embedded real-time system



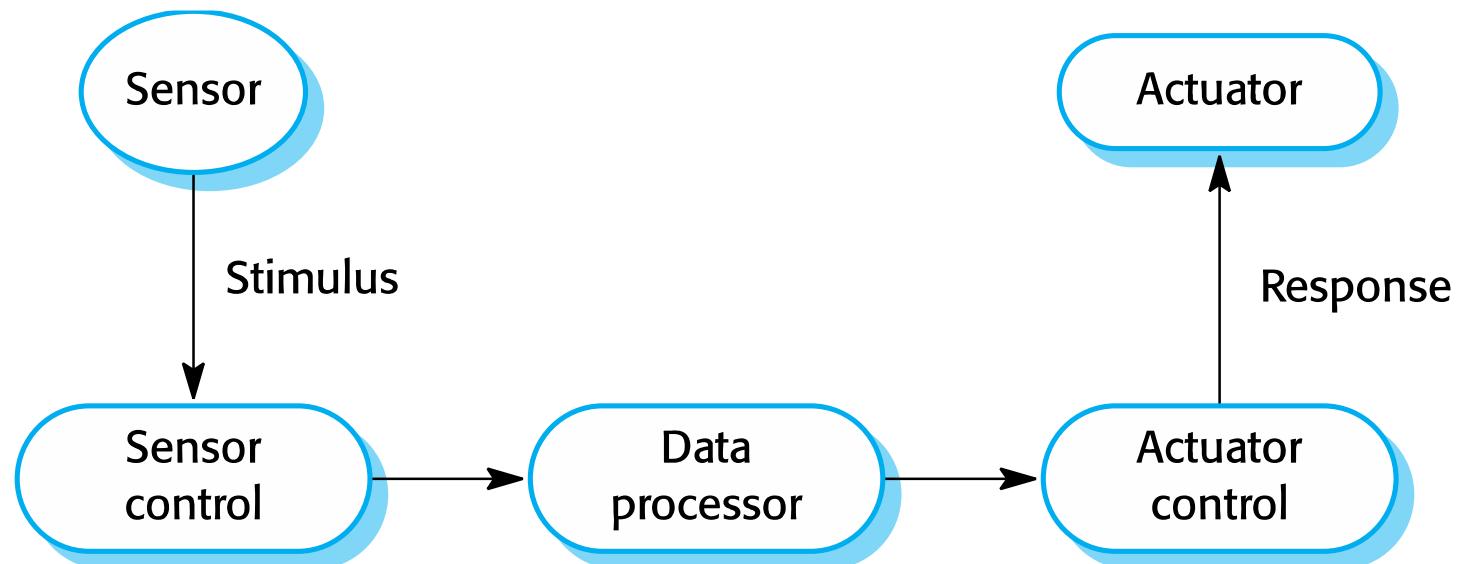


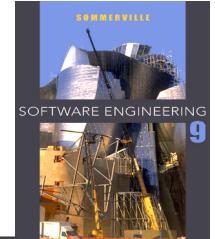
Architectural considerations

- ✧ 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

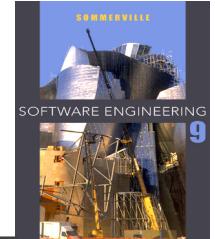
- Collect information from sensors. May buffer information collected in response to a sensor stimulus.

✧ Data processor

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

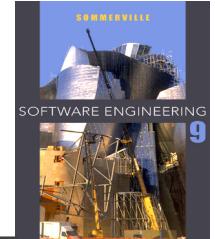
✧ Actuator control processes

- Generates control signals for the actuators.



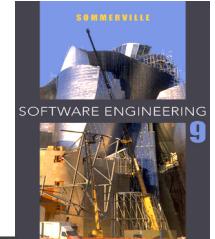
Design process activities

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



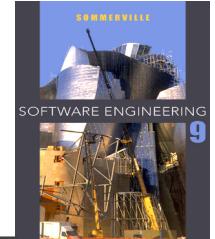
Process coordination

- ✧ 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.

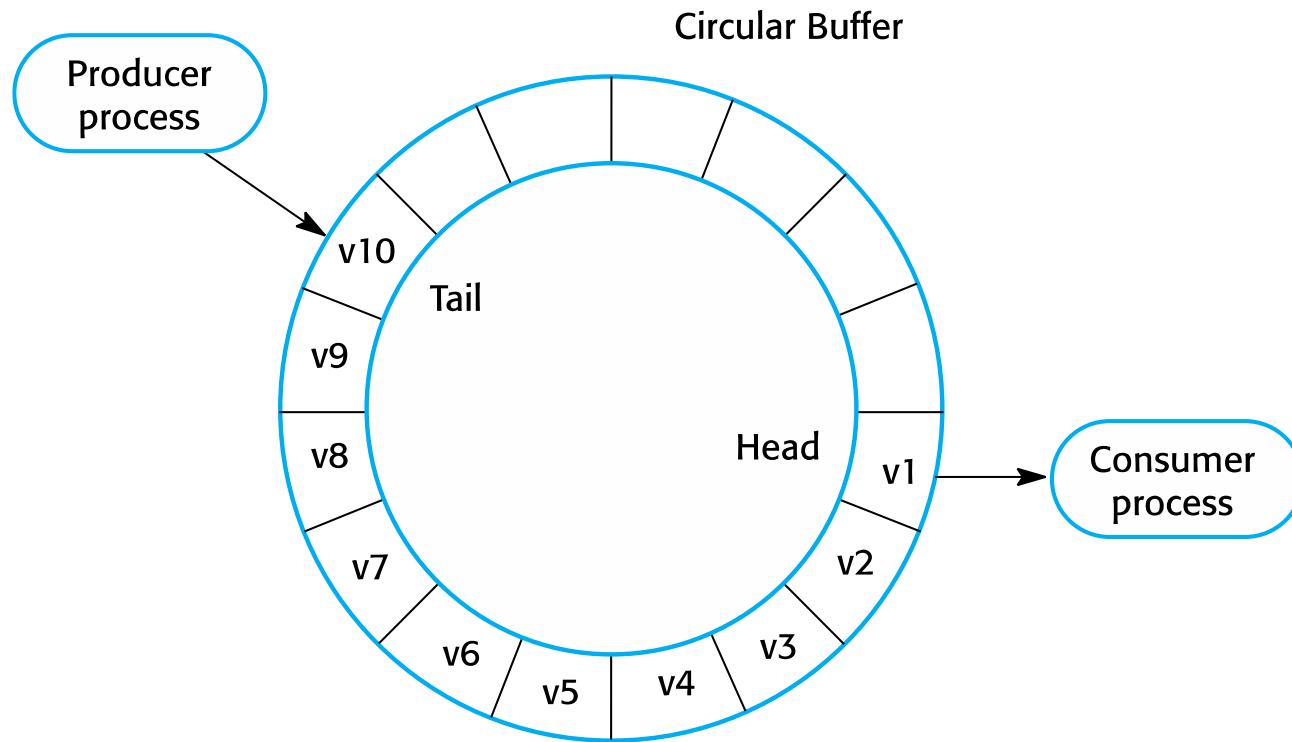


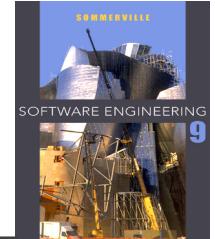
Mutual exclusion

- ✧ 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.



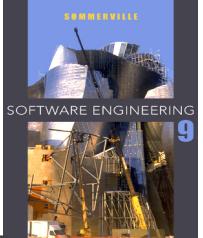
Producer/consumer processes sharing a circular buffer



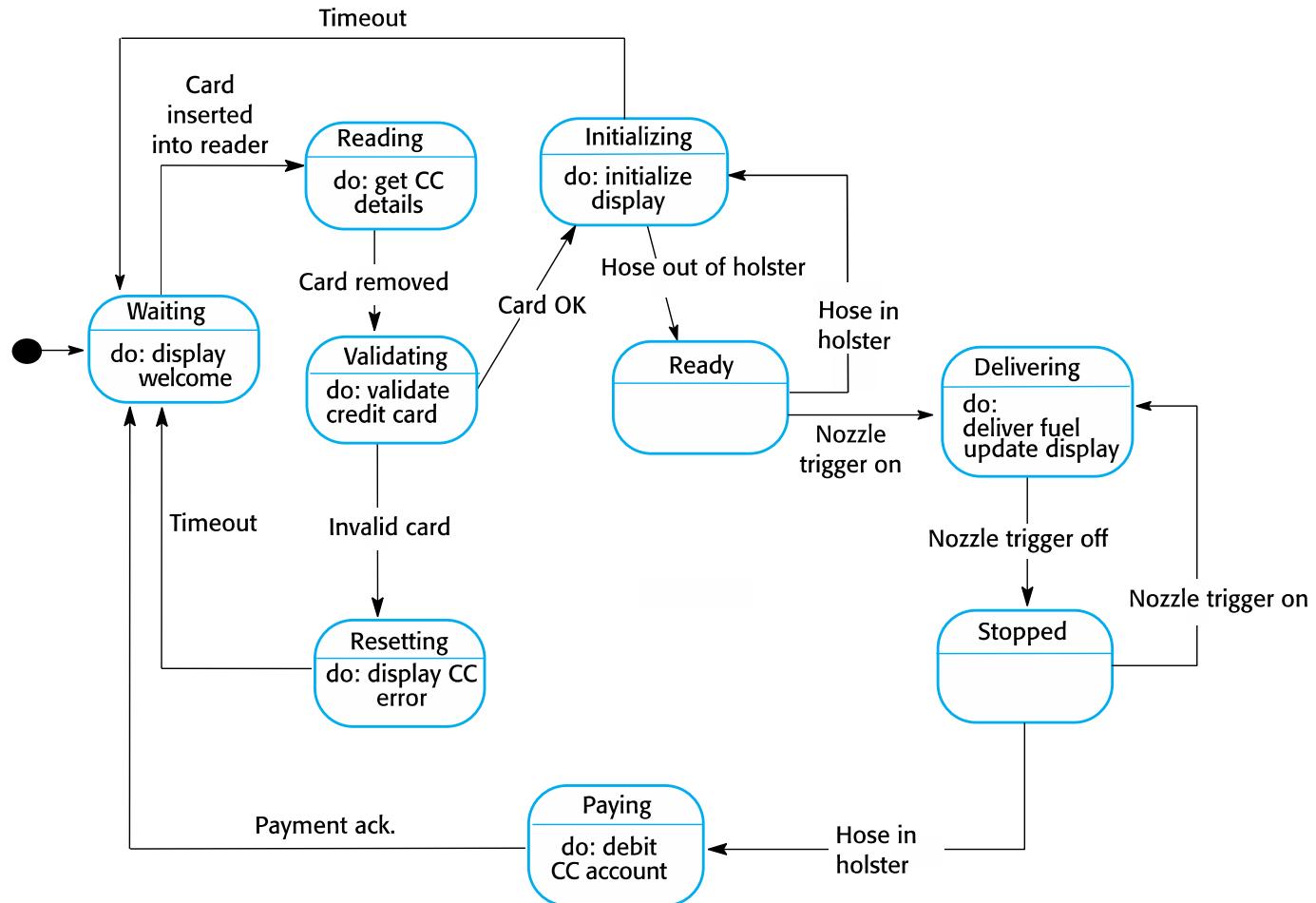


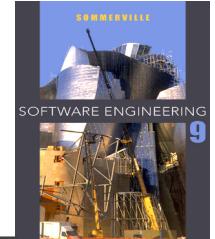
Real-time system modelling

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



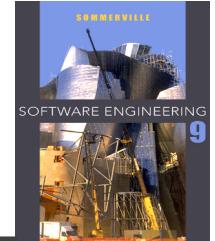
State machine model of a petrol (gas) pump





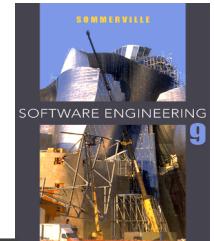
Sequence of actions in real-time pump control system

- ✧ 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

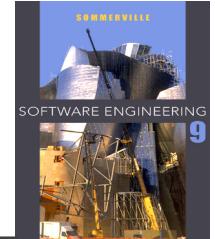


Real-time programming

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

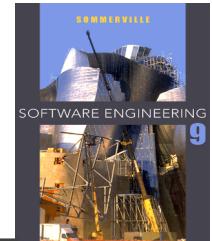


Architectural patterns for real-time software



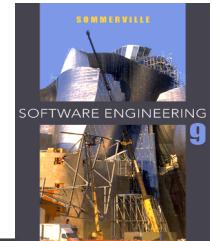
Architectural patterns for embedded systems

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

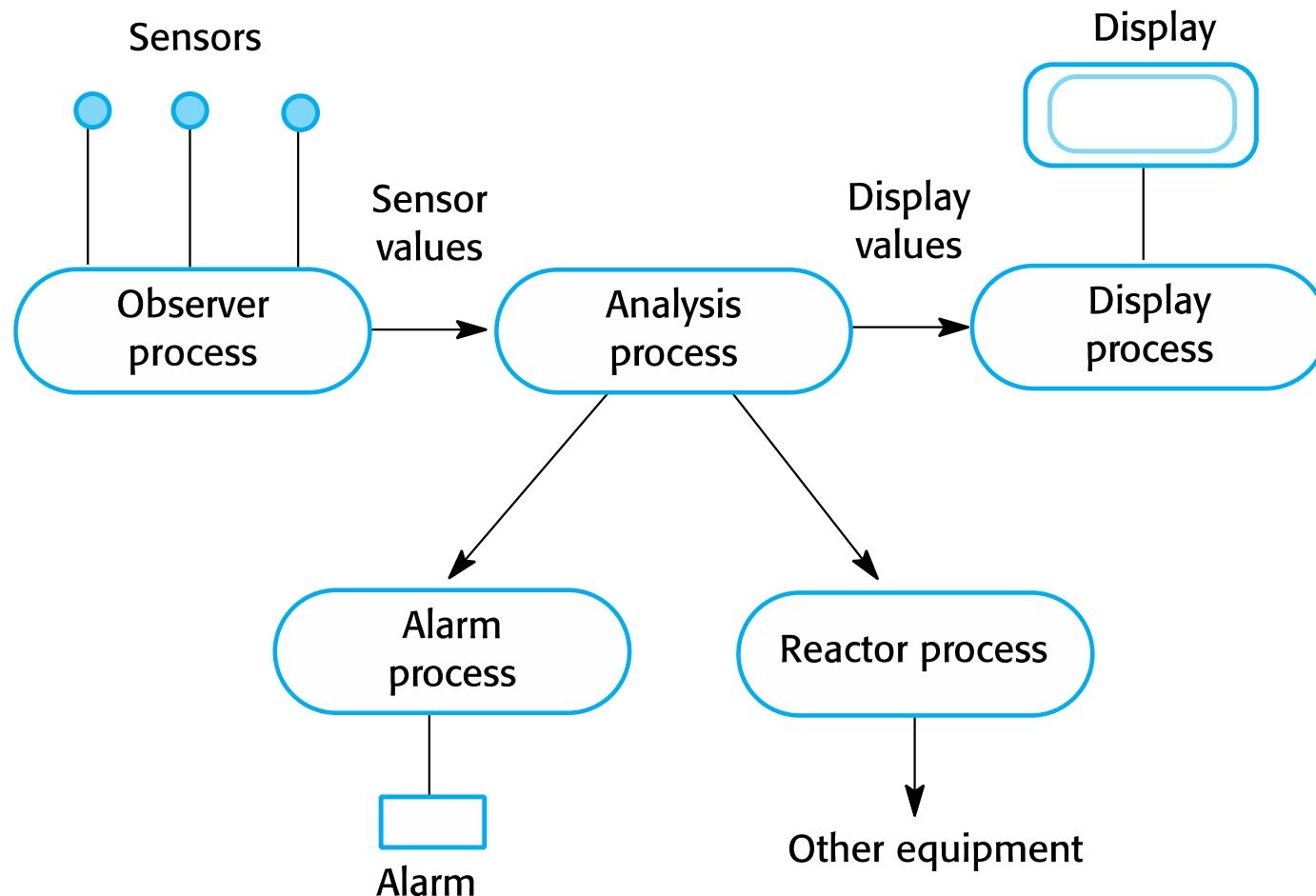


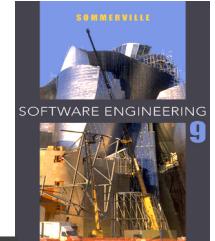
The Observe and React pattern

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



Observe and React process structure

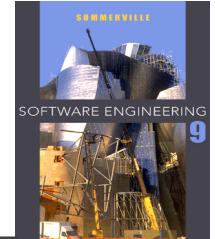




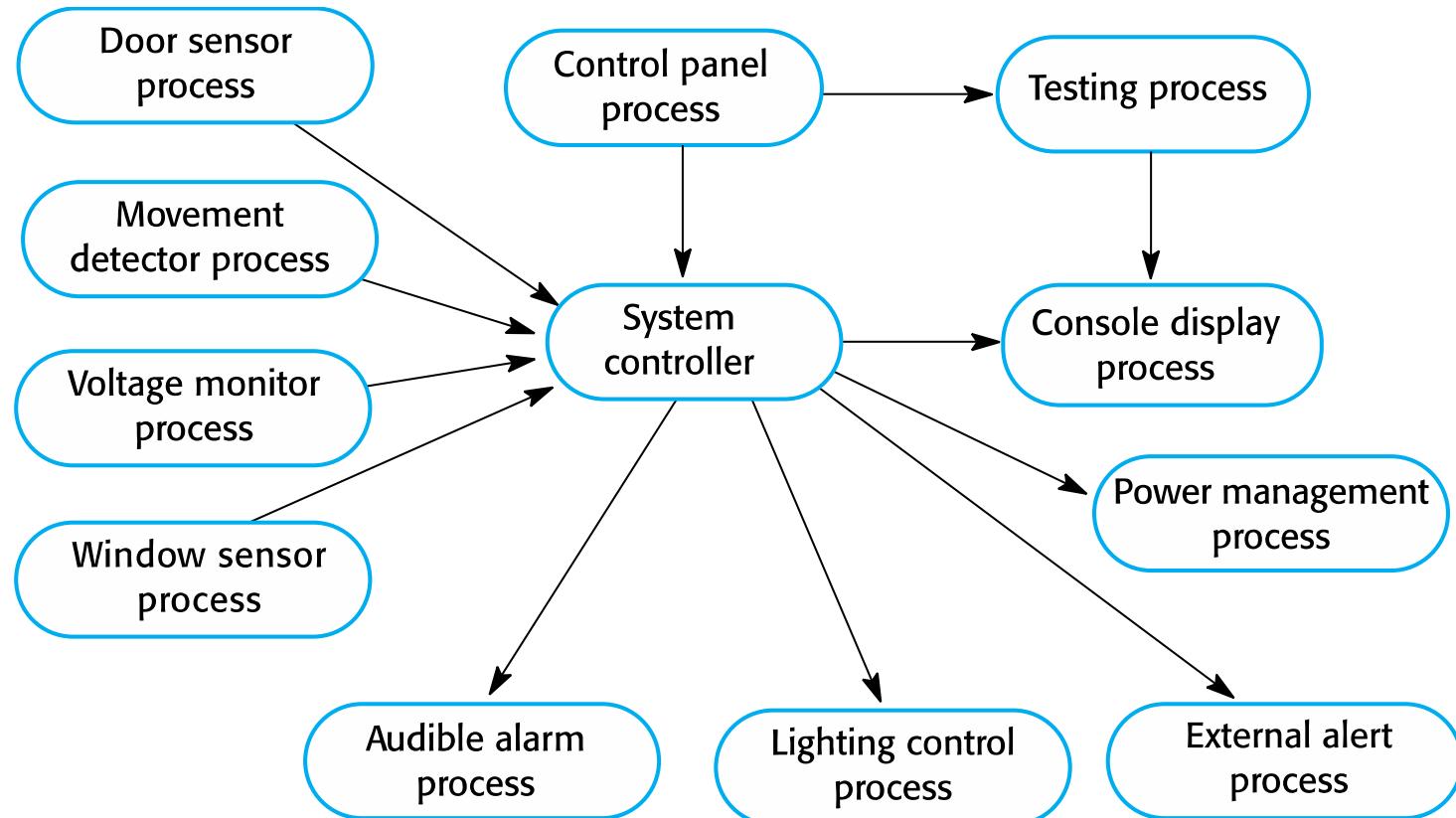
Alarm system description

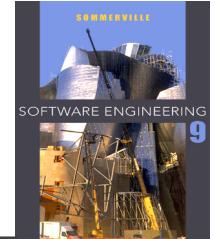
A software system is to be implemented as part of a burglar alarm system for commercial buildings. This uses several different types of sensor. These include movement detectors in individual rooms, door sensors that detect corridor doors opening, and window sensors on ground-floor windows that detect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



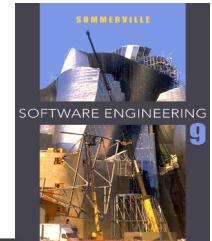
Process structure for a burglar alarm system



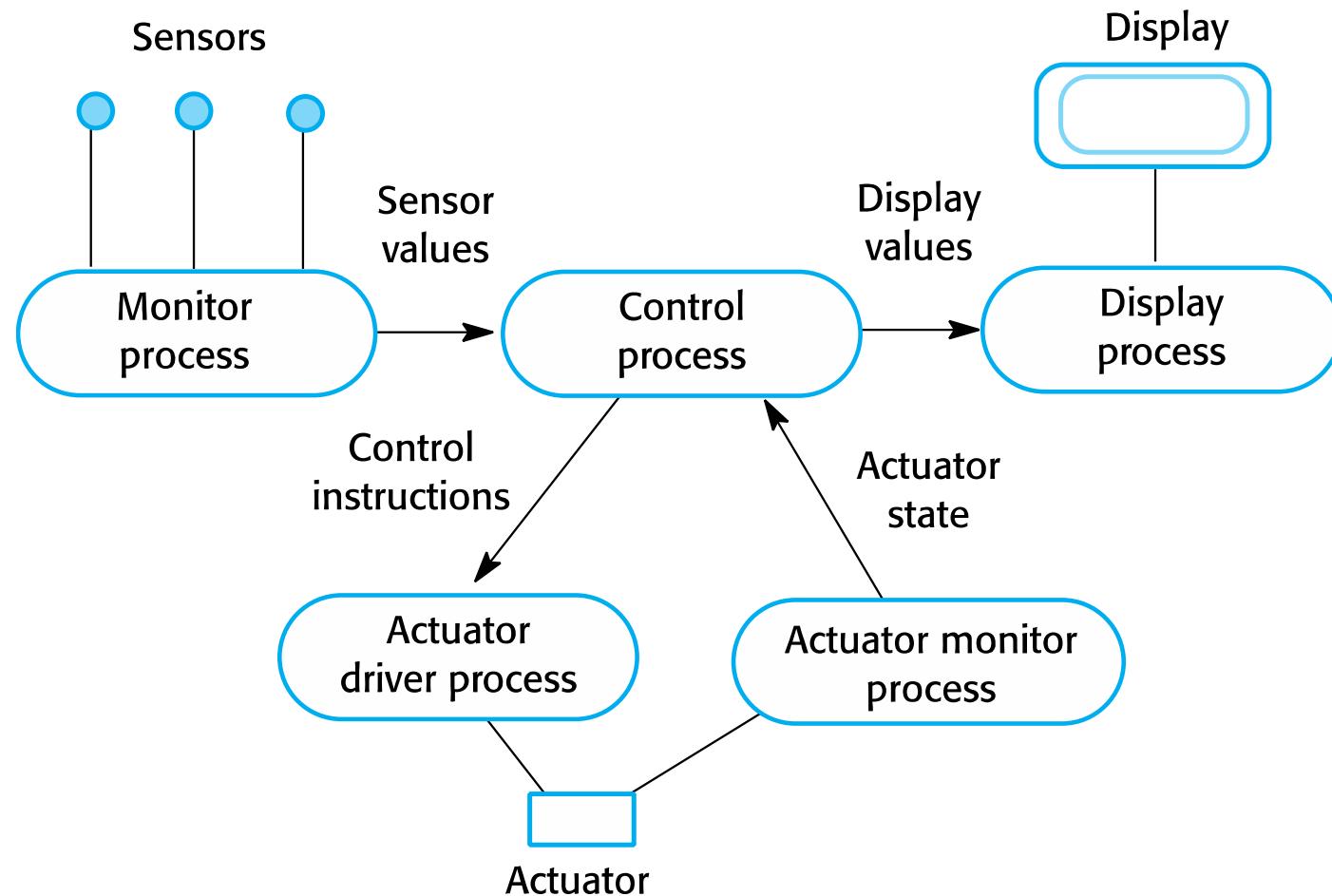


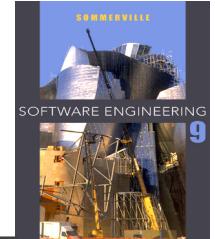
The Environmental Control pattern

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.

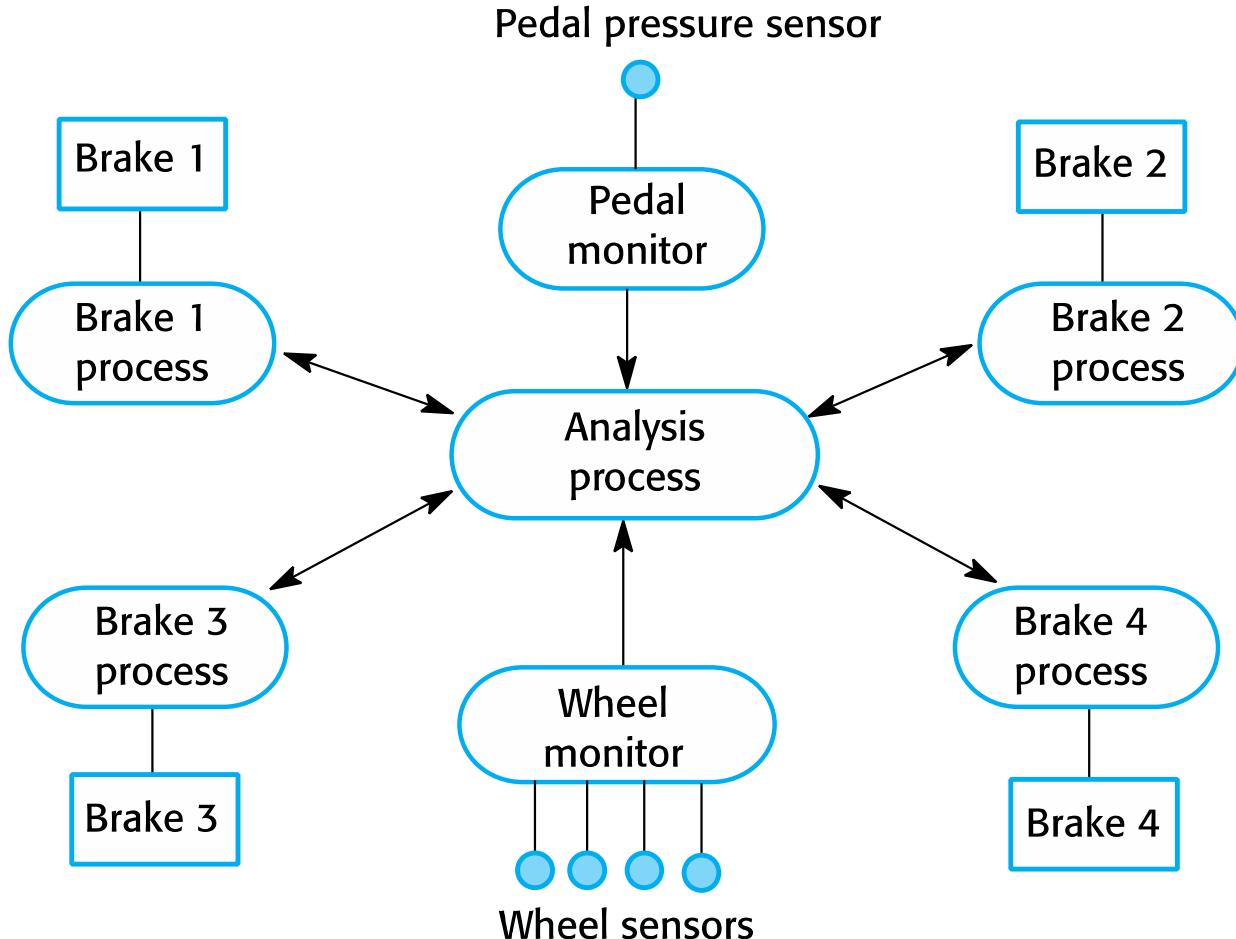


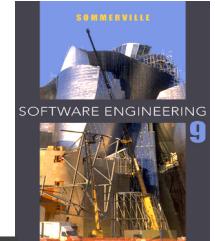
Environmental Control process structure





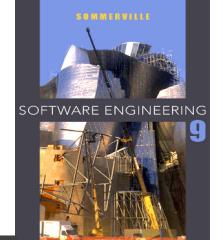
Control system architecture for an anti-skid braking system



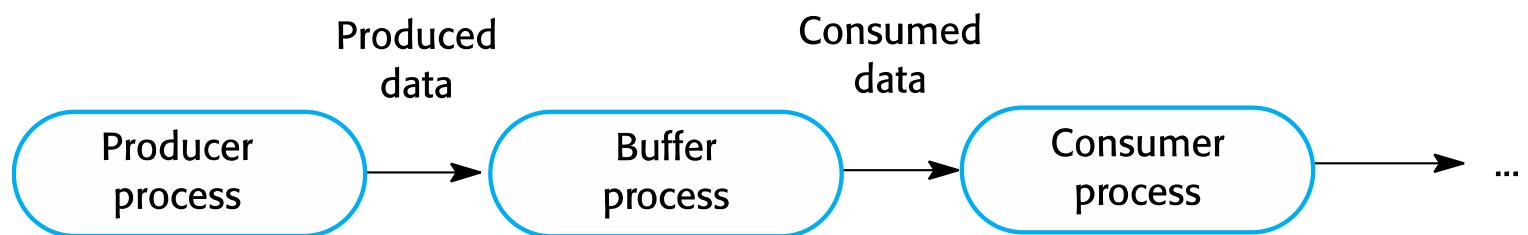


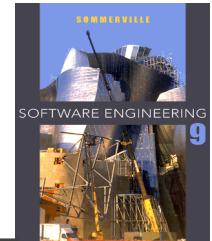
The Process Pipeline pattern

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

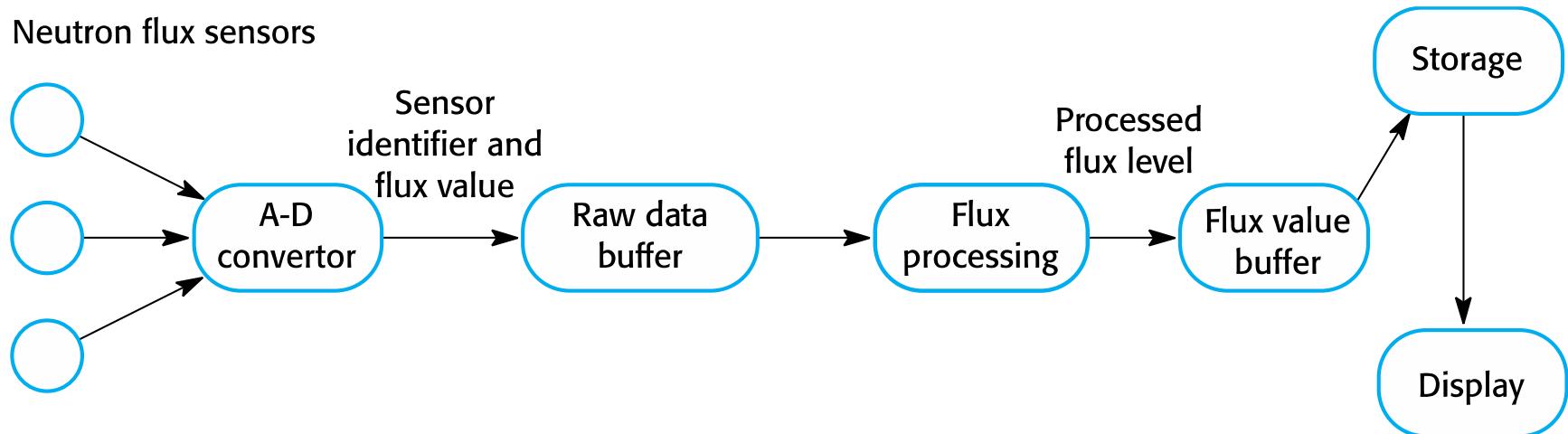


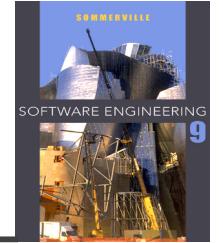
Process Pipeline process structure



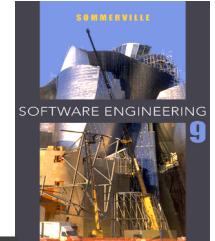


Neutron flux data acquisition



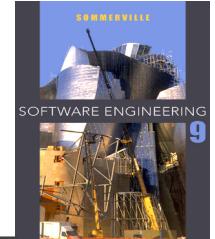


Timing analysis



Timing analysis

- ✧ The correctness of a real-time system depends not just on the correctness of its outputs but also on the time at which these outputs were produced.
- ✧ In a timing analysis, you 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.
- ✧ The results of the timing analysis are used to decide how frequently each process should execute and how these processes should be scheduled by the real-time operating system.



Factors in timing analysis

✧ *Deadlines*

- The times by which stimuli must be processed and some response produced by the system.

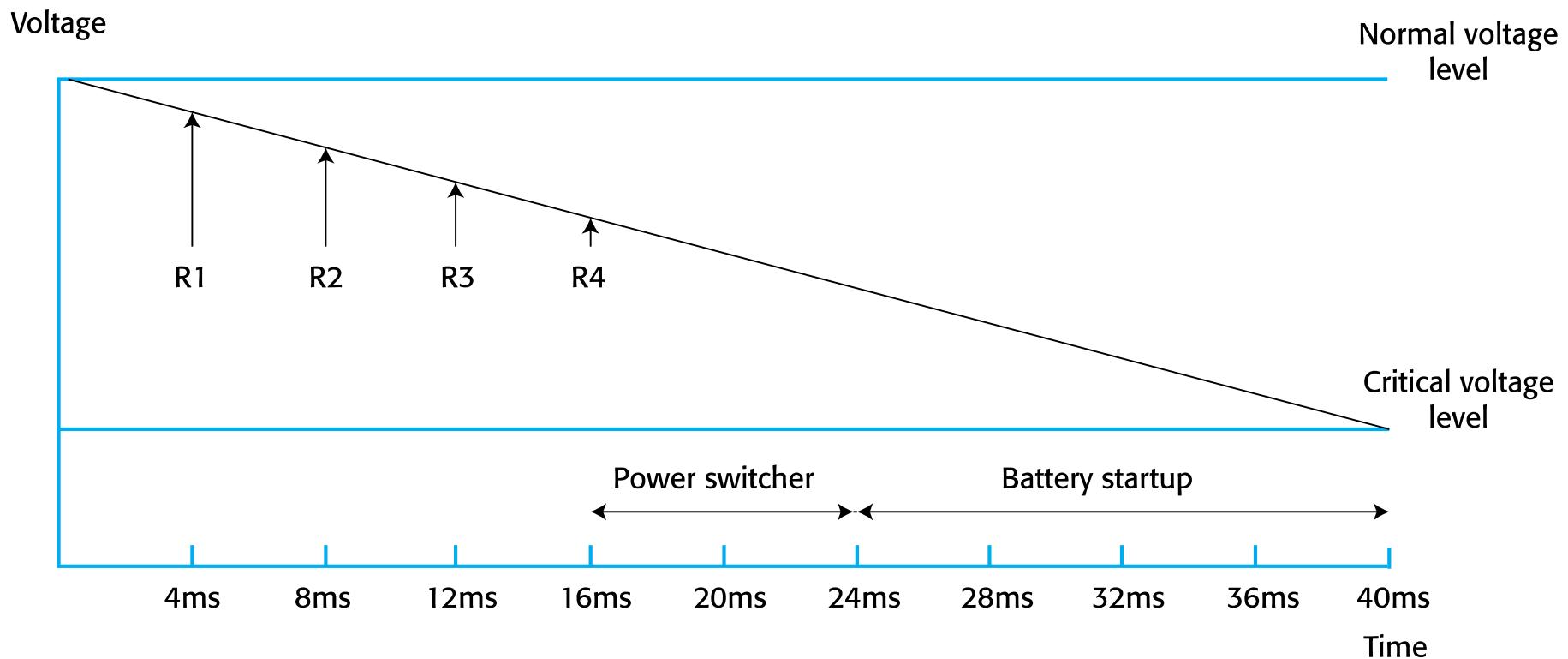
✧ *Frequency*

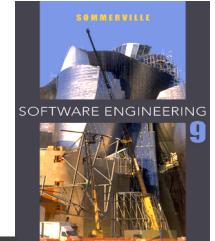
- The number of times per second that a process must execute so that you are confident that it can always meet its deadlines.

✧ *Execution time*

- The time required to process a stimulus and produce a response.

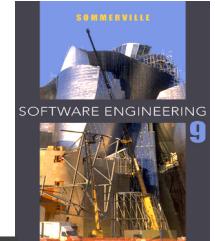
Power failure timing analysis





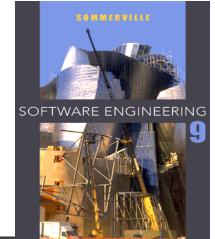
Power failure timings

- ✧ 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.



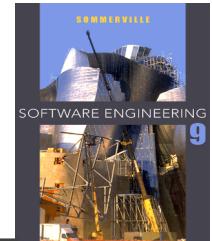
Power failure timings

- ✧ Assume the power fails immediately after a reading has been taken. Therefore reading R1 is the start reading for the power fail check. The 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.

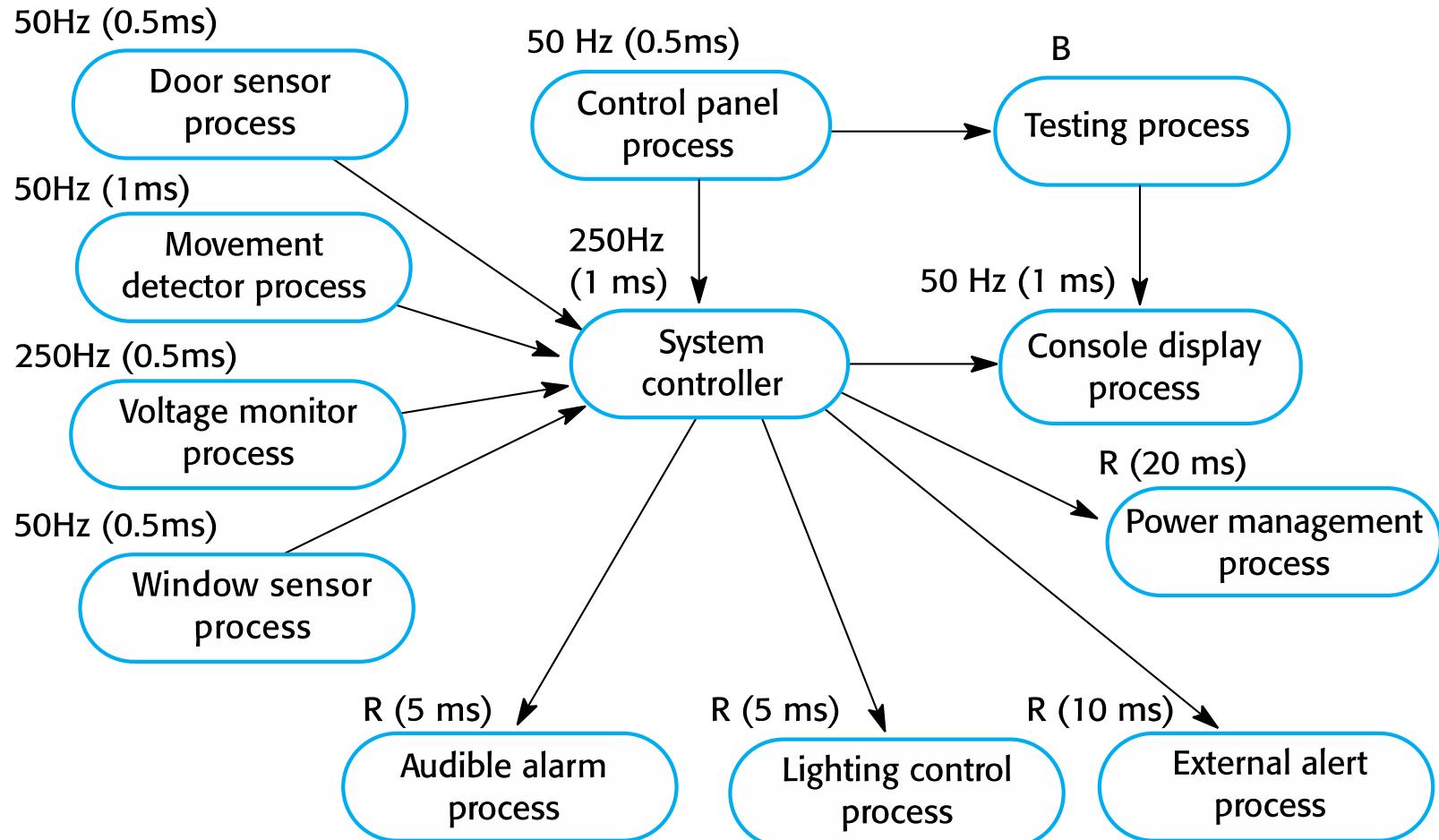


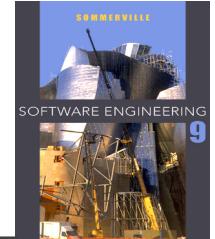
Timing requirements for the burglar alarm system

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



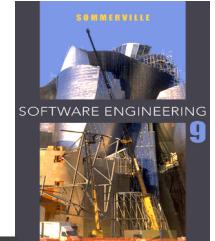
Alarm process timing





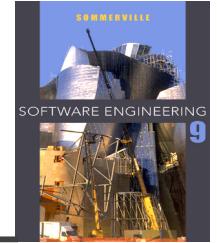
Stimuli to be processed

- ✧ 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.

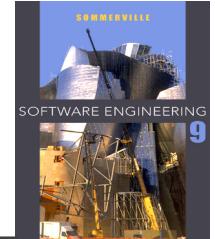


Frequency and execution time

- ✧ 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.

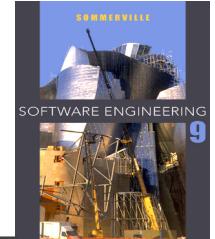


Real-time operating systems



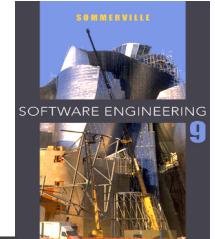
Real-time operating systems

- ✧ 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

- ✧ 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.



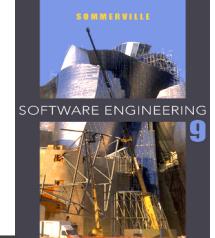
Non-stop system components

✧ Configuration manager

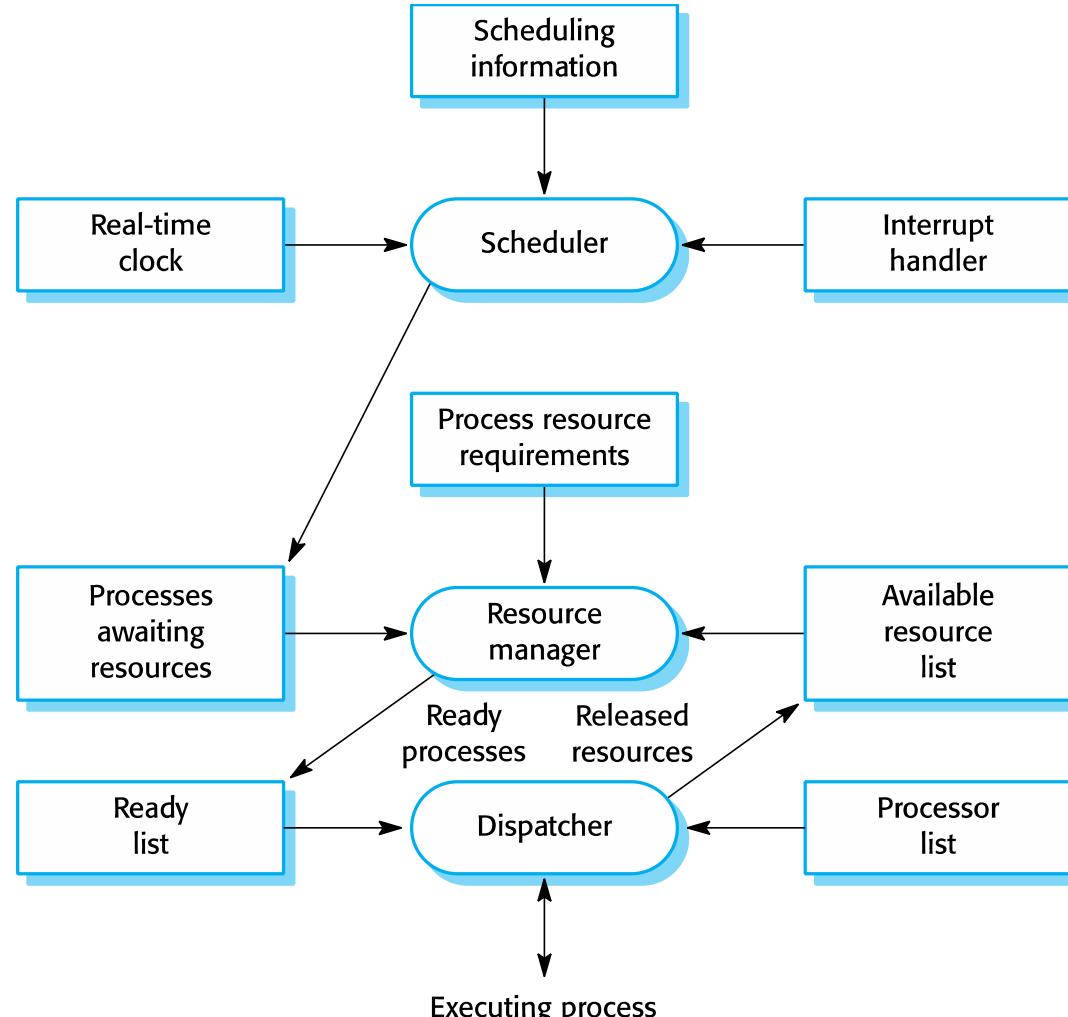
- Responsible for the dynamic reconfiguration of the system software and hardware. Hardware modules may be replaced and software upgraded without stopping the systems.

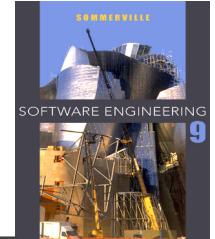
✧ Fault manager

- 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.



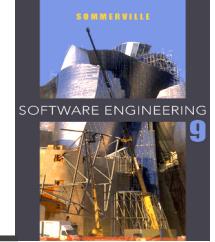
Components of a real-time operating system





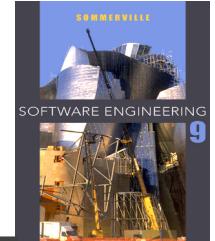
Process management

- ✧ 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.



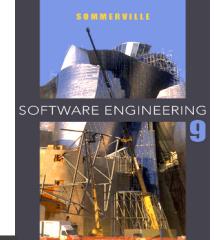
Process management

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



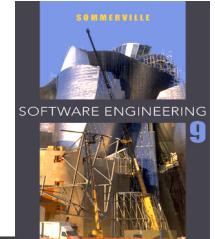
Interrupt servicing

- ✧ 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.

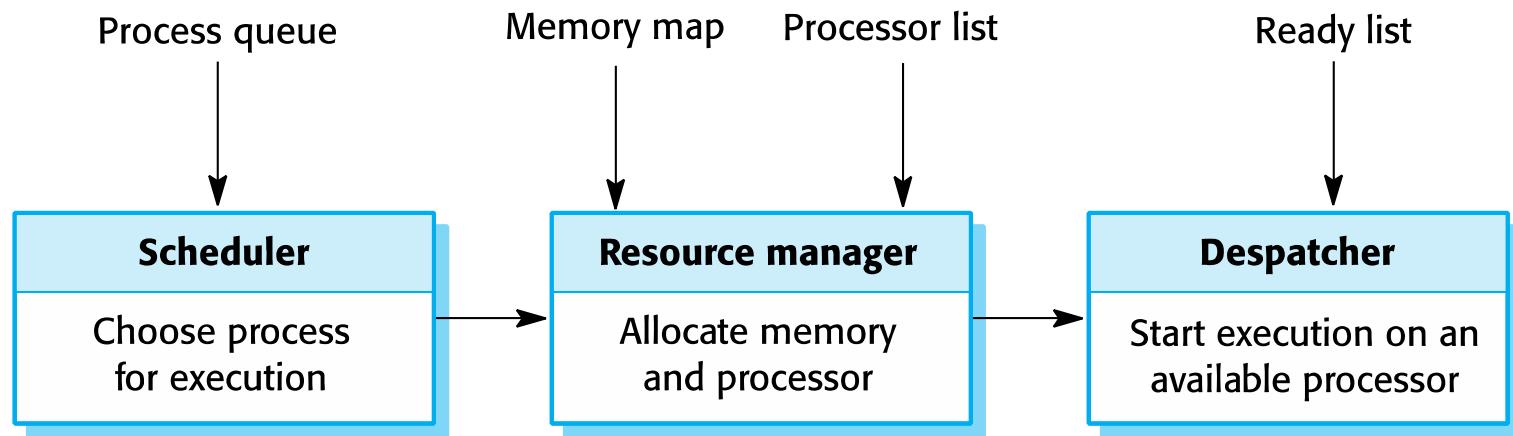


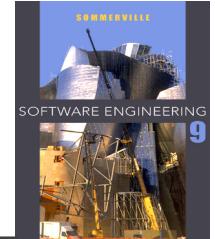
Periodic process servicing

- ✧ 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.



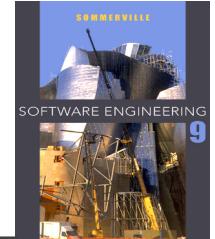
RTOS actions required to start a process





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

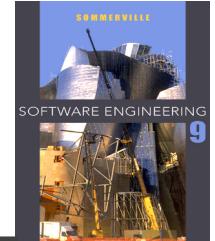
- Once a process has been scheduled for execution, it runs to completion or until it is blocked for some reason (e.g. waiting for I/O).

✧ Pre-emptive scheduling

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

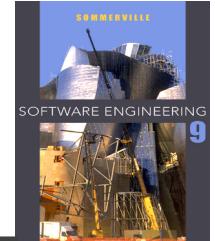
✧ Scheduling algorithms

- 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.



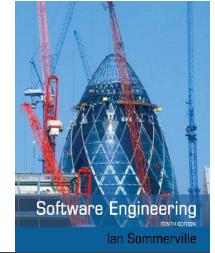
Chapter 22 – Project Management

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 coherent and well-functioning development team.

Software management distinctions



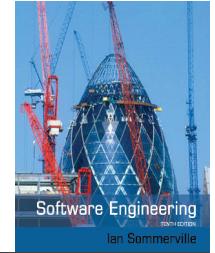
- ✧ The product is intangible.
 - Software cannot be seen or touched. Software project managers cannot see progress by simply looking at the artefact that is being constructed.
- ✧ Many software projects are 'one-off' projects.
 - Large software projects are usually different in some ways from previous 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.

Factors influencing project management



- ✧ Company size
- ✧ Software customers
- ✧ Software size
- ✧ Software type
- ✧ Organizational culture
- ✧ Software development processes
- ✧ These factors mean that project managers in different organizations may work in quite different ways.

Universal management activities



✧ *Project planning*

- Project managers are responsible for planning, estimating and scheduling project development and assigning people to tasks.
- Covered in Chapter 23.

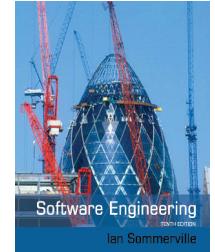
✧ *Risk management*

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

✧ *People management*

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

Management activities



✧ *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.

✧ *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



- ✧ Risk management is concerned with identifying risks and drawing up plans to minimise their effect on a project.
- ✧ Software risk management is important because of the inherent uncertainties in software development.
 - These uncertainties stem from loosely defined requirements, requirements changes due to changes in customer needs, difficulties in estimating the time and resources required for software development, and differences in individual skills.
- ✧ You have to anticipate risks, understand the impact of these risks on the project, the product and the business, and take steps to avoid these risks.

Risk classification



- ✧ There are two dimensions of risk classification
 - The type of risk (technical, organizational, ..)
 - what is affected by the risk:
- ✧ *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.

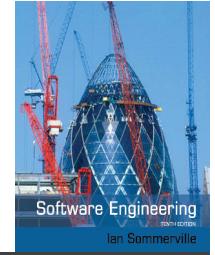


Examples of project, product, and business risks

Software Engineering
Ian Sommerville

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;

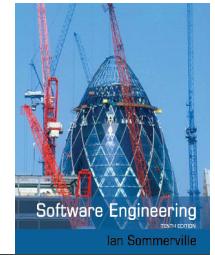
✧ Risk planning

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

✧ Risk monitoring

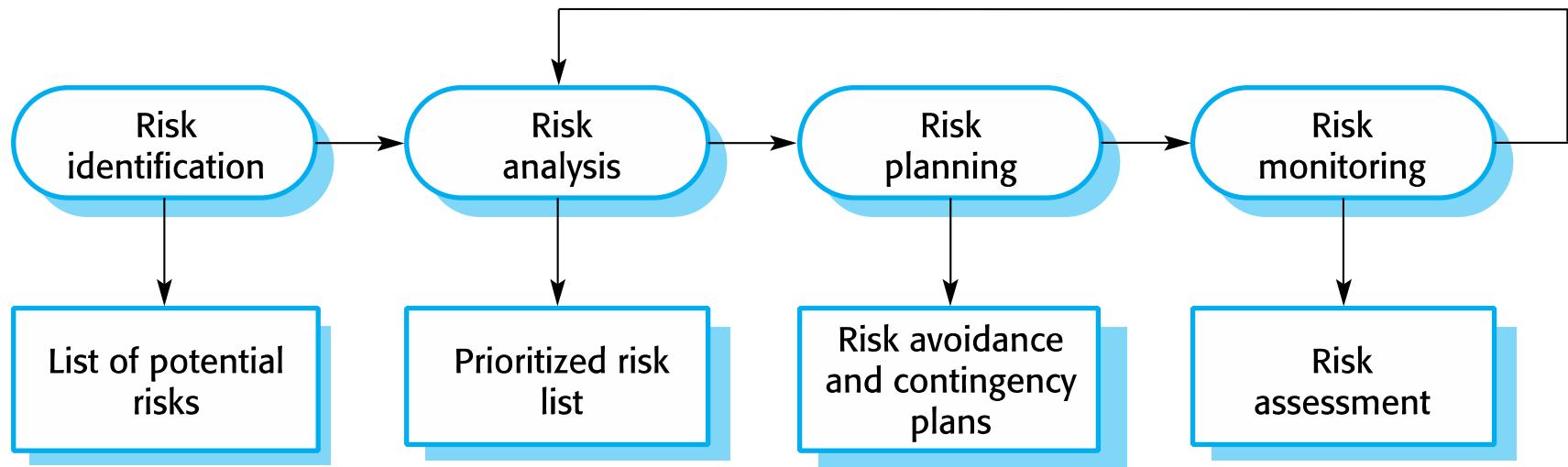
- Monitor the risks throughout the project;

The risk management process

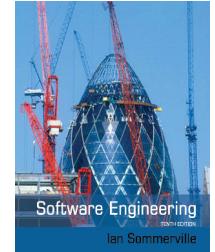


Software Engineering

Ian Sommerville

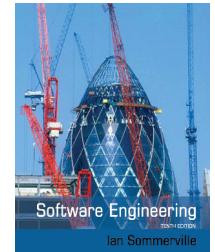


Risk identification



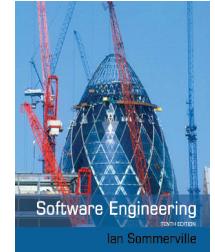
- ✧ May be a team activities or based on the individual project manager's experience.
- ✧ A checklist of common risks may be used to identify risks in a project
 - Technology risks.
 - Organizational risks.
 - People risks.
 - Requirements risks.
 - Estimation risks.

Examples of different risk types



Risk type	Possible risks
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)
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)
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)
Requirements	Changes to requirements that require major design rework are proposed. (10) Customers fail to understand the impact of requirements changes. (11)
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)
Tools	The code generated by software code generation tools is inefficient. (8) Software tools cannot work together in an integrated way. (9)

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



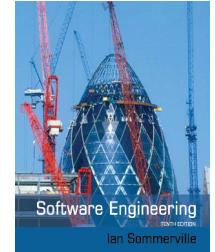
Risk	Probability	Effects
The time required to develop the software is underestimated (12).	High	Serious
Software tools cannot be integrated (9).	High	Tolerable
Customers fail to understand the impact of requirements changes (11).	Moderate	Tolerable
Required training for staff is not available (5).	Moderate	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



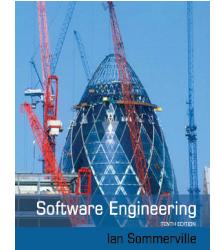
- ✧ Consider each risk and develop a strategy to manage that risk.
- ✧ Avoidance strategies
 - The probability that the risk will arise is reduced;
- ✧ Minimization strategies
 - The impact of the risk on the project or product will be reduced;
- ✧ Contingency plans
 - If the risk arises, contingency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

What-if questions



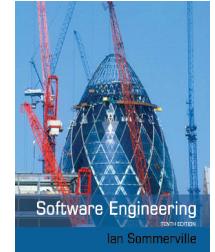
- ✧ What if several engineers are ill at the same time?
- ✧ What if an economic downturn leads to budget cuts of 20% for the project?
- ✧ What if the performance of open-source software is inadequate and the only expert on that open source software leaves?
- ✧ What if the company that supplies and maintains software components goes out of business?
- ✧ What if the customer fails to deliver the revised requirements as predicted?

Strategies to help manage risk



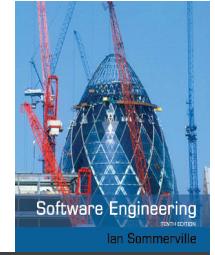
Risk	Strategy
Organizational financial problems	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business and presenting reasons why cuts to the project budget would not be cost-effective.
Recruitment problems	Alert 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

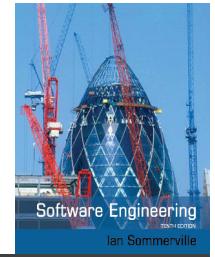


Risk type	Potential indicators
Estimation	Failure to meet agreed schedule; failure to clear reported defects.
Organizational	Organizational gossip; lack of action by senior management.
People	Poor staff morale; poor relationships amongst team members; high staff turnover.
Requirements	Many requirements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



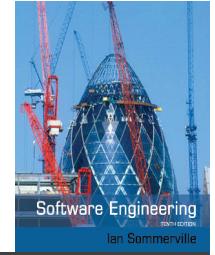
Managing people

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



Software Engineering
Ian Sommerville

✧ 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.

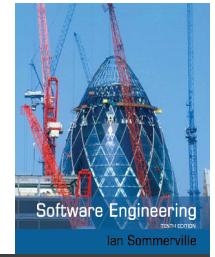
✧ Inclusion

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

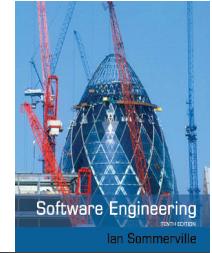
✧ Honesty

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

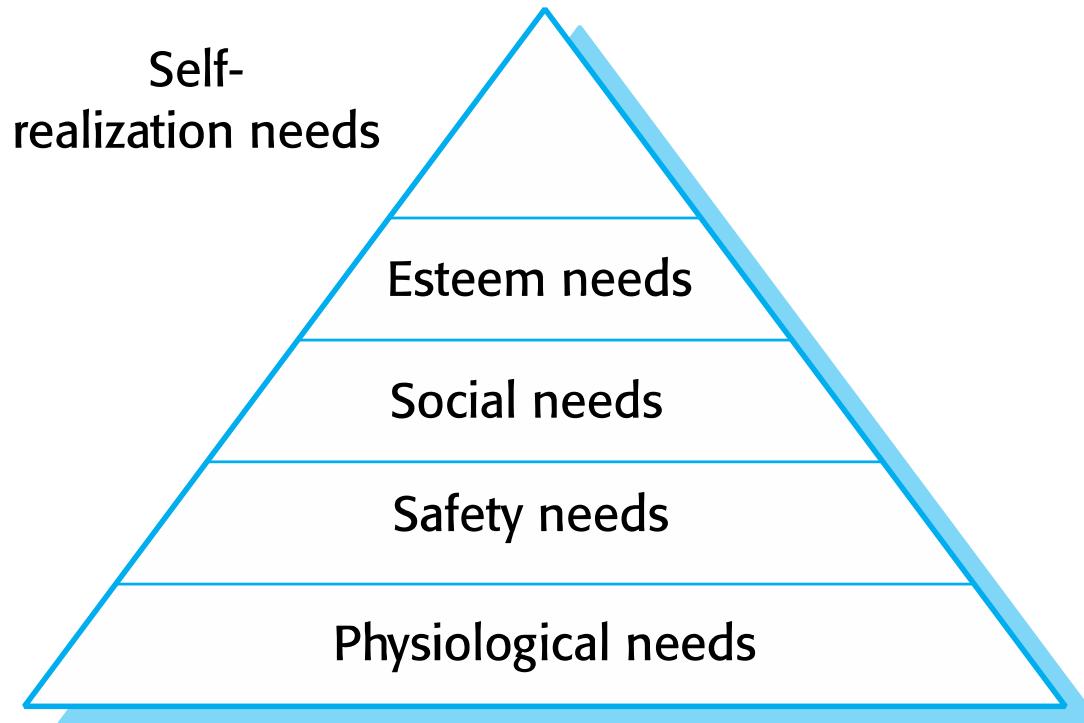
Motivating people



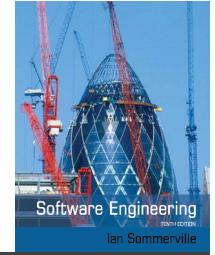
- ✧ An important role of a manager is to motivate the people working on a project.
- ✧ Motivation means organizing the work and the working environment to encourage people to work effectively.
 - If people are not motivated, they will not be interested in the work they are doing. They will work slowly, 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

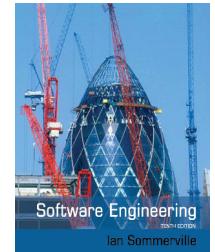


Need satisfaction



- ✧ 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.

Case study: 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.

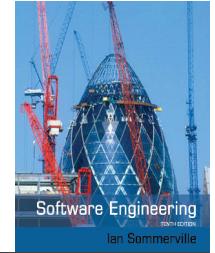
Alice talks about the problem informally with other team members to try to find out if Dorothy's personal circumstances have changed, and if this might be affecting her work. They don't know of anything, so Alice decides to talk with Dorothy to try to understand the problem.

Case study: 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.

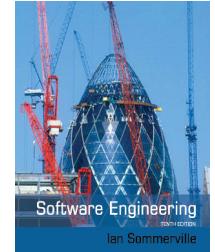
Although she admits that the work is challenging, she is concerned that she is not developing her interfacing skills. She is worried that finding a job that involves hardware interfacing will be difficult after this project. Because she does not want to upset the team by revealing that she is thinking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



Comments on case study

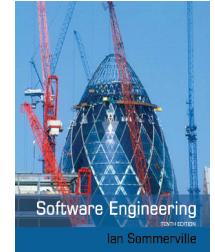
- ✧ If you don't sort out the problem of unacceptable work, the other group members will become dissatisfied and feel that they are doing an unfair share of the work.
- ✧ Personal difficulties affect motivation because people can't concentrate on their work. They need time and support to resolve these issues, although you have to make clear that they still have a responsibility to their employer.
- ✧ Alice gives Dorothy more design autonomy and organizes training courses in software engineering that will give her more opportunities after her current project has finished.

Personality types



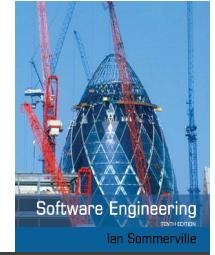
- ✧ The needs hierarchy is almost certainly an over-simplification of motivation in practice.
- ✧ Motivation should also take into account different personality types:
 - Task-oriented people, who are motivated by the work they do. In software engineering.
 - Interaction-oriented people, who are motivated by the presence and actions of co-workers.
 - Self-oriented people, who are principally motivated by personal success and recognition.

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.

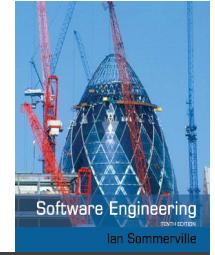


Software Engineering

Ian Sommerville

Teamwork

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.

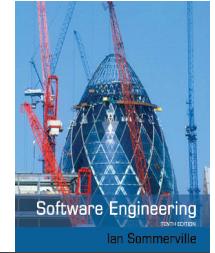
Team spirit



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

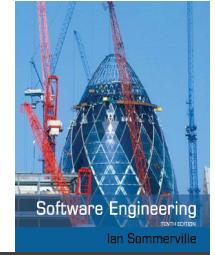
- Good communications between group members, and between the software engineering team and other project stakeholders, is essential.

Selecting group members



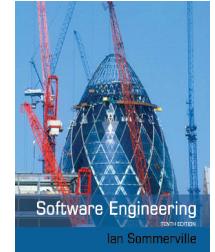
- ✧ A manager or team leader's job is to create a cohesive group and organize their group so that they can work together effectively.
- ✧ This involves creating a group with the right balance of technical skills and personalities, and organizing that group so that the members work together effectively.

Assembling a team



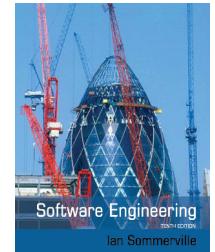
- ✧ May not be possible to appoint the ideal people to work on a project
 - Project budget may not allow for the use of highly-paid staff;
 - Staff with the appropriate experience may not be available;
 - An organisation may wish to develop employee skills on a software project.
- ✧ Managers have to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

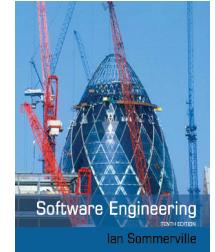
- Alice—self-oriented
- Brian—task-oriented
- Bob—task-oriented
- Carol—interaction-oriented
- Dorothy—self-oriented
- Ed—interaction-oriented
- Fred—task-oriented

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



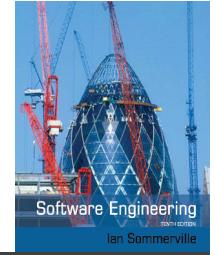
- ✧ Small software engineering groups are usually organised informally without a rigid structure.
- ✧ For large projects, there may be a hierarchical structure where different groups are responsible for different sub-projects.
- ✧ Agile development is always based around an informal group on the principle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

Informal groups

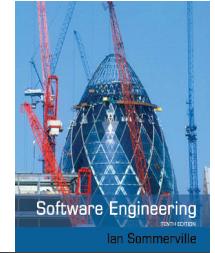


- ✧ The group acts as a whole and comes to a consensus on decisions affecting the system.
- ✧ The group leader serves as the external interface of the group but does not allocate specific work items.
- ✧ Rather, work is discussed by the group as a whole and tasks are allocated according to ability and 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



- ✧ 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 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.



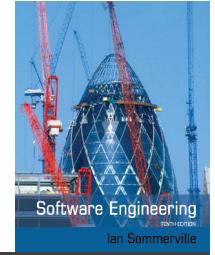
Key points

- ✧ People management involves choosing the right people to work on a project and organizing the team and its working environment.
- ✧ 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.



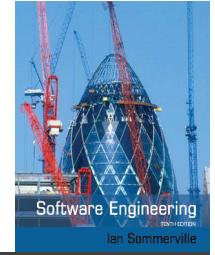
Chapter 23 – Project planning

Topics covered



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

Project planning



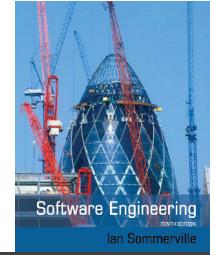
- ✧ Project planning involves breaking down the work into parts and assign these to project team members, anticipate problems that might arise and prepare tentative solutions to those problems.
- ✧ The project plan, which is created at the start of a project, is used to communicate how the work will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

Planning stages



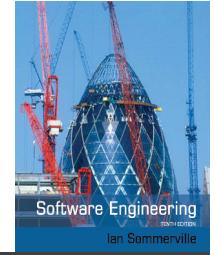
- ✧ At the proposal stage, when you are bidding for a contract to develop or provide a software system.
- ✧ During the project startup phase, when you have to plan who will work on the project, how the project will be broken down into increments, how resources will be allocated across your company, etc.
- ✧ 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



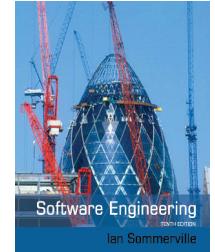
- ✧ Planning may be necessary with only outline software requirements.
- ✧ The aim of planning at this stage is to provide information that will be used in setting a price for the system to customers.
- ✧ Project pricing involves estimating how much the software will cost to develop, taking factors such as staff costs, hardware costs, software costs, etc. into account

Project startup planning



- ✧ At this stage, you know more about the system requirements but do not have design or implementation information
- ✧ Create a plan with enough detail to make decisions about the project budget and staffing.
 - This plan is the basis for project resource allocation
- ✧ The startup plan should also define project monitoring mechanisms
- ✧ A startup plan is still needed for agile development to allow resources to be allocated to the project

Development planning

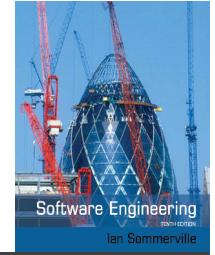


- ✧ The project plan should be regularly amended as the project progresses and you know more about the software and its development
- ✧ The project schedule, cost-estimate and risks have to be regularly revised



Software pricing

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



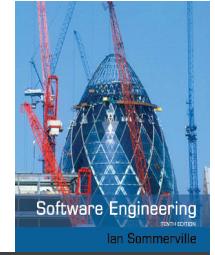
Factor	Description
Contractual terms	A customer may be willing to allow the developer to retain ownership of the source code and reuse it in other projects. The price charged may then be less than if the software source code is handed over to the customer.
Cost estimate uncertainty	If an organization is unsure of its cost estimate, it may increase its price by a contingency over and above its normal profit.
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.

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.
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.

Pricing strategies



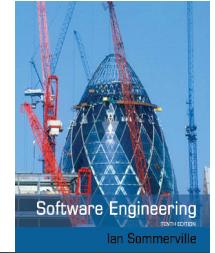
✧ Under pricing

- A company may underprice a system in order to gain a contract that allows them to retain staff for future opportunities
- A company may underprice a system to gain access to a new market area

✧ Increased pricing

- The price may be increased when a buyer wishes a fixed-price contract and so the seller increases the price to allow for unexpected risks

Pricing to win



- ✧ The software is priced according to what the software developer believes the buyer is willing to pay
- ✧ If this is less than the development costs, the software functionality may be reduced accordingly with a view to extra functionality being added in a later release
- ✧ Additional costs may be added as the requirements change and these may be priced at a higher level to make up the shortfall in the original price

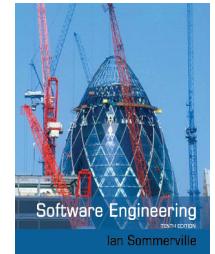


Software Engineering

Ian Sommerville

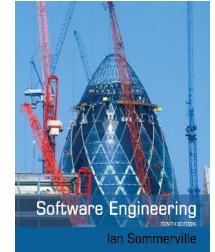
Plan-driven development

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



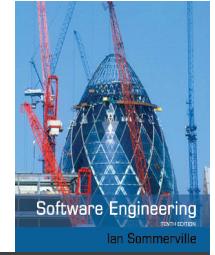
- ✧ In a plan-driven development project, a project plan sets out the resources available to the project, the work breakdown and a schedule for carrying out the work.
- ✧ Plan sections
 - Introduction
 - Project organization
 - Risk analysis
 - Hardware and software resource requirements
 - Work breakdown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

Project plan supplements



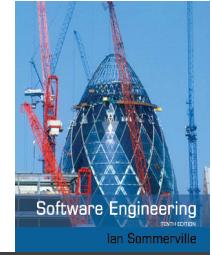
Plan	Description
Configuration management plan	Describes the configuration management procedures and structures to be used.
Deployment plan	Describes how the software and associated hardware (if required) will be deployed in the customer's environment. This should include a plan for migrating data from existing systems.
Maintenance plan	Predicts the maintenance requirements, costs, and effort.
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.

The planning process



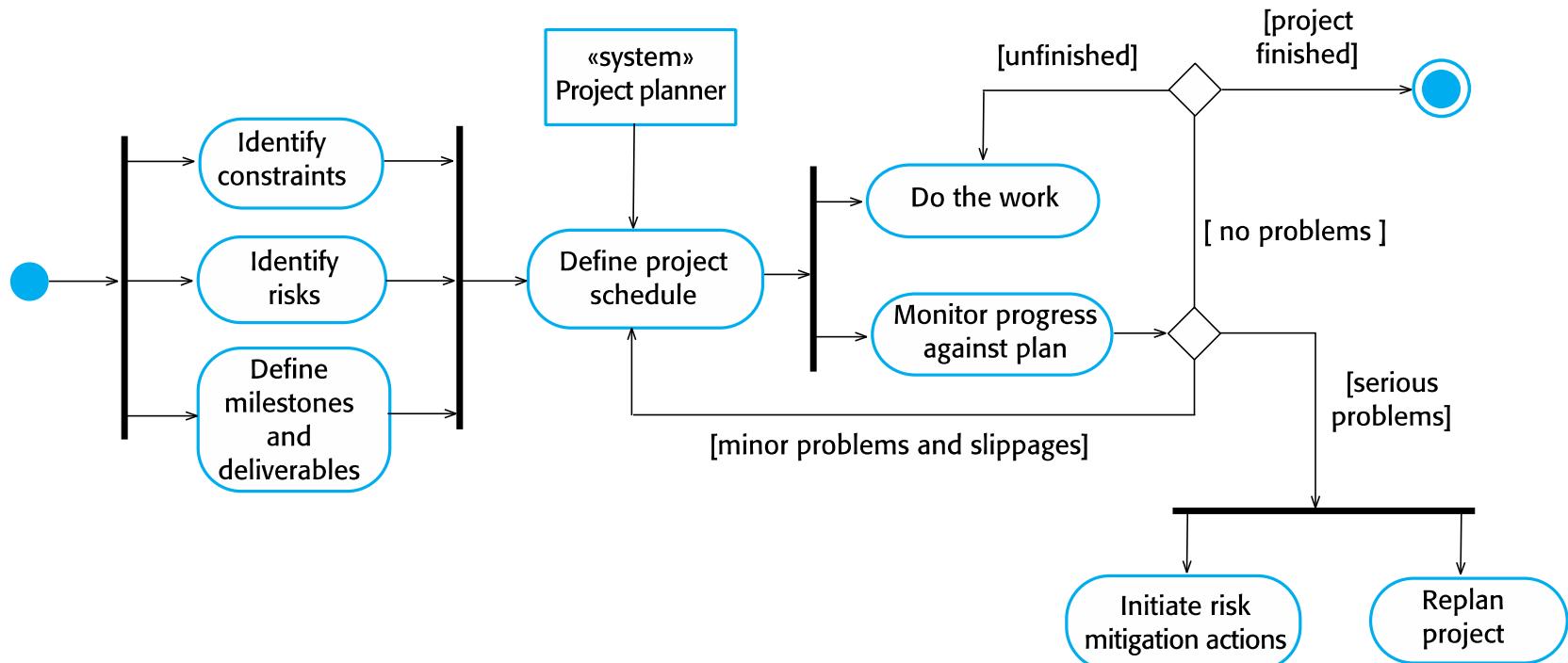
- ✧ Project planning is an iterative process that starts when you create an initial project plan during the project startup phase.
- ✧ Plan changes are inevitable.
 - As more information about the system and the project team becomes available during the project, you should regularly revise the plan to 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



Software Engineering

Ian Sommerville

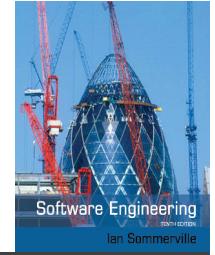


Planning assumptions



- ✧ You should make realistic rather than optimistic assumptions when you are defining a project plan.
- ✧ Problems of some description always arise during a project, and these lead to project delays.
- ✧ Your initial assumptions and scheduling should therefore take unexpected problems into account.
- ✧ You should include contingency in your plan so that if things go wrong, then your delivery schedule is not seriously disrupted.

Risk mitigation



- ✧ If there are serious problems with the development work that are likely to lead to significant delays, you need to initiate risk mitigation actions to reduce the risks of project failure.
- ✧ In conjunction with these actions, you also have to re-plan the project.
- ✧ This may involve renegotiating the project constraints and deliverables with the customer. A new schedule of when work should be completed also has to be established and agreed with the customer.



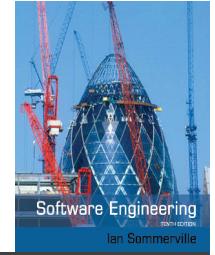
Project scheduling

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



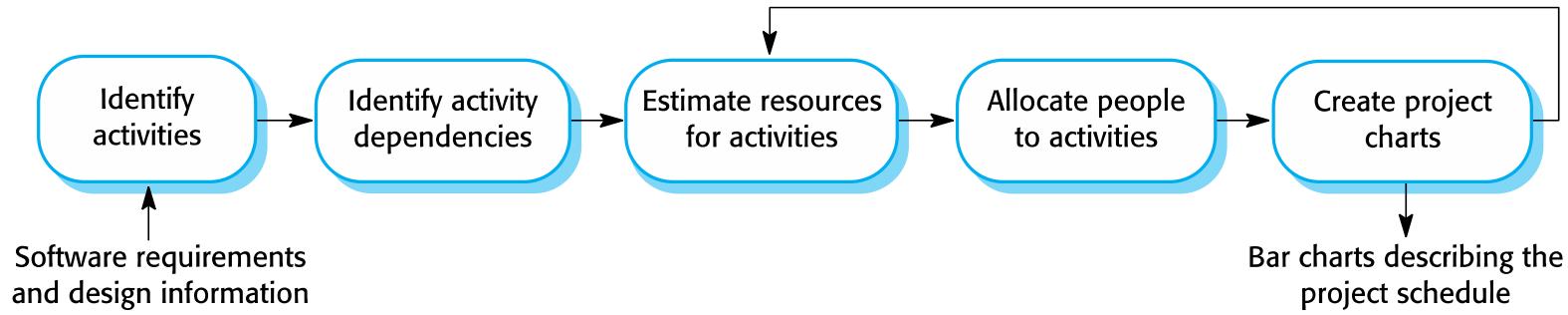
- ✧ Split project into tasks and estimate time and resources required to complete each task.
- ✧ Organize tasks concurrently to make optimal use of workforce.
- ✧ Minimize task dependencies to avoid delays caused by one task waiting for another to complete.
- ✧ Dependent on project managers intuition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

The project scheduling process



Software Engineering

Ian Sommerville

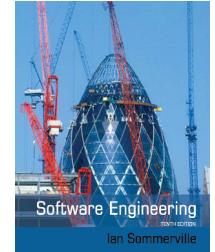


Scheduling problems



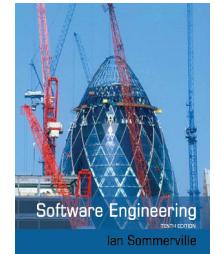
- ✧ Estimating the difficulty of problems and hence the cost of developing a solution is hard.
- ✧ Productivity is not proportional to the number of people working on a task.
- ✧ Adding people to a late project makes it later because of communication overheads.
- ✧ The unexpected always happens. Always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

Schedule presentation



- ✧ 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.
- ✧ Calendar-based
 - Bar charts are the most commonly used representation for project schedules. They show the schedule as activities or resources against time.
- ✧ Activity networks
 - Show task dependencies

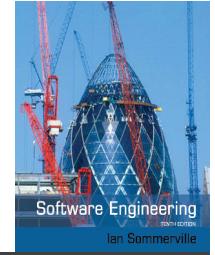
Project activities



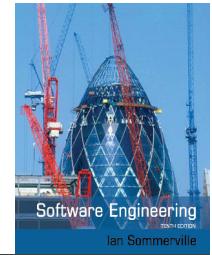
❖ Project activities (tasks) are the basic planning element.
Each activity has:

- a duration in calendar days or months,
- an effort estimate, which shows the number of person-days or person-months to complete the work,
- a deadline by which the activity should be complete,
- a defined end-point, which might be a document, the holding of a review meeting, the successful execution of all tests, etc.

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.

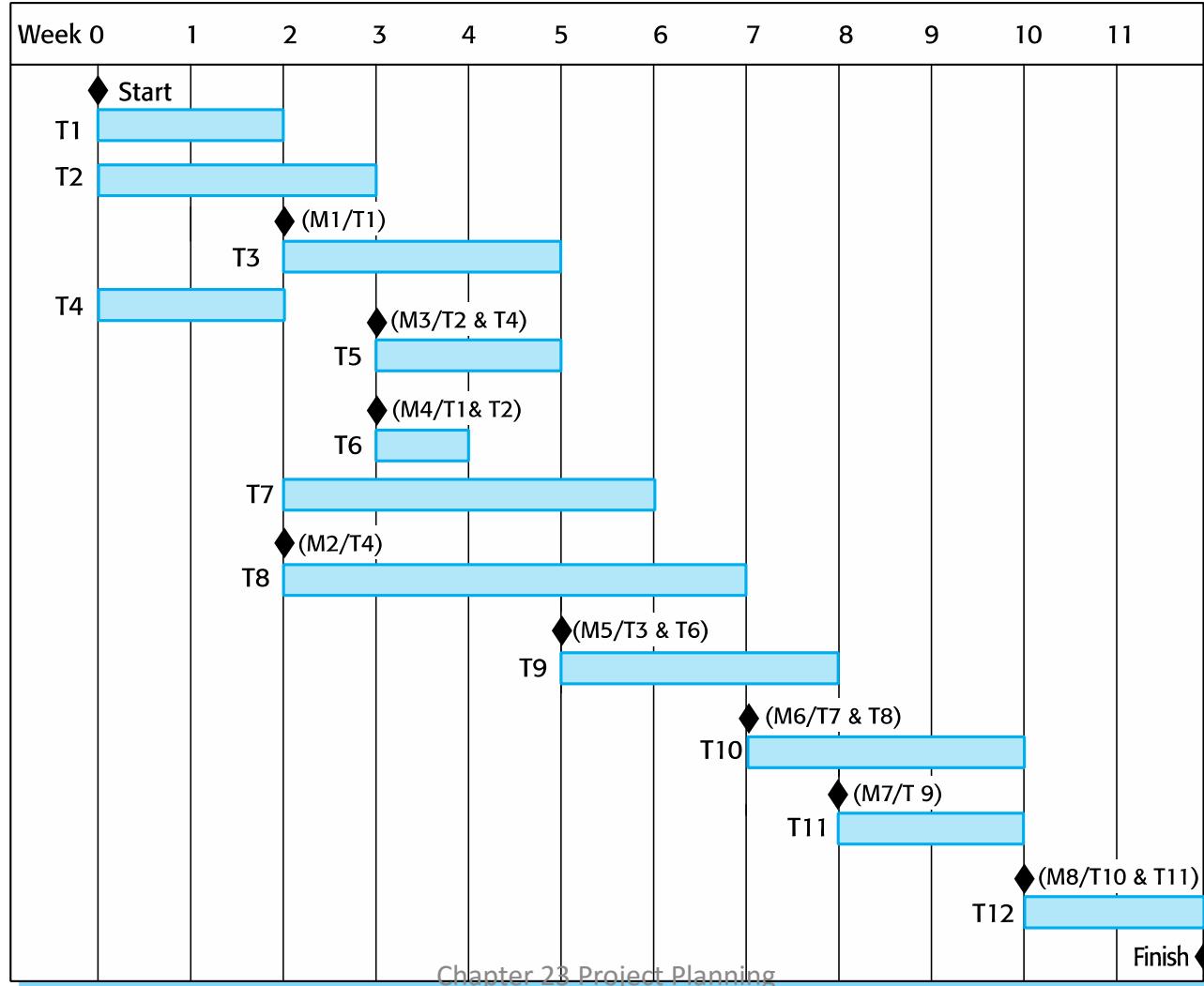
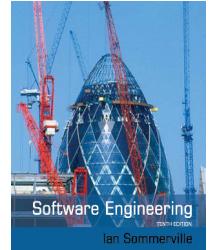


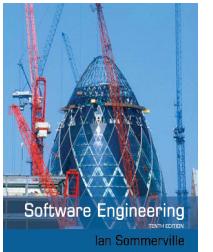
Tasks, durations, and dependencies

Software Engineering
Ian Sommerville

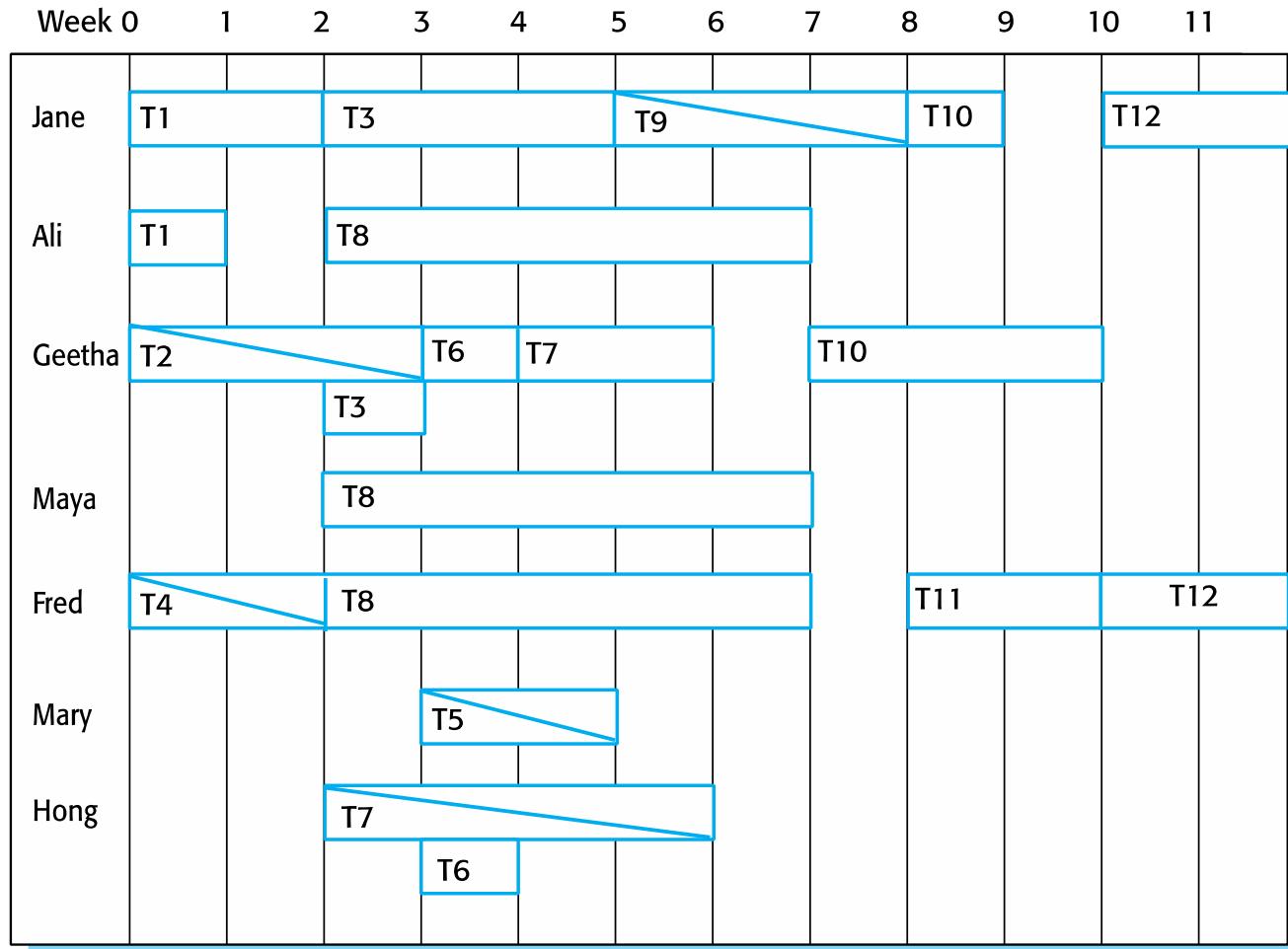
Task	Effort (person-days)	Duration (days)	Dependencies
T1	15	10	
T2	8	15	
T3	20	15	T1 (M1)
T4	5	10	
T5	5	10	T2, T4 (M3)
T6	10	5	T1, T2 (M4)
T7	25	20	T1 (M1)
T8	75	25	T4 (M2)
T9	10	15	T3, T6 (M5)
T10	20	15	T7, T8 (M6)
T11	10	10	T9 (M7)
T12	20	10	T10, T11 (M8)

Activity bar chart





Staff allocation chart





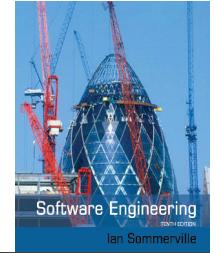
Agile planning

Agile planning



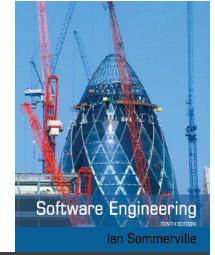
- ✧ Agile methods of software development are iterative approaches where the software is developed and delivered to customers in increments.
- ✧ Unlike plan-driven approaches, the functionality of these increments is not planned in advance but is decided during the development.
 - The decision on what 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



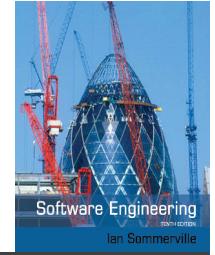
- ✧ Release planning, which looks ahead for several months and decides on the features that should be included in a release of a system.
- ✧ Iteration planning, which has a shorter term outlook, and focuses on planning the next increment of a system. This is typically 2-4 weeks of work for the team.

Approaches to agile planning



- ✧ Planning in Scrum
 - Covered in Chapter 3
- ✧ Based on managing a project backlog (things to be done) with daily reviews of progress and problems
- ✧ The planning game
 - Developed originally as part of Extreme Programming (XP)
 - Dependent on user stories as a measure of progress in the project

Story-based planning



- ✧ The planning game 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.
- ✧ Stories are assigned 'effort points' reflecting their size and difficulty of implementation
- ✧ The number of effort points implemented per day is measured giving an estimate of the team's 'velocity'
- ✧ This allows the total effort required to implement the system to be estimated

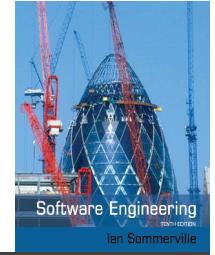
The planning game



Software Engineering
Ian Sommerville

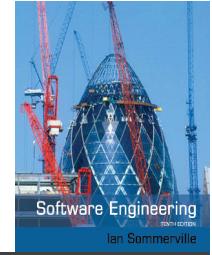


Release and iteration planning



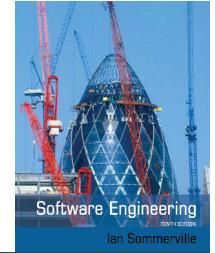
- ✧ Release planning involves selecting and refining the stories that will reflect the features to be implemented in a release of a system and the order in which the stories should be implemented.
- ✧ Stories to be implemented in each iteration are chosen, with the number of stories reflecting the time to deliver an iteration (usually 2 or 3 weeks).
- ✧ The team's velocity is used to guide the choice of stories so that they can be delivered within an iteration.

Task allocation



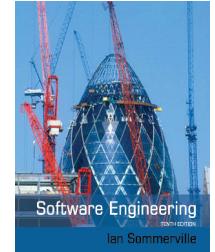
- ✧ During the task planning stage, the developers break down stories into development tasks.
 - A development task should take 4–16 hours.
 - All of the tasks that must be completed to implement all of the stories in that iteration are listed.
 - The individual developers then sign up for the specific tasks that they will implement.
- ✧ Benefits of this approach:
 - The whole team gets an overview of the tasks to be completed in an iteration.
 - Developers have a sense of ownership in these tasks and this is likely to motivate them to complete the task.

Software delivery



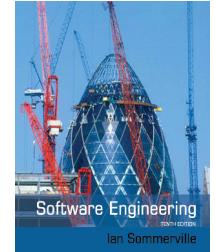
- ✧ A software increment is always delivered at the end of each project iteration.
- ✧ If the features to be included in the increment cannot be completed in the time allowed, the scope of the work is reduced.
- ✧ The delivery schedule is never extended.

Agile planning difficulties



- ✧ Agile planning is reliant on customer involvement and availability.
- ✧ This can be difficult to arrange, as customer representatives sometimes have to prioritize other work and are not available for the planning game.
- ✧ Furthermore, some customers may be more familiar with traditional project plans and may find it difficult to engage in an agile planning process.

Agile planning applicability



- ✧ Agile planning works well with small, stable development teams that can get together and discuss the stories to be implemented.
- ✧ However, where teams are large and/or geographically distributed, or when team membership changes frequently, it is practically impossible for everyone to be involved in the collaborative planning that is essential for agile project management.



Estimation techniques

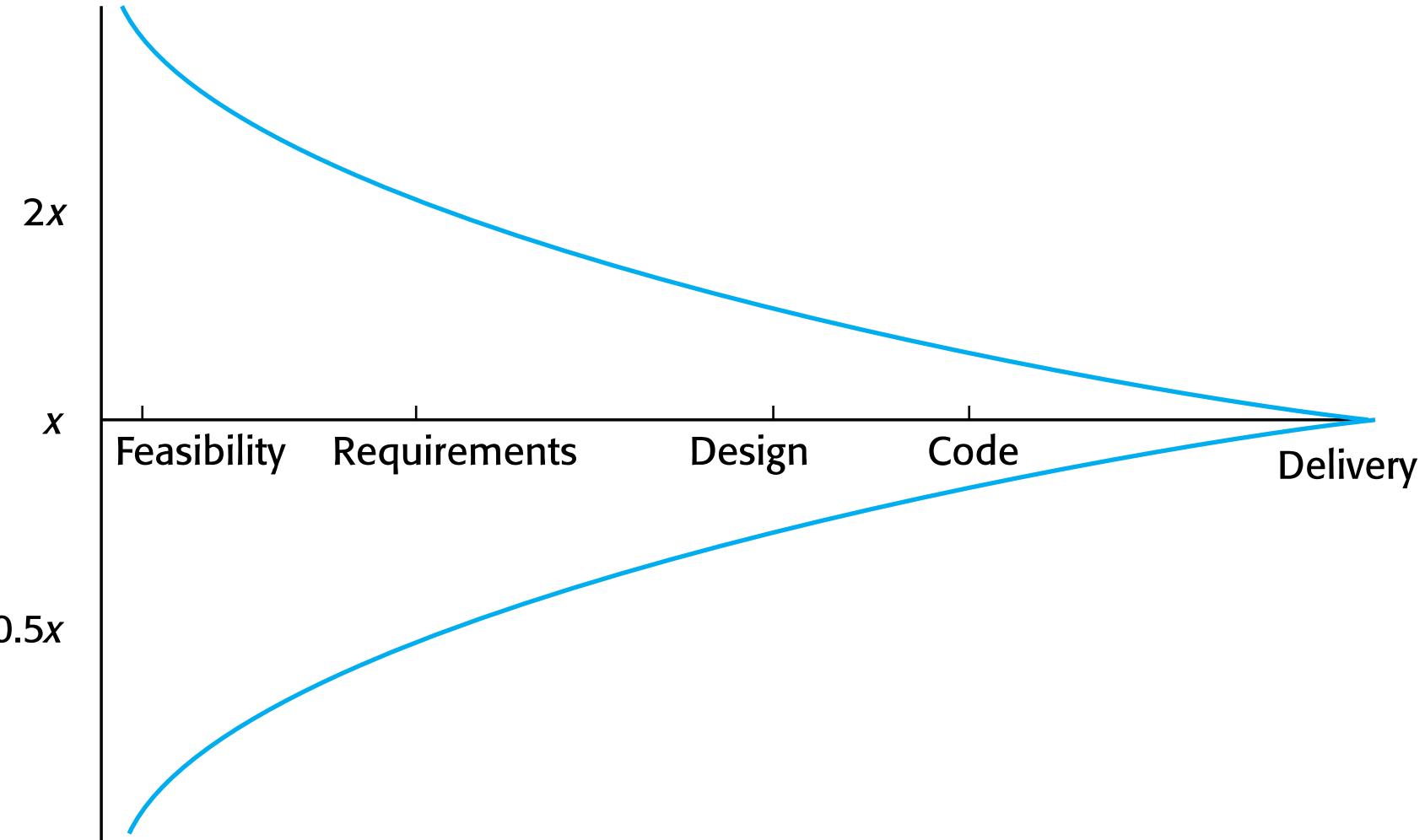
Estimation techniques

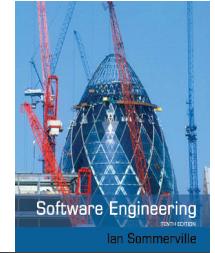


- ✧ Organizations need to make software effort and cost estimates. There are two types of technique that can be used to do this:
 - *Experience-based techniques* The estimate of future effort requirements is based on the manager's experience of past projects and the application domain. Essentially, the 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.



Estimate uncertainty

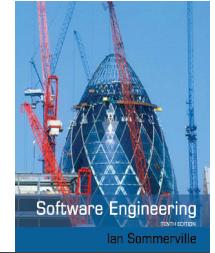




Experience-based approaches

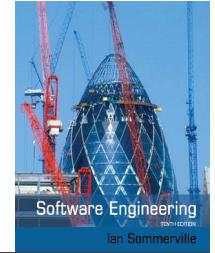
- ✧ Experience-based techniques rely on judgments based on experience of past projects and the effort expended in these projects on software development activities.
- ✧ Typically, you identify the deliverables to be produced in a project and the different software components or systems that are to be 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.

Problem with experience-based approaches

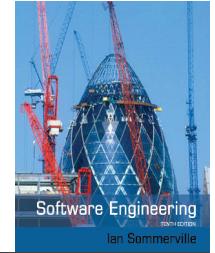


- ✧ The difficulty with experience-based techniques is that a new software project may not have much in common with previous projects.
- ✧ Software development changes very quickly and a project will often use unfamiliar techniques such as web services, application system configuration or HTML5.
- ✧ If you have not worked with these techniques, your previous experience may not help you to estimate the effort required, making it more difficult to produce accurate costs and schedule estimates.

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 reused systems 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.

Effectiveness of algorithmic models



- ✧ Algorithmic cost models are a systematic way to estimate the effort required to develop a system. However, these models are complex and difficult to use.
- ✧ There are many attributes and considerable scope for uncertainty in estimating their values.
- ✧ This complexity means that the practical application of algorithmic cost modeling has been limited to a relatively small number of large companies, mostly working in defense and aerospace systems engineering.



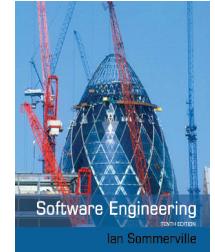
COCOMO cost modeling

COCOMO cost modeling



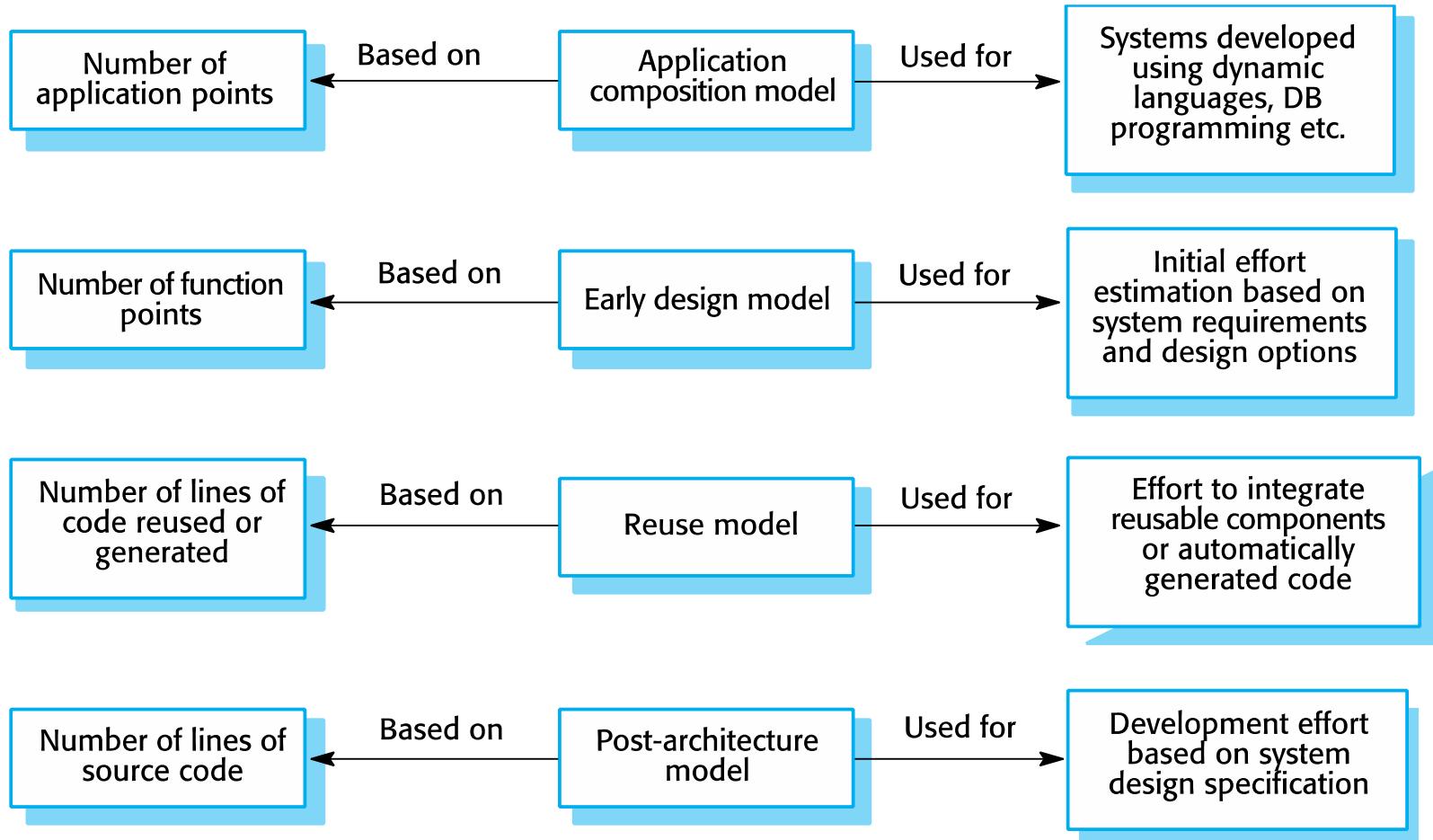
- ✧ An empirical model based on project experience.
- ✧ Well-documented, ‘independent’ model which is not tied to a specific software vendor.
- ✧ Long history from initial version published in 1981 (COCOMO-81) through various instantiations to COCOMO 2.
- ✧ COCOMO 2 takes into account different approaches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

COCOMO 2 models

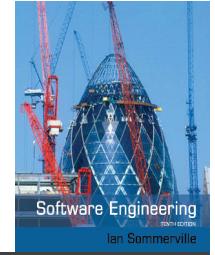


- ✧ COCOMO 2 incorporates a range of sub-models that produce increasingly detailed software estimates.
- ✧ The sub-models in COCOMO 2 are:
 - **Application composition model.** Used when software is composed from existing parts.
 - **Early design model.** Used when requirements are available but design 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

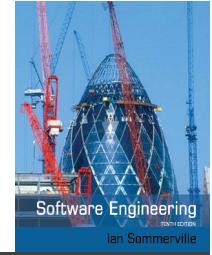


Application composition model

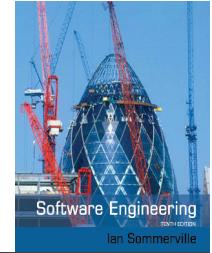


- ✧ Supports prototyping projects and projects where there is extensive reuse.
- ✧ Based on standard estimates of developer productivity in application (object) points/month.
- ✧ Takes software tool use into account.
- ✧ Formula is
 - $PM = (NAP \cdot (1 - \%reuse/100)) / PROD$
 - PM is the effort in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



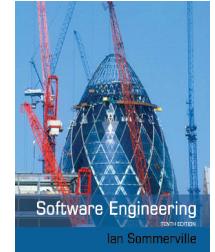
Software Engineering

Ian Sommerville

Early design model

- ✧ Estimates can be made after the requirements have been agreed.
- ✧ Based on a standard formula for algorithmic models
- ✧ $PM = A \cdot \text{Size}^B \cdot M$ 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



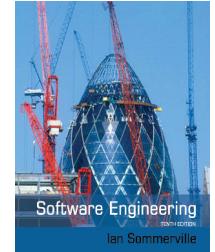
- ✧ Multipliers reflect the capability of the developers, the non-functional requirements, the familiarity with the development platform, etc.
 - RCPX - product reliability and complexity;
 - RUSE - the reuse required;
 - PDIF - platform difficulty;
 - PREX - personnel experience;
 - PERS - personnel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



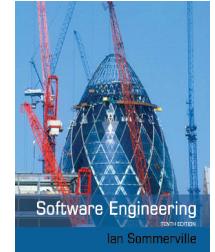
- ✧ For generated code:
- ✧ $PM = (ASLOC * AT/100)/ATPROD$
 - ASLOC is the number of lines of generated code
 - AT is the percentage of code automatically generated.
 - ATPROD is the productivity of engineers in integrating this code.

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



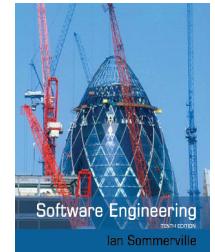
- ✧ This depends on 5 scale factors (see next slide). Their sum/100 is added to 1.01
- ✧ A company takes on a project in a new domain. The client has not defined the process to be used and has not allowed time for risk analysis. The company has a CMM level 2 rating.
 - Precedenteness - new project (4)
 - 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
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.
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.
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.
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.
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.

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



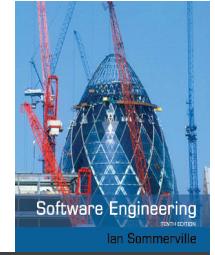
Exponent value	1.17
Reliability	Very low, multiplier = 0.75
Complexity	Very low, multiplier = 0.75
Memory constraint	None, multiplier = 1
Tool use	Very high, multiplier = 0.72
Schedule	Normal, multiplier = 1
Adjusted COCOMO estimate	295 person-months

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



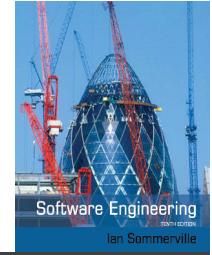
- ✧ Staff required can't be computed by diving the development time by the required schedule.
- ✧ The number of people working on a project varies depending on the phase of the project.
- ✧ The more people who work on the project, the more total effort is usually required.
- ✧ A very rapid build-up of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

Key points



- ✧ The price charged for a system does not just depend on its estimated development costs and the profit required by the development company. Organizational factors may mean that the price is increased to compensate for increased risk or decreased to gain competitive advantage.
- ✧ Software is often priced to gain a contract and the functionality of the system is then adjusted to meet the estimated price.
- ✧ 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.

Key points



- ✧ Project scheduling involves the creation of various graphical representations of part of the project plan. Bar charts, which show the activity duration and staffing timelines, are the most commonly used schedule representations.
- ✧ A project milestone is a predictable outcome of an activity or set of activities. At each milestone, a formal report of progress should be presented to management. A deliverable is a work product that is delivered to the project customer.
- ✧ The agile planning game involves the whole team in project planning. The plan is developed incrementally and, if problems arise, it is adjusted so that software functionality is reduced instead of delaying the delivery of an increment.

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 a mature algorithmic cost model that takes project, product, hardware and personnel attributes into account when formulating a cost estimate.

Chapter 24 - Quality Management

10/12/2014 Chapter 24 Quality management 1

Topics covered

- ✧ Software quality
- ✧ Software standards
- ✧ Reviews and inspections
- ✧ Quality management and agile development
- ✧ Software measurement

10/12/2014 Chapter 24 Quality management 2

Software quality management

✧ Concerned with ensuring that the required level of quality is achieved in a software product.

✧ Three principal concerns:

- At the organizational level, quality management is concerned with establishing a framework of organizational processes and standards that will lead to high-quality software.
- At the project level, quality management involves the application of specific quality processes and checking that these planned processes have been followed.
- At the project level, quality management is also concerned with establishing a quality plan for a project. The quality plan should set out the quality goals for the project and define what processes and standards are to be used.

10/12/2014 Chapter 24 Quality management 3

Quality management activities

✧ Quality management provides an independent check on the software development process.

✧ The quality management process checks the project deliverables to ensure that they are consistent with organizational standards and goals

✧ The quality team should be independent from the development team so that they can take an objective view of the software. This allows them to report on software quality without being influenced by software development issues.

10/12/2014 Chapter 24 Quality management 4

Quality management and software development

Software development process: D1, D2, D3, D4, D5

Quality management process: Standards and procedures, Quality plan, Quality review reports

10/12/2014 Chapter 24 Quality management 5

Quality planning

✧ A quality plan sets out the desired product qualities and how these are assessed and defines the most significant quality attributes.

✧ The quality plan should define the quality assessment process.

✧ It should set out which organisational standards should be applied and, where necessary, define new standards to be used.

10/12/2014 Chapter 24 Quality management 6

Quality plans

- ✧ Quality plan structure
 - Product introduction;
 - Product plans;
 - Process descriptions;
 - Quality goals;
 - Risks and risk management.
- ✧ Quality plans should be short, succinct documents
 - If they are too long, no-one will read them.

10/12/2014

Chapter 24 Quality management

7

Scope of quality management

- ✧ Quality management is particularly important for large, complex systems. The quality documentation is a record of progress and supports continuity of development as the development team changes.
- ✧ For smaller systems, quality management needs less documentation and should focus on establishing a quality culture.
- ✧ Techniques have to evolve when agile development is used.

10/12/2014

Chapter 24 Quality management

8

Software quality

10/12/2014

Chapter 24 Quality management

9

Software quality

- ✧ Quality, simplistically, means that a product should meet its specification.
- ✧ This is problematical for software systems
 - There is a tension between customer quality requirements (efficiency, reliability, etc.) and developer quality requirements (maintainability, reusability, etc.);
 - Some quality requirements are difficult to specify in an unambiguous way;
 - Software specifications are usually incomplete and often inconsistent.
- ✧ The focus may be 'fitness for purpose' rather than specification conformance.

10/12/2014

Chapter 24 Quality management

10

Software fitness for purpose

- ✧ Has the software been properly tested?
- ✧ Is the software sufficiently dependable to be put into use?
- ✧ Is the performance of the software acceptable for normal use?
- ✧ Is the software usable?
- ✧ Is the software well-structured and understandable?
- ✧ Have programming and documentation standards been followed in the development process?

10/12/2014

Chapter 24 Quality management

11

Non-functional characteristics

- ✧ The subjective quality of a software system is largely based on its non-functional characteristics.
- ✧ This reflects practical user experience – if the software's functionality is not what is expected, then users will often just work around this and find other ways to do what they want to do.
- ✧ However, if the software is unreliable or too slow, then it is practically impossible for them to achieve their goals.

10/12/2014

Chapter 24 Quality management

12

Software quality attributes

Safety	Understandability	Portability
Security	Testability	Usability
Reliability	Adaptability	Reusability
Resilience	Modularity	Efficiency
Robustness	Complexity	Learnability

10/12/2014

Chapter 24 Quality management

13

Quality conflicts

- ✧ It is not possible for any system to be optimized for all of these attributes – for example, improving robustness may lead to loss of performance.
- ✧ The quality plan should therefore define the most important quality attributes for the software that is being developed.
- ✧ The plan should also include a definition of the quality assessment process, an agreed way of assessing whether some quality, such as maintainability or robustness, is present in the product.

10/12/2014

Chapter 24 Quality management

14

Process and product quality

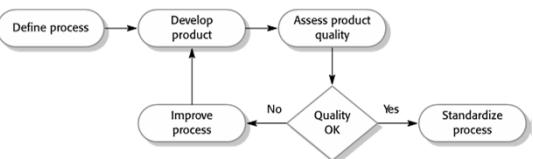
- ✧ The quality of a developed product is influenced by the quality of the production process.
- ✧ This is important in software development as some product quality attributes are hard to assess.
- ✧ However, there is a very complex and poorly understood relationship between software processes and product quality.
 - The application of individual skills and experience is particularly important in software development;
 - External factors such as the novelty of an application or the need for an accelerated development schedule may impair product quality.

10/12/2014

Chapter 24 Quality management

15

Process-based quality



10/12/2014

Chapter 24 Quality management

16

Quality culture

- ✧ Quality managers should aim to develop a 'quality culture' where everyone responsible for software development is committed to achieving a high level of product quality.
- ✧ They should encourage teams to take responsibility for the quality of their work and to develop new approaches to quality improvement.
- ✧ They should support people who are interested in the intangible aspects of quality and encourage professional behavior in all team members.

10/12/2014

Chapter 24 Quality management

17

Software standards

10/12/2014

Chapter 24 Quality management

18

Software standards

- ✧ Standards define the required attributes of a product or process. They play an important role in quality management.
- ✧ Standards may be international, national, organizational or project standards.

10/12/2014

Chapter 24 Quality management

19

Importance of standards

- ✧ Encapsulation of best practice- avoids repetition of past mistakes.
- ✧ They are a framework for defining what quality means in a particular setting i.e. that organization's view of quality.
- ✧ They provide continuity - new staff can understand the organisation by understanding the standards that are used.

10/12/2014

Chapter 24 Quality management

20

Product and process standards

Product standards

- Apply to the software product being developed. They include document standards, such as the structure of requirements documents, documentation standards, such as a standard comment header for an object class definition, and coding standards, which define how a programming language should be used.

Process standards

- These define the processes that should be followed during software development. Process standards may include definitions of specification, design and validation processes, process support tools and a description of the documents that should be written during these processes.

10/12/2014

Chapter 24 Quality management

21

Product and process standards

Product standards	Process standards
Design review form	Design review conduct
Requirements document structure	Submission of new code for system building
Method header format	Version release process
Java programming style	Project plan approval process
Project plan format	Change control process
Change request form	Test recording process

10/12/2014

Chapter 24 Quality management

22

Problems with standards

- ✧ They may not be seen as relevant and up-to-date by software engineers.
- ✧ They often involve too much bureaucratic form filling.
- ✧ If they are unsupported by software tools, tedious form filling work is often involved to maintain the documentation associated with the standards.

10/12/2014

Chapter 24 Quality management

23

Standards development

- ✧ Involve practitioners in development. Engineers should understand the rationale underlying a standard.
- ✧ Review standards and their usage regularly. Standards can quickly become outdated and this reduces their credibility amongst practitioners.
- ✧ Detailed standards should have specialized tool support. Excessive clerical work is the most significant complaint against standards.
 - Web-based forms are not good enough.

10/12/2014

Chapter 24 Quality management

24

ISO 9001 standards framework

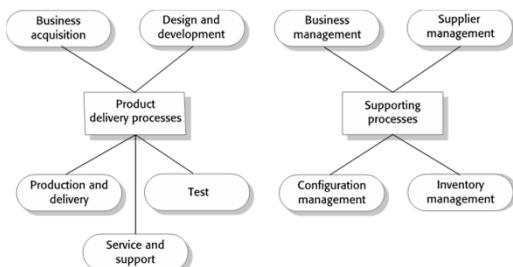
- ✧ An international set of standards that can be used as a basis for developing quality management systems.
- ✧ ISO 9001, the most general of these standards, applies to organizations that design, develop and maintain products, including software.
- ✧ The ISO 9001 standard is a framework for developing software standards.
 - It sets out general quality principles, describes quality processes in general and lays out the organizational standards and procedures that should be defined. These should be documented in an organizational quality manual.

10/12/2014

Chapter 24 Quality management

25

ISO 9001 core processes

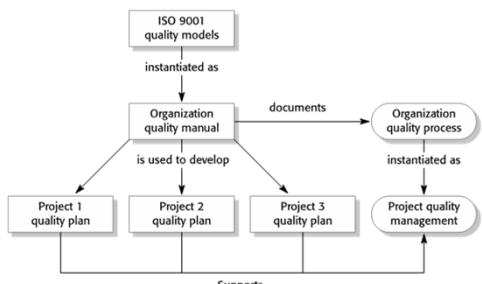


10/12/2014

Chapter 24 Quality management

26

ISO 9001 and quality management



10/12/2014

Chapter 24 Quality management

27

ISO 9001 certification

- ✧ Quality standards and procedures should be documented in an organisational quality manual.
- ✧ An external body may certify that an organisation's quality manual conforms to ISO 9000 standards.
- ✧ Some customers require suppliers to be ISO 9000 certified although the need for flexibility here is increasingly recognised.

10/12/2014

Chapter 24 Quality management

28

Software quality and ISO9001

- ✧ The ISO 9001 certification is inadequate because it defines quality to be the conformance to standards.
- ✧ It takes no account of quality as experienced by users of the software. For example, a company could define test coverage standards specifying that all methods in objects must be called at least once.
- ✧ Unfortunately, this standard can be met by incomplete software testing that does not include tests with different method parameters. So long as the defined testing procedures are followed and test records maintained, the company could be ISO 9001 certified.

10/12/2014

Chapter 24 Quality management

29

Reviews and inspections

10/12/2014

Chapter 24 Quality management

30

Reviews and inspections

- ❖ A group examines part or all of a process or system and its documentation to find potential problems.
- ❖ Software or documents may be 'signed off' at a review which signifies that progress to the next development stage has been approved by management.
- ❖ There are different types of review with different objectives
 - Inspections for defect removal (product);
 - Reviews for progress assessment (product and process);
 - Quality reviews (product and standards).

10/12/2014

Chapter 24 Quality management

31

Quality reviews

- ❖ A group of people carefully examine part or all of a software system and its associated documentation.
- ❖ Code, designs, specifications, test plans, standards, etc. can all be reviewed.
- ❖ Software or documents may be 'signed off' at a review which signifies that progress to the next development stage has been approved by management.

10/12/2014

Chapter 24 Quality management

32

Phases in the review process

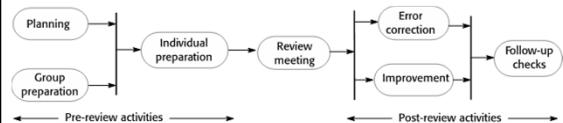
- ❖ Pre-review activities
 - Pre-review activities are concerned with review planning and review preparation
- ❖ The review meeting
 - During the review meeting, an author of the document or program being reviewed should 'walk through' the document with the review team.
- ❖ Post-review activities
 - These address the problems and issues that have been raised during the review meeting.

10/12/2014

Chapter 24 Quality management

33

The software review process



10/12/2014

Chapter 24 Quality management

34

Distributed reviews

- ❖ The processes suggested for reviews assume that the review team has a face-to-face meeting to discuss the software or documents that they are reviewing.
- ❖ However, project teams are now often distributed, sometimes across countries or continents, so it is impractical for team members to meet face to face.
- ❖ Remote reviewing can be supported using shared documents where each review team member can annotate the document with their comments.

10/12/2014

Chapter 24 Quality management

35

Program inspections

- ❖ These are peer reviews where engineers examine the source of a system with the aim of discovering anomalies and defects.
- ❖ Inspections do 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.

10/12/2014

Chapter 24 Quality management

36

Inspection checklists

- ◊ Checklist of common errors should be used to drive the inspection.
- ◊ Error checklists are programming language dependent and reflect the characteristic errors that are likely to arise in the language.
- ◊ In general, the 'weaker' the type checking, the larger the checklist.
- ◊ Examples: Initialisation, Constant naming, loop termination, array bounds, etc.

10/12/2014

Chapter 24 Quality management

37

An inspection checklist (a)

Fault class	Inspection check
Data faults	<ul style="list-style-type: none"> • Are all program variables initialized before their values are used? • Have all constants been named? • Should the upper bound of arrays be equal to the size of the array or Size-1? • If character strings are used, is a delimiter explicitly assigned? • Is there any possibility of buffer overflow?
Control faults	<ul style="list-style-type: none"> • For each conditional statement, is the condition correct? • Is each loop certain to terminate? • Are compound statements correctly bracketed? • In case statements, are all possible cases accounted for? • If a break is required after each case in case statements, has it been included?
Input/output faults	<ul style="list-style-type: none"> • Are all input variables used? • Are all output variables assigned a value before they are output? • Can unexpected inputs cause corruption?

10/12/2014

Chapter 24 Quality management

38

An inspection checklist (b)

Fault class	Inspection check
Interface faults	<ul style="list-style-type: none"> • Do all function and method calls have the correct number of parameters? • Do formal and actual parameter types match? • Are the parameters in the right order? • If components access shared memory, do they have the same model of the shared memory structure?
Storage management faults	<ul style="list-style-type: none"> • If a linked structure is modified, have all links been correctly reassigned? • If dynamic storage is used, has space been allocated correctly? • Is space explicitly deallocated after it is no longer required?
Exception management faults	<ul style="list-style-type: none"> • Have all possible error conditions been taken into account?

10/12/2014

Chapter 24 Quality management

39

Quality management and agile development

Quality management and agile development

- ◊ Quality management in agile development is informal rather than document-based.
- ◊ It relies on establishing a quality culture, where all team members feel responsible for software quality and take actions to ensure that quality is maintained.
- ◊ The agile community is fundamentally opposed to what it sees as the bureaucratic overheads of standards-based approaches and quality processes as embodied in ISO 9001.

10/12/2014

Chapter 24 Quality management

41

Shared good practice

- ◊ **Check before check-in**
 - Programmers are responsible for organizing their own code reviews with other team members before the code is checked in to the build system.
- ◊ **Never break the build**
 - Team members should not check in code that causes the system to fail. Developers have to test their code changes against the whole system and be confident that these work as expected.
- ◊ **Fix problems when you see them**
 - If a programmer discovers problems or obscurities in code developed by someone else, they can fix these directly rather than referring them back to the original developer.

10/12/2014

Chapter 24 Quality management

42

Reviews and agile methods



- ◊ The review process in agile software development is usually informal.
- ◊ In Scrum, there is a review meeting after each iteration of the software has been completed (a sprint review), where quality issues and problems may be discussed.
- ◊ In Extreme Programming, pair programming ensures that code is constantly being examined and reviewed by another team member.

10/12/2014

Chapter 24 Quality management

43

Pair programming



- ◊ This is an approach where 2 people are responsible for code development and work together to achieve this.
- ◊ Code developed by an individual is therefore constantly being examined and reviewed by another team member.
- ◊ Pair programming leads to a deep knowledge of a program, as both programmers have to understand the program in detail to continue development.
- ◊ This depth of knowledge is difficult to achieve in inspection processes and pair programming can find bugs that would not be discovered in formal inspections.

10/12/2014

Chapter 24 Quality management

44

Pair programming weaknesses



- ◊ *Mutual misunderstandings*
 - Both members of a pair may make the same mistake in understanding the system requirements. Discussions may reinforce these errors.
- ◊ *Pair reputation*
 - Pairs may be reluctant to look for errors because they do not want to slow down the progress of the project.
- ◊ *Working relationships*
 - The pair's ability to discover defects is likely to be compromised by their close working relationship that often leads to reluctance to criticize work partners.

10/12/2014

Chapter 24 Quality management

45

Agile QM and large systems



- ◊ When a large system is being developed for an external customer, agile approaches to quality management with minimal documentation may be impractical.
 - If the customer is a large company, it may have its own quality management processes and may expect the software development company to report on progress in a way that is compatible with them.
 - Where there are several geographically distributed teams involved in development, perhaps from different companies, then informal communications may be impractical.
 - For long-lifetime systems, the team involved in development will changeWithout documentation, new team members may find it impossible to understand development.

10/12/2014

Chapter 24 Quality management

46

Software measurement



10/12/2014

Chapter 24 Quality management

47

Software measurement



- ◊ Software measurement is concerned with deriving a numeric value for an attribute of a software product or process.
- ◊ This allows for objective comparisons between techniques and processes.
- ◊ Although some companies have introduced measurement programmes, most organisations still don't make systematic use of software measurement.
- ◊ There are few established standards in this area.

10/12/2014

Chapter 24 Quality management

48

Software metric

- ❖ Any type of measurement which relates to a software system, process or related documentation
 - Lines of code in a program, the Fog index, number of person-days required to develop a component.
- ❖ Allow the software and the software process to be quantified.
- ❖ May be used to predict product attributes or to control the software process.
- ❖ Product metrics can be used for general predictions or to identify anomalous components.

10/12/2014

Chapter 24 Quality management

49

Types of process metric

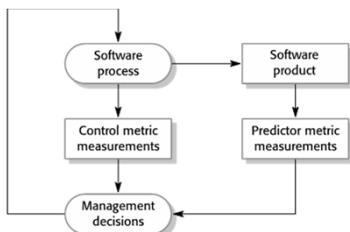
- ❖ *The time taken for a particular process to be completed*
 - This can be the total time devoted to the process, calendar time, the time spent on the process by particular engineers, and so on.
- ❖ *The resources required for a particular process*
 - Resources might include total effort in person-days, travel costs or computer resources.
- ❖ *The number of occurrences of a particular event*
 - Examples of events that might be monitored include the number of defects discovered during code inspection, the number of requirements changes requested, the number of bug reports in a delivered system and the average number of lines of code modified in response to a requirements change.

10/12/2014

Chapter 24 Quality management

50

Predictor and control measurements



10/12/2014

Chapter 24 Quality management

51

Use of measurements

- ❖ To assign a value to system quality attributes
 - By measuring the characteristics of system components, such as their cyclomatic complexity, and then aggregating these measurements, you can assess system quality attributes, such as maintainability.
- ❖ To identify the system components whose quality is sub-standard
 - Measurements can identify individual components with characteristics that deviate from the norm. For example, you can measure components to discover those with the highest complexity. These are most likely to contain bugs because the complexity makes them harder to understand.

10/12/2014

Chapter 24 Quality management

52

Metrics assumptions

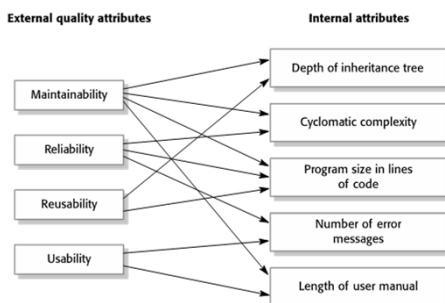
- ❖ A software property can be measured accurately.
- ❖ The relationship exists between what we can measure and what we want to know. We can only measure internal attributes but are often more interested in external software attributes.
- ❖ This relationship has been formalised and validated.
- ❖ It may be difficult to relate what can be measured to desirable external quality attributes.

10/12/2014

Chapter 24 Quality management

53

Relationships between internal and external software



10/12/2014

Chapter 24 Quality management

54

Problems with measurement in industry



- ✧ It is impossible to quantify the return on investment of introducing an organizational metrics program.
- ✧ There are no standards for software metrics or standardized processes for measurement and analysis.
- ✧ In many companies, software processes are not standardized and are poorly defined and controlled.
- ✧ Most work on software measurement has focused on code-based metrics and plan-driven development processes. However, more and more software is now developed by configuring ERP systems or COTS.
- ✧ Introducing measurement adds additional overhead to processes.

10/12/2014

Chapter 24 Quality management

55

Empirical software engineering



- ✧ Software measurement and metrics are the basis of empirical software engineering.
- ✧ This is a research area in which experiments on software systems and the collection of data about real projects has been used to form and validate hypotheses about software engineering methods and techniques.
- ✧ Research on empirical software engineering, this has not had a significant impact on software engineering practice.
- ✧ It is difficult to relate generic research to a project that is different from the research study.

10/12/2014

Chapter 24 Quality management

56

Product metrics



- ✧ A quality metric should be a predictor of product quality.
- ✧ Classes of product metric
 - Dynamic metrics which are collected by measurements made of a program in execution;
 - Static metrics which are collected by measurements made of the system representations;
 - Dynamic metrics help assess efficiency and reliability
 - Static metrics help assess complexity, understandability and maintainability.

10/12/2014

Chapter 24 Quality management

57

Dynamic and static metrics



- ✧ Dynamic metrics are closely related to software quality attributes
 - It is relatively easy to measure the response time of a system (performance attribute) or the number of failures (reliability attribute).
- ✧ Static metrics have an indirect relationship with quality attributes
 - You need to try and derive a relationship between these metrics and properties such as complexity, understandability and maintainability.

10/12/2014

Chapter 24 Quality management

58

Static software product metrics



Software metric	Description
Fan-in/Fan-out	Fan-in is a measure of the number of functions or methods that call another function or method (say X). Fan-out is the number of functions that are called by function X. A high value for fan-in means that X is tightly coupled to the rest of the design and changes to X will have extensive knock-on effects. A high value for fan-out suggests that the overall complexity of X may be high because of the complexity of the control logic needed to coordinate the called components.
Length of code	This is a measure of the size of a program. Generally, the larger the size of the code of a component, the more complex and error-prone that component is likely to be. Length of code has been shown to be one of the most reliable metrics for predicting error-proneness in components.

10/12/2014

Chapter 24 Quality management

59

Static software product metrics



Software metric	Description
Cyclomatic complexity	This is a measure of the control complexity of a program. This control complexity may be related to program understandability. I discuss cyclomatic complexity in Chapter 8.
Length of identifiers	This is a measure of the average length of identifiers (names for variables, classes, methods, etc.) in a program. The longer the identifiers, the more likely they are to be meaningful and hence the more understandable the program.
Depth of conditional nesting	This is a measure of the depth of nesting of if-statements in a program. Deeply nested if-statements are hard to understand and potentially error-prone.
Fog index	This is a measure of the average length of words and sentences in documents. The higher the value of a document's Fog index, the more difficult the document is to understand.

10/12/2014

Chapter 24 Quality management

60

The CK object-oriented metrics suite

Object-oriented metric	Description
Weighted methods per class (WMC)	This is the number of methods in each class, weighted by the complexity of each method. Therefore, a simple method may have a complexity of 1, and a large and complex method a much higher value. The larger the value for this metric, the more complex the object class. Complex objects are more likely to be difficult to understand. They may not be logically cohesive, so cannot be reused effectively as superclasses in an inheritance tree.
Depth of inheritance tree (DIT)	This represents the number of discrete levels in the inheritance tree where subclasses inherit attributes and operations (methods) from superclasses. The deeper the inheritance tree, the more complex the design. Many object classes may have to be understood to understand the object classes at the leaves of the tree.
Number of children (NOC)	This is a measure of the number of immediate subclasses in a class. It measures the breadth of a class hierarchy, whereas DIT measures its depth. A high value for NOC may indicate greater reuse. It may mean that more effort should be made in validating base classes because of the number of subclasses that depend on them.

10/12/2014 Chapter 24 Quality management 61

The CK object-oriented metrics suite

Object-oriented metric	Description
Coupling between object classes (CBO)	Classes are coupled when methods in one class use methods or instance variables defined in a different class. CBO is a measure of how much coupling exists. A high value for CBO means that classes are highly dependent, and therefore it is more likely that changing one class will affect other classes in the program.
Response for a class (RFC)	RFC is a measure of the number of methods that could potentially be executed in response to a message received by an object of that class. Again, RFC is related to complexity. The higher the value for RFC, the more complex a class and hence the more likely it is that it will include errors.
Lack of cohesion in methods (LCOM)	LCOM is calculated by considering pairs of methods in a class. LCOM is the difference between the number of method pairs without shared attributes and the number of method pairs with shared attributes. The value of this metric has been widely debated and it exists in several variations. It is not clear if it really adds any additional, useful information over and above that provided by other metrics.

10/12/2014 Chapter 24 Quality management 62

Software component analysis

- ❖ System component can be analyzed separately using a range of metrics.
- ❖ The values of these metrics may then compared for different components and, perhaps, with historical measurement data collected on previous projects.
- ❖ Anomalous measurements, which deviate significantly from the norm, may imply that there are problems with the quality of these components.

10/12/2014 Chapter 24 Quality management 63

The process of product measurement

```

graph TD
    A([Choose measurements to be made]) --> B([Select components to be assessed])
    B --> C([Measure component characteristics])
    C --> D([Identify anomalous measurements])
    D --> E([Analyze anomalous components])
    E -- feedback loop --> B
  
```

10/12/2014 Chapter 24 Quality management 64

Measurement ambiguity

- ❖ When you collect quantitative data about software and software processes, you have to analyze that data to understand its meaning.
- ❖ It is easy to misinterpret data and to make inferences that are incorrect.
- ❖ You cannot simply look at the data on its own. You must also consider the context where the data is collected.

10/12/2014 Chapter 24 Quality management 65

Measurement surprises

- ❖ Reducing the number of faults in a program leads to an increased number of help desk calls
 - The program is now thought of as more reliable and so has a wider more diverse market. The percentage of users who call the help desk may have decreased but the total may increase;
 - A more reliable system is used in a different way from a system where users work around the faults. This leads to more help desk calls.

10/12/2014 Chapter 24 Quality management 66

Software context

- ✧ Processes and products that are being measured are not insulated from their environment.
- ✧ The business environment is constantly changing and it is impossible to avoid changes to work practice just because they may make comparisons of data invalid.
- ✧ Data about human activities cannot always be taken at face value. The reasons why a measured value changes are often ambiguous. These reasons must be investigated in detail before drawing conclusions from any measurements that have been made.

10/12/2014

Chapter 24 Quality management

67

Software analytics

- ✧ *Software analytics is analytics on software data for managers and software engineers with the aim of empowering software development individuals and teams to gain and share insight from their data to make better decisions.*

10/12/2014

Chapter 24 Quality management

68

Software analytics enablers

- ✧ The automated collection of user data by software product companies when their product is used.
 - If the software fails, information about the failure and the state of the system can be sent over the Internet from the user's computer to servers run by the product developer.
- ✧ The use of open source software available on platforms such as Sourceforge and GitHub and open source repositories of software engineering data.
 - The source code of open source software is available for automated analysis and this can sometimes be linked with data in the open source repository.

10/12/2014

Chapter 24 Quality management

69

Analytics tool use

- ✧ Tools should be easy to use as managers are unlikely to have experience with analysis.
- ✧ • Tools should run quickly and produce concise outputs rather than large volumes of information.
- ✧ • Tools should make many measurements using as many parameters as possible. It is impossible to predict in advance what insights might emerge.
- ✧ • Tools should be interactive and allow managers and developers to explore the analyses.

10/12/2014

Chapter 24 Quality management

70

Status of software analytics

- ✧ Software analytics is still immature and it is too early to say what effect it will have.
- ✧ Not only are there general problems of 'big data' processing, our knowledge depends on collected data from large companies.
 - This is primarily from software products and it is unclear if the tools and techniques that are appropriate for products can also be used with custom software.
- ✧ Small companies are unlikely to invest in the data collection systems that are required for automated analysis so may not be able to use software analytics.

10/12/2014

Chapter 24 Quality management

71

Key points

- ✧ Software quality management is concerned with ensuring that software has a low number of defects and that it reaches the required standards of maintainability, reliability, portability etc. Software standards are important for quality assurance as they represent an identification of 'best practice'. When developing software, standards provide a solid foundation for building good quality software.
- ✧ Reviews of the software process deliverables involve a team of people who check that quality standards are being followed. Reviews are the most widely used technique for assessing quality.

10/12/2014

Chapter 24 Quality management

72

Key points



- ✧ In a program inspection or peer review, a small team systematically checks the code. They read the code in detail and look for possible errors and omissions. The problems detected are discussed at a code review meeting.
- ✧ Agile quality management relies on establishing a quality culture where the development team works together to improve software quality.
- ✧ Software measurement can be used to gather quantitative data about software and the software process.

10/12/2014

Chapter 24 Quality management

73

Key points



- ✧ You may be able to use the values of the software metrics that are collected to make inferences about product and process quality.
- ✧ Product quality metrics are particularly useful for highlighting anomalous components that may have quality problems. These components should then be analyzed in more detail.
- ✧ Software analytics is the automated analysis of large volumes of software product and process data to discover relationships that may provide insights for project managers and developers.

10/12/2014

Chapter 24 Quality management

74



Software Engineering

Ian Sommerville

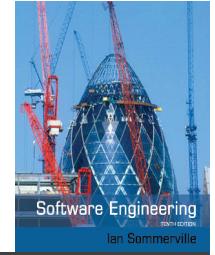
Chapter 25 – Configuration Management

Topics covered



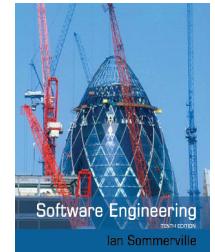
- ✧ Version management
- ✧ System building
- ✧ Change management
- ✧ Release management

Configuration management



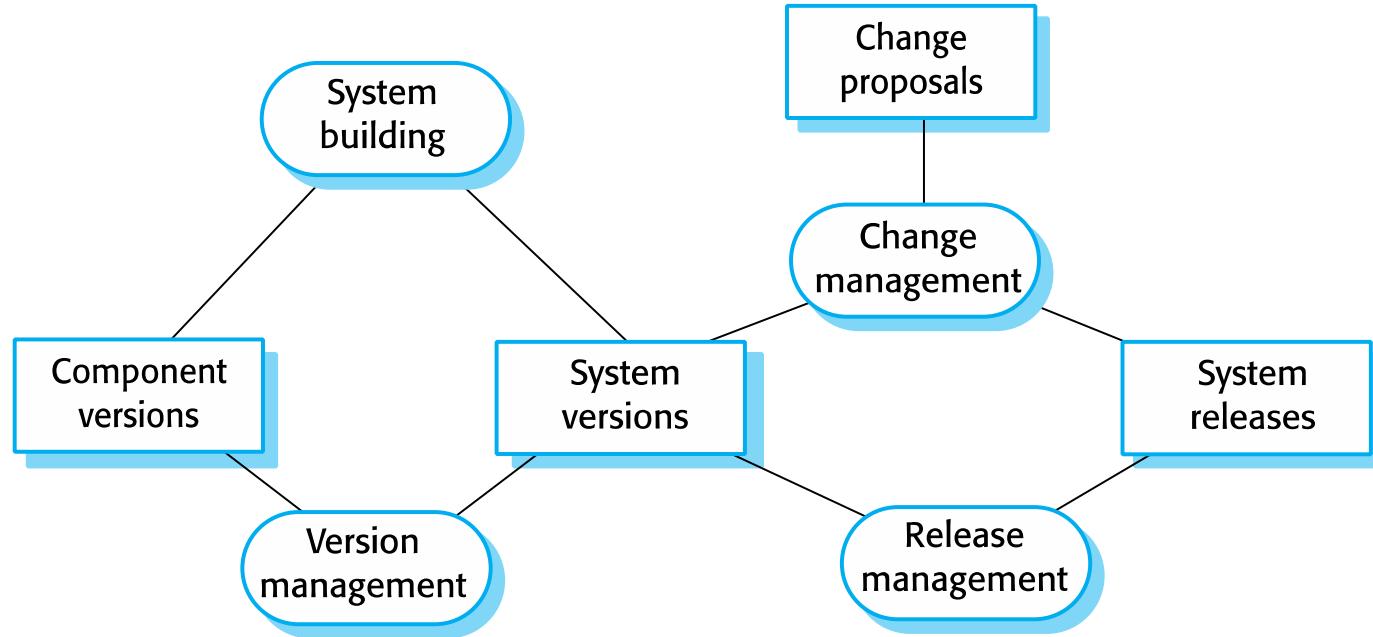
- ✧ Software systems are constantly changing during development and use.
- ✧ Configuration management (CM) is concerned with the policies, processes and tools for managing changing software systems.
- ✧ You need CM because it is easy to lose track of what changes and component versions have been incorporated into each system version.
- ✧ CM is essential for team projects to control changes made by different developers

CM activities

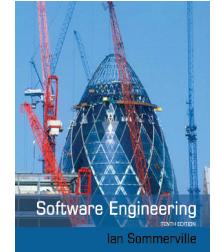


- ✧ Version management
 - Keeping track of the multiple versions of system components and ensuring that changes made to components by different developers do not interfere with each other.
- ✧ System building
 - The process of assembling program components, data and libraries, then compiling these to create an executable system.
- ✧ Change management
 - Keeping track of requests for changes to the software from customers and developers, working out the costs and impact of changes, and deciding the changes should be implemented.
- ✧ Release management
 - Preparing software for external release and keeping track of the system versions that have been released for customer use.

Configuration management activities



Agile development and CM



- ✧ Agile development, where components and systems are changed several times per day, is impossible without using CM tools.
- ✧ The definitive versions of components are held in a shared project repository and developers copy these into their own workspace.
- ✧ They make changes to the code then use system building tools to create a new system on their own computer for testing. Once they are happy with the changes made, they return the modified components to the project repository.



Development phases

- ✧ A development phase where the development team is responsible for managing the software configuration and new functionality is being added to the software.
- ✧ A system testing phase where a version of the system is released internally for testing.
 - No new system functionality is added. Changes made are bug fixes, performance improvements and security vulnerability repairs.
- ✧ A release phase where the software is released to customers for use.
 - New versions of the released system are developed to repair bugs and vulnerabilities and to include new features.

Multi-version systems



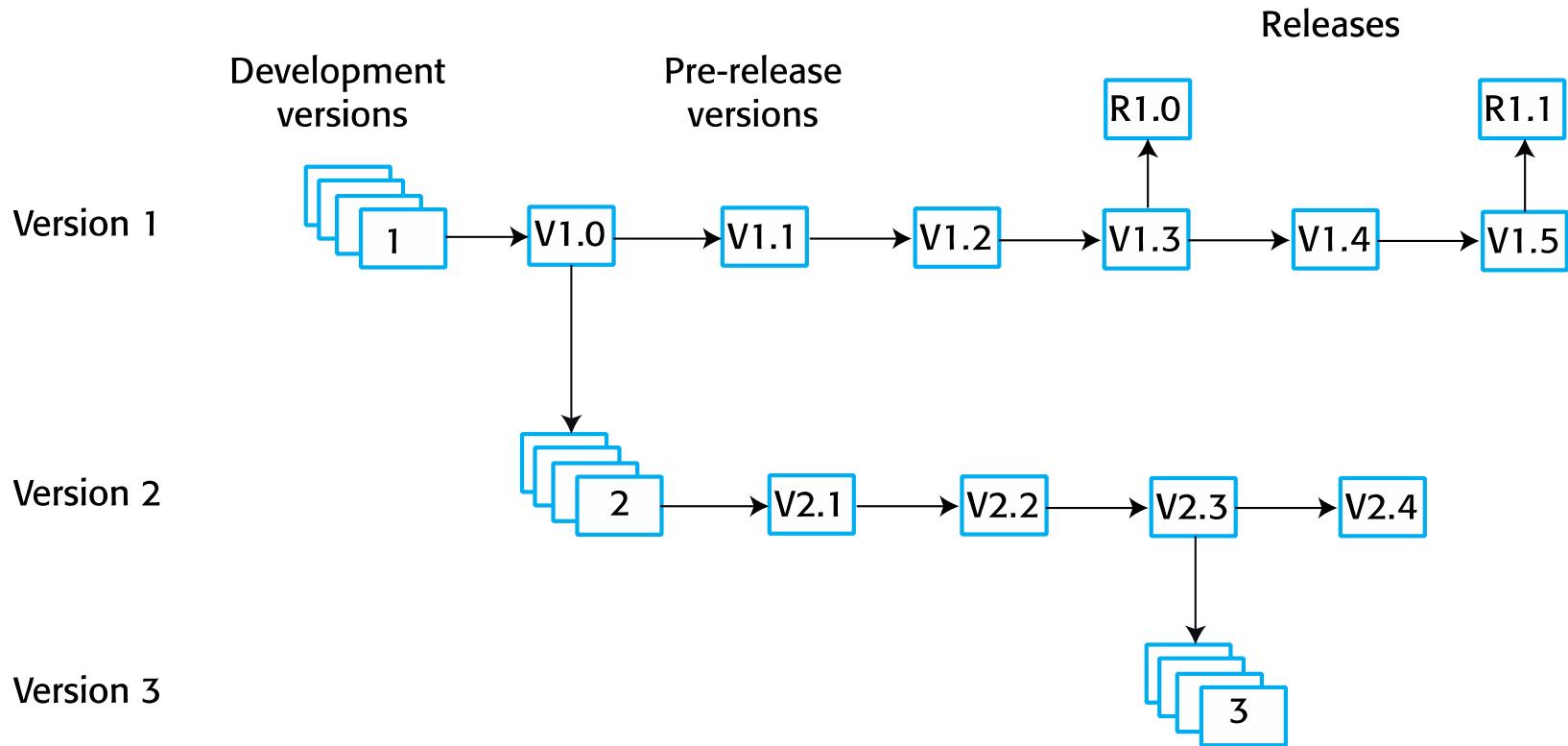
- ✧ For large systems, there is never just one ‘working’ version of a system.
- ✧ There are always several versions of the system at different stages of development.
- ✧ There may be several teams involved in the development of different system versions.

Multi-version system development

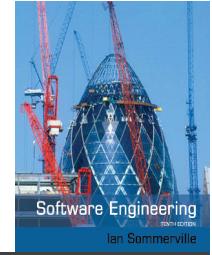


Software Engineering

Ian Sommerville



CM terminology



Term	Explanation
Baseline	A baseline is a collection of component versions that make up a system. Baselines are controlled, which means that the versions of the components making up the system cannot be changed. This means that it is always possible to recreate a baseline from its constituent components.
Branching	The creation of a new codeline from a version in an existing codeline. The new codeline and the existing codeline may then develop independently.
Codeline	A codeline is a set of versions of a software component and other configuration items on which that component depends.
Configuration (version) control	The process of ensuring that versions of systems and components are recorded and maintained so that changes are managed and all versions of components are identified and stored for the lifetime of the system.
Configuration item or software configuration item (SCI)	Anything associated with a software project (design, code, test data, document, etc.) that has been placed under configuration control. There are often different versions of a configuration item. Configuration items have a unique name.
Mainline	A sequence of baselines representing different versions of a system.

CM terminology

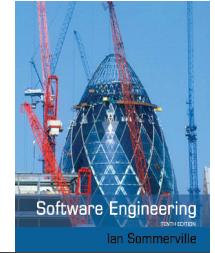


Term	Explanation
Merging	The creation of a new version of a software component by merging separate versions in different codelines. These codelines may have been created by a previous branch of one of the codelines involved.
Release	A version of a system that has been released to customers (or other users in an organization) for use.
Repository	A shared database of versions of software components and meta-information about changes to these components.
System building	The creation of an executable system version by compiling and linking the appropriate versions of the components and libraries making up the system.
Version	An instance of a configuration item that differs, in some way, from other instances of that item. Versions always have a unique identifier.
Workspace	A private work area where software can be modified without affecting other developers who may be using or modifying that software.



Version management

Version management



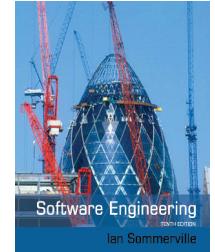
- ✧ Version management (VM) is the process of keeping track of different versions of software components or configuration items and the systems in which these components are used.
- ✧ It also involves ensuring that changes made by different developers to these versions do not interfere with each other.
- ✧ Therefore version management can be thought of as the process of managing codelines and baselines.

Codelines and baselines



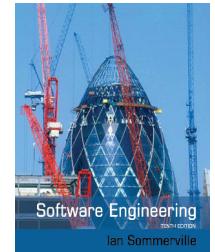
- ✧ A codeline is a sequence of versions of source code with later versions in the sequence derived from earlier versions.
- ✧ Codelines normally apply to components of systems so that there are different versions of each component.
- ✧ A baseline is a definition of a specific system.
- ✧ The baseline therefore specifies the component versions that are included in the system plus a specification of the libraries used, configuration files, etc.

Baselines



- ✧ Baselines may be specified using a configuration language, which allows you to define what components are included in a version of a particular system.
- ✧ Baselines are important because you often have to recreate a specific version of a complete system.
 - For example, a product line may be instantiated so that there are individual system versions for different customers. You may have to recreate the version delivered to a specific customer if, for example, that customer reports bugs in their system that have to be repaired.

Codelines and baselines



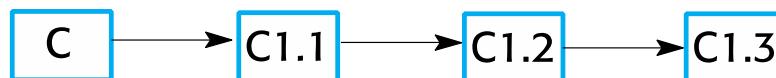
Codeline (A)



Codeline (B)



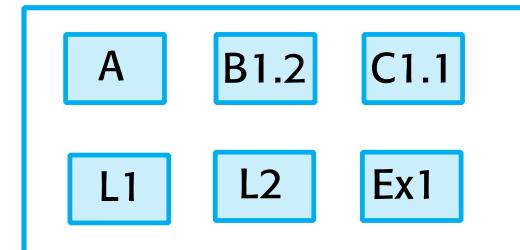
Codeline (C)



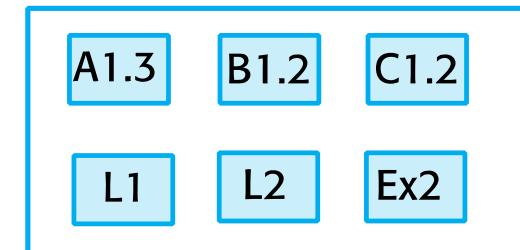
Libraries and external components



Baseline - V1



Baseline - V2



Mainline

Version control systems



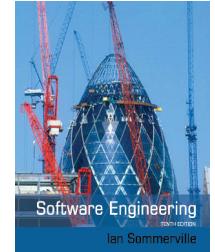
- ✧ Version control (VC) systems identify, store and control access to the different versions of components. There are two types of modern version control system
 - Centralized systems, where there is a single master repository that maintains all versions of the software components that are being developed. Subversion is a widely used example of a centralized VC system.
 - Distributed systems, where multiple versions of the component repository exist at the same time. Git is a widely-used example of a distributed VC system.

Key features of version control systems



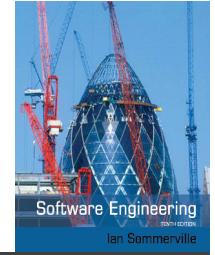
- ✧ Version and release identification
- ✧ Change history recording
- ✧ Support for independent development
- ✧ Project support
- ✧ Storage management

Public repository and private workspaces



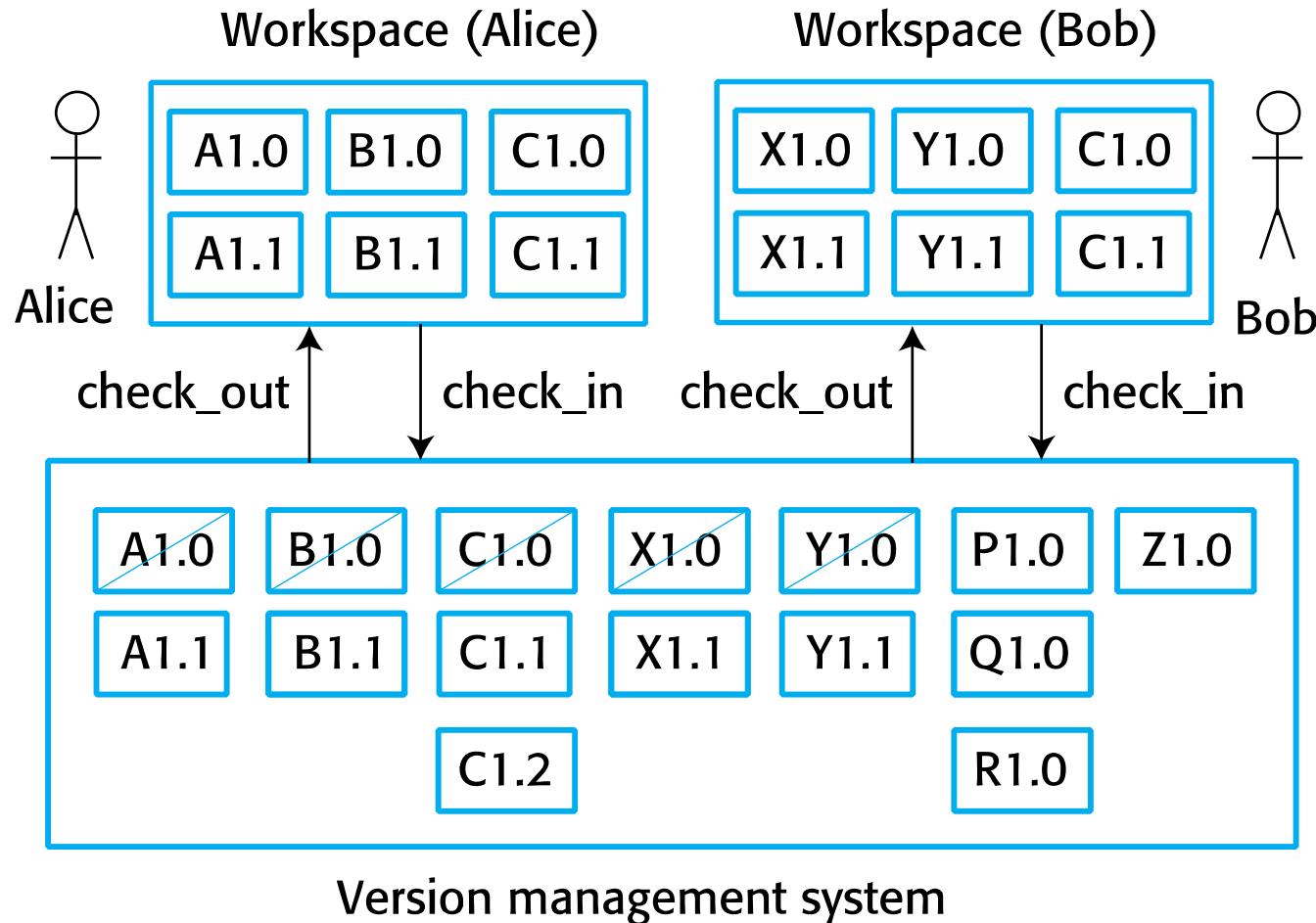
- ✧ To support independent development without interference, version control systems use the concept of a project repository and a private workspace.
- ✧ The project repository maintains the ‘master’ version of all components. It is used to create baselines for system building.
- ✧ When modifying components, developers copy (check-out) these from the repository into their workspace and work on these copies.
- ✧ When they have finished their changes, the changed components are returned (checked-in) to the repository.

Centralized version control



- ✧ Developers check out components or directories of components from the project repository into their private workspace and work on these copies in their private workspace.
- ✧ When their changes are complete, they check-in the components back to the repository.
- ✧ If several people are working on a component at the same time, each check it out from the repository. If a component has been checked out, the VC system warns other users wanting to check out that component that it has been checked out by someone else.

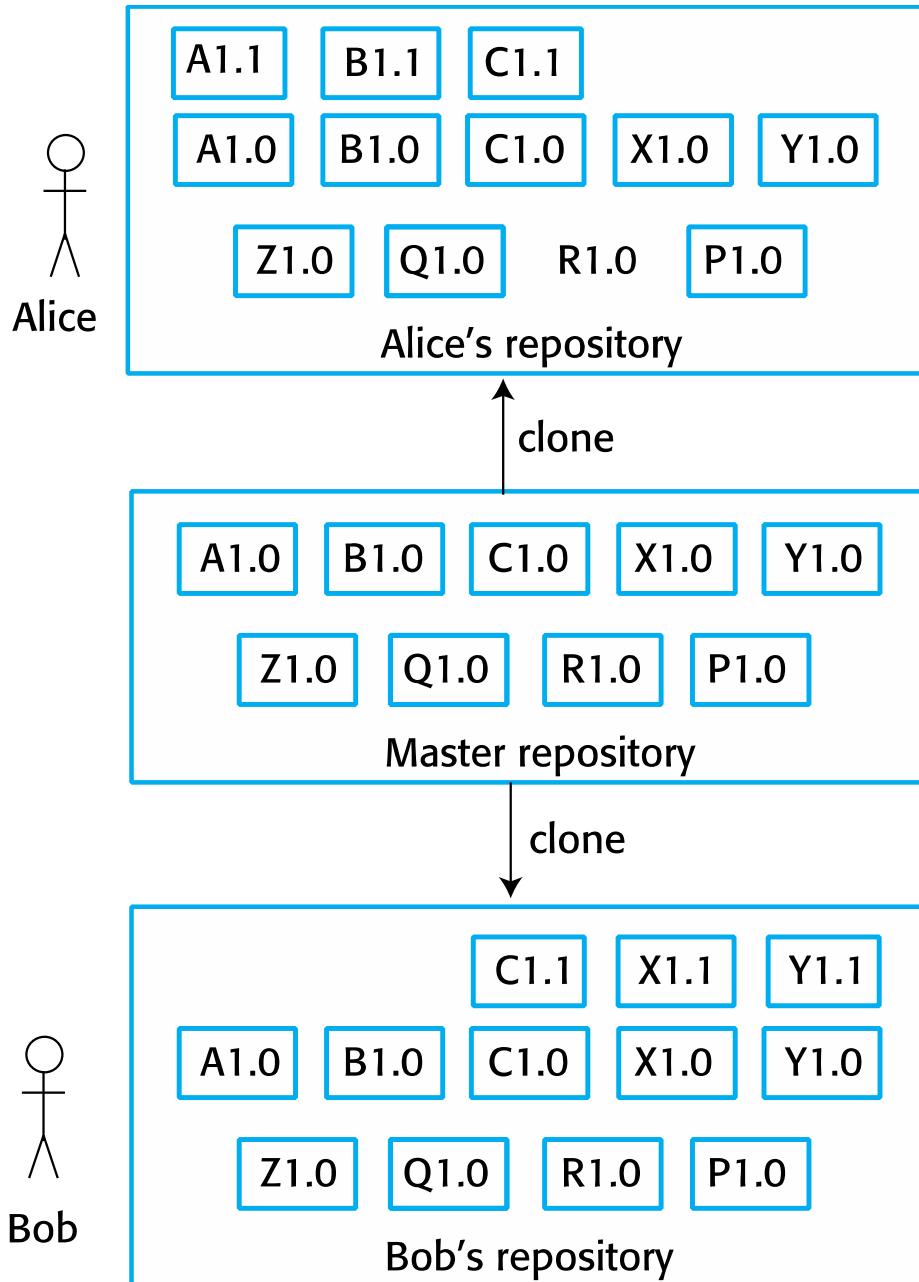
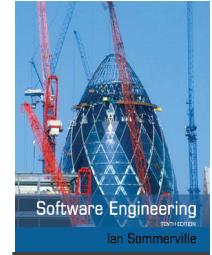
Repository Check-in/Check-out



Distributed version control



- ✧ A ‘master’ repository is created on a server that maintains the code produced by the development team.
- ✧ Instead of checking out the files that they need, a developer creates a clone of the project repository that is downloaded and installed on their computer.
- ✧ Developers work on the files required and maintain the new versions on their private repository on their own computer.
- ✧ When changes are done, they ‘commit’ these changes and update their private server repository. They may then ‘push’ these changes to the project repository.



Repository cloning

Benefits of distributed version control



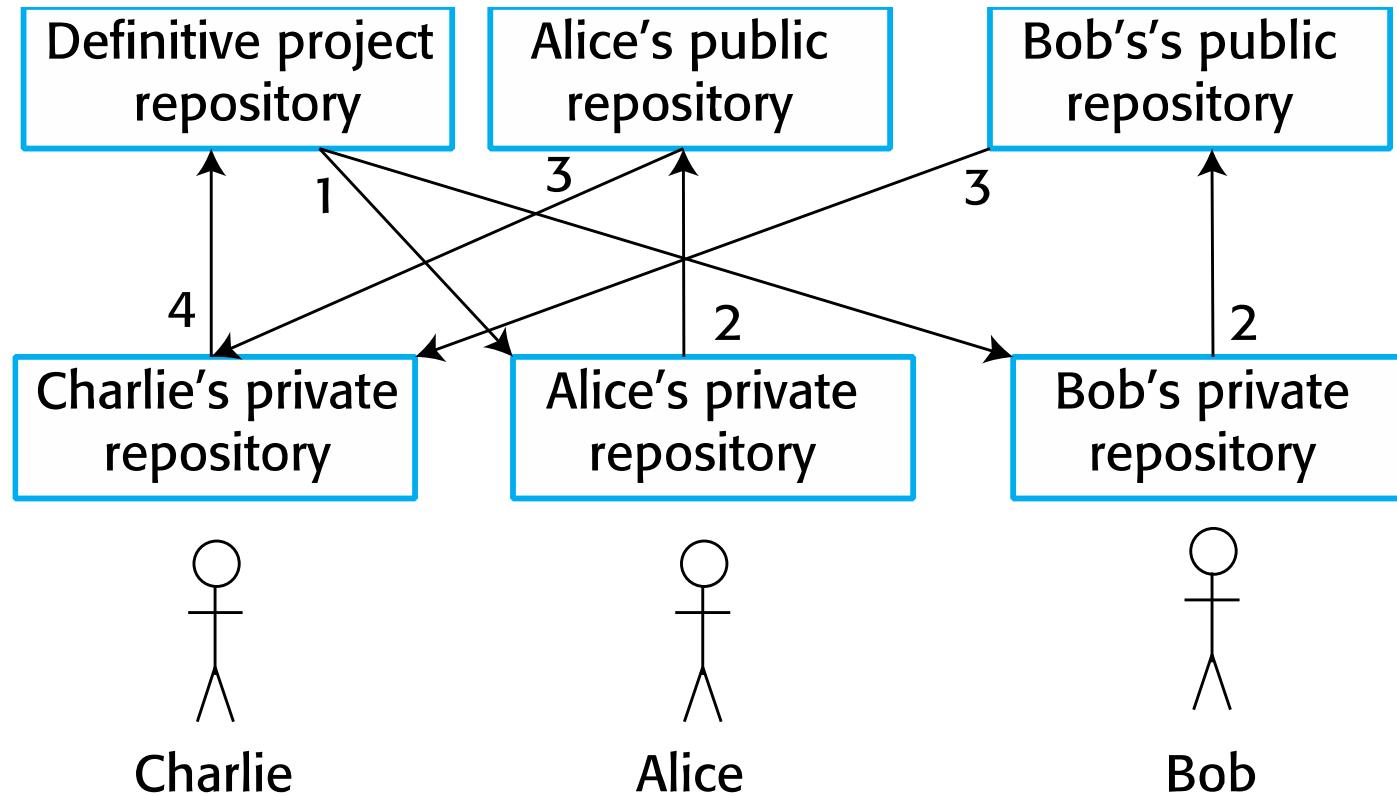
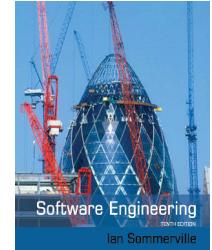
- ✧ It provides a backup mechanism for the repository.
 - If the repository is corrupted, work can continue and the project repository can be restored from local copies.
- ✧ It allows for off-line working so that developers can commit changes if they do not have a network connection.
- ✧ Project support is the default way of working.
 - Developers can compile and test the entire system on their local machines and test the changes that they have made.

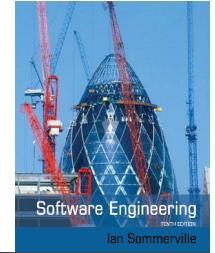
Open source development



- ✧ Distributed version control is essential for open source development.
 - Several people may be working simultaneously on the same system without any central coordination.
- ✧ As well as a private repository on their own computer, developers also maintain a public server repository to which they push new versions of components that they have changed.
 - It is then up to the open-source system ‘manager’ to decide when to pull these changes into the definitive system.

Open-source development

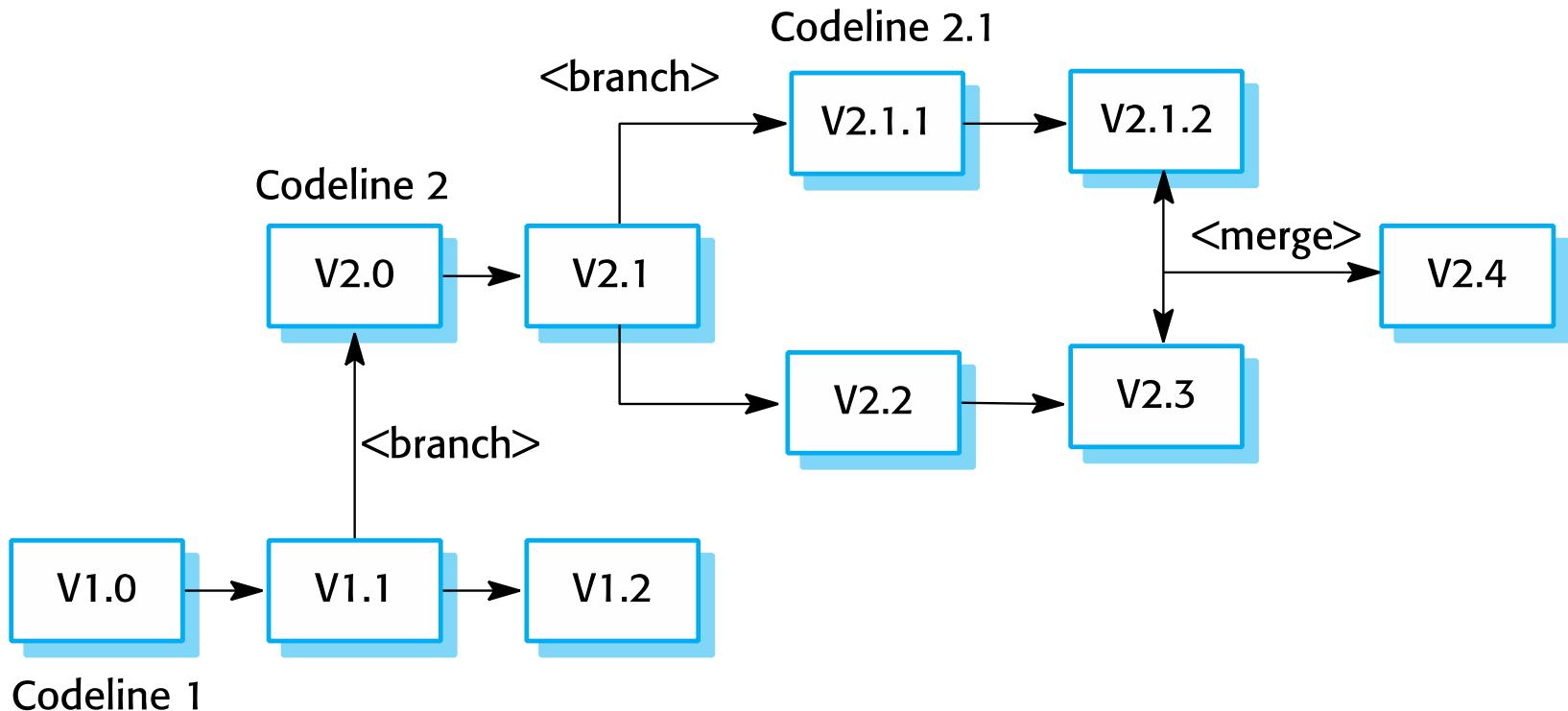




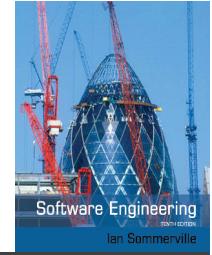
Branching and merging

- ✧ Rather than a linear sequence of versions that reflect changes to the component over time, there may be several independent sequences.
 - This is normal in system development, where different developers work independently on different versions of the source code and so change it in different ways.
- ✧ At some stage, it may be necessary to merge codeline branches to create a new version of a component that includes all changes that have been made.
 - If the changes made involve different parts of the code, the component versions may be merged automatically by combining the deltas that apply to the code.

Branching and merging

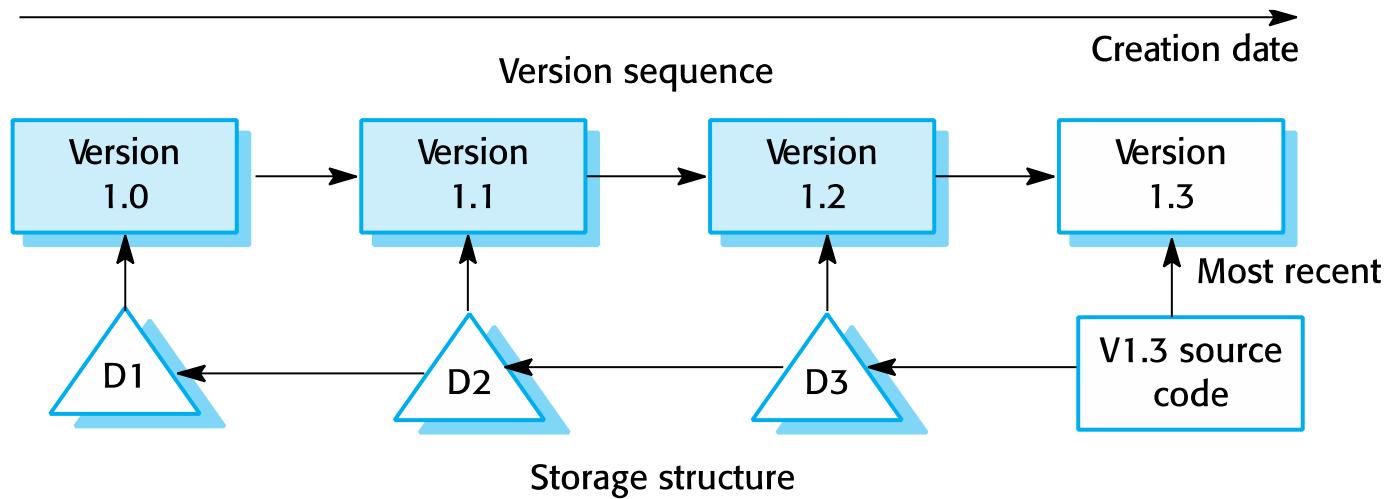
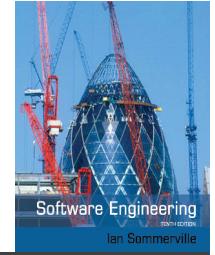


Storage management

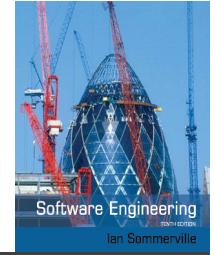


- ✧ When version control systems were first developed, storage management was one of their most important functions.
- ✧ Disk space was expensive and it was important to minimize the disk space used by the different copies of components.
- ✧ Instead of keeping a complete copy of each version, the system stores a list of differences (deltas) between one version and another.
 - By applying these to a master version (usually the most recent version), a target version can be recreated.

Storage management using deltas



Storage management in Git



- ✧ As disk storage is now relatively cheap, Git uses an alternative, faster approach.
- ✧ Git does not use deltas but applies a standard compression algorithm to stored files and their associated meta-information.
- ✧ It does not store duplicate copies of files. Retrieving a file simply involves decompressing it, with no need to apply a chain of operations.
- ✧ Git also uses the notion of packfiles where several smaller files are combined into an indexed single file.

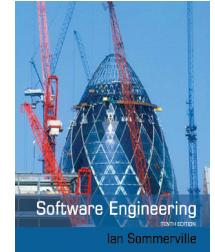


Software Engineering

Ian Sommerville

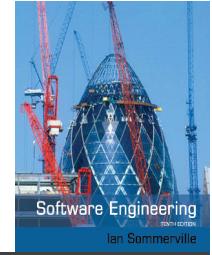
System building

System building



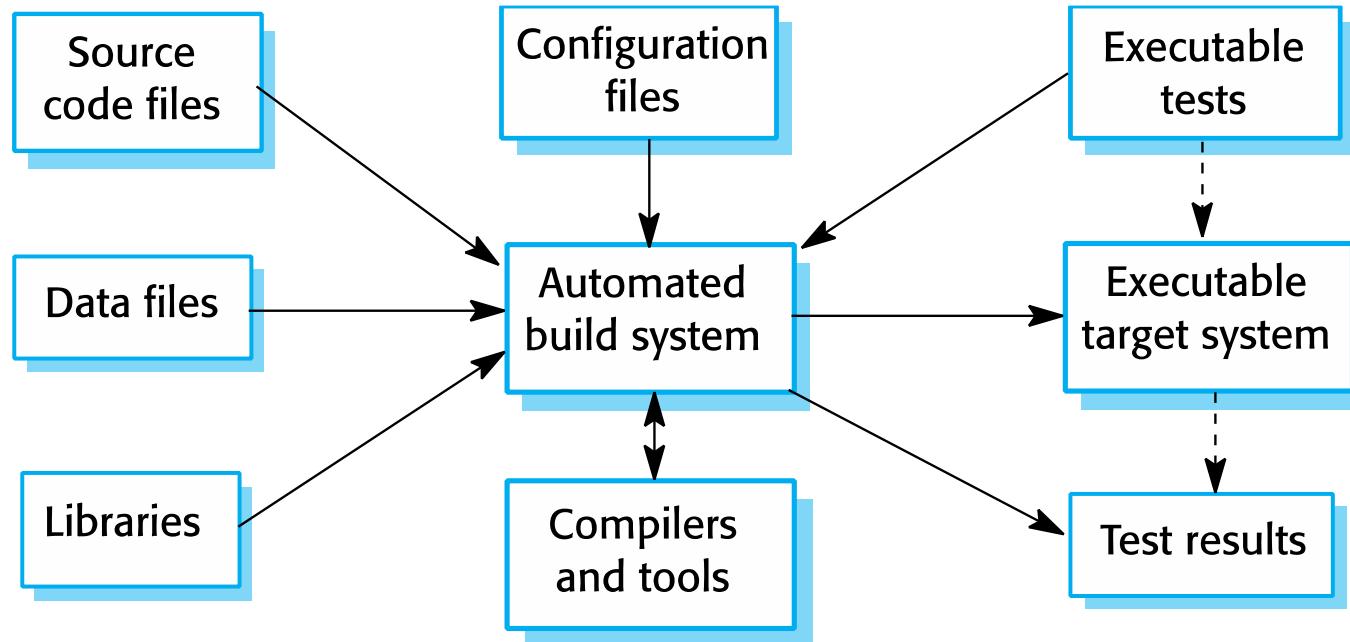
- ✧ System building is the process of creating a complete, executable system by compiling and linking the system components, external libraries, configuration files, etc.
- ✧ System building tools and version management tools must communicate as the build process involves checking out component versions from the repository managed by the version management system.
- ✧ The configuration description used to identify a baseline is also used by the system building tool.

Build platforms

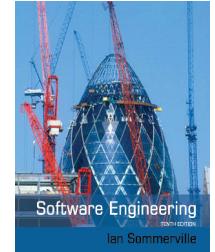


- ✧ The development system, which includes development tools such as compilers, source code editors, etc.
 - Developers check out code from the version management system into a private workspace before making changes to the system.
- ✧ The build server, which is used to build definitive, executable versions of the system.
 - Developers check-in code to the version management system before it is built. The system build may rely on external libraries that are not included in the version management system.
- ✧ The target environment, which is the platform on which the system executes.

System building

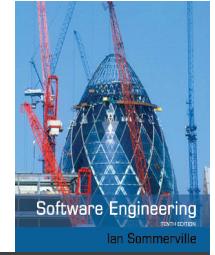


Build system functionality



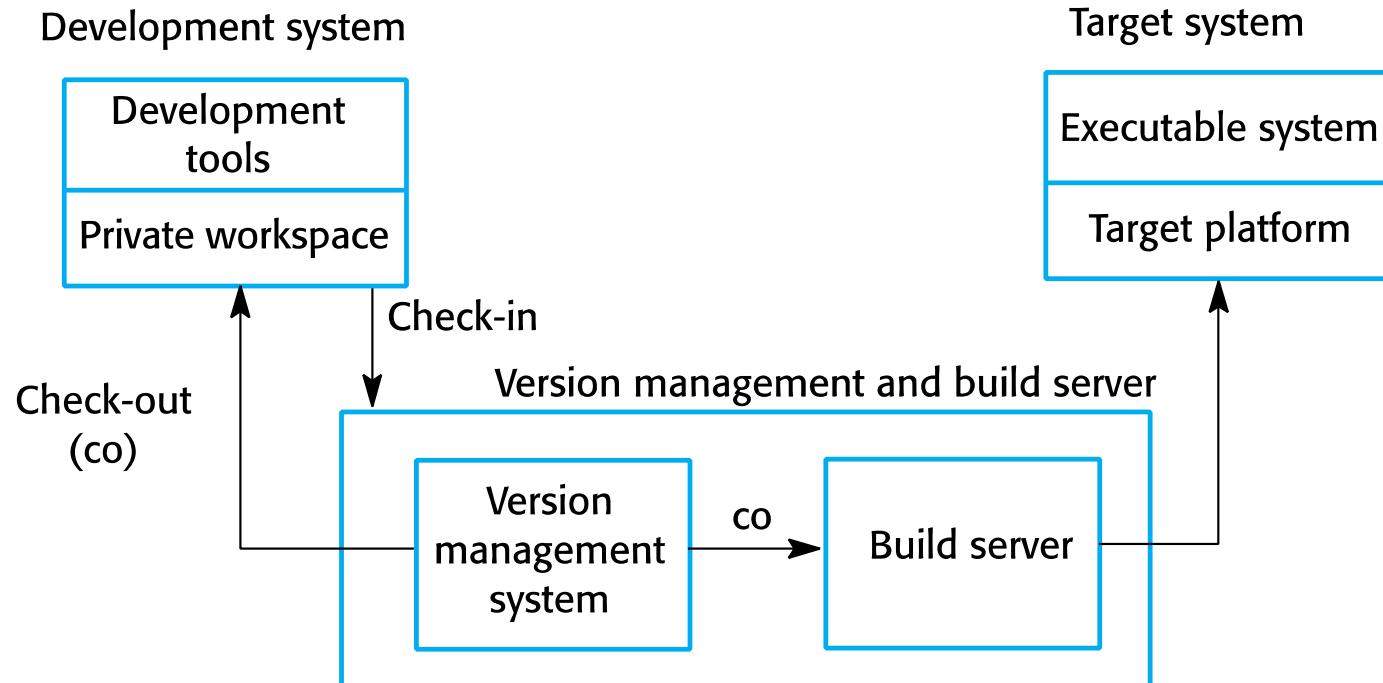
- ✧ Build script generation
- ✧ Version management system integration
- ✧ Minimal re-compilation
- ✧ Executable system creation
- ✧ Test automation
- ✧ Reporting
- ✧ Documentation generation

System platforms

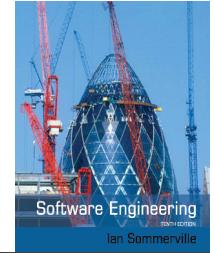


- ✧ The development system, which includes development tools such as compilers, source code editors, etc.
- ✧ The build server, which is used to build definitive, executable versions of the system. This server maintains the definitive versions of a system.
- ✧ The target environment, which is the platform on which the system executes.
 - For real-time and embedded systems, the target environment is often smaller and simpler than the development environment (e.g. a cell phone)

Development, build, and target platforms



Agile building



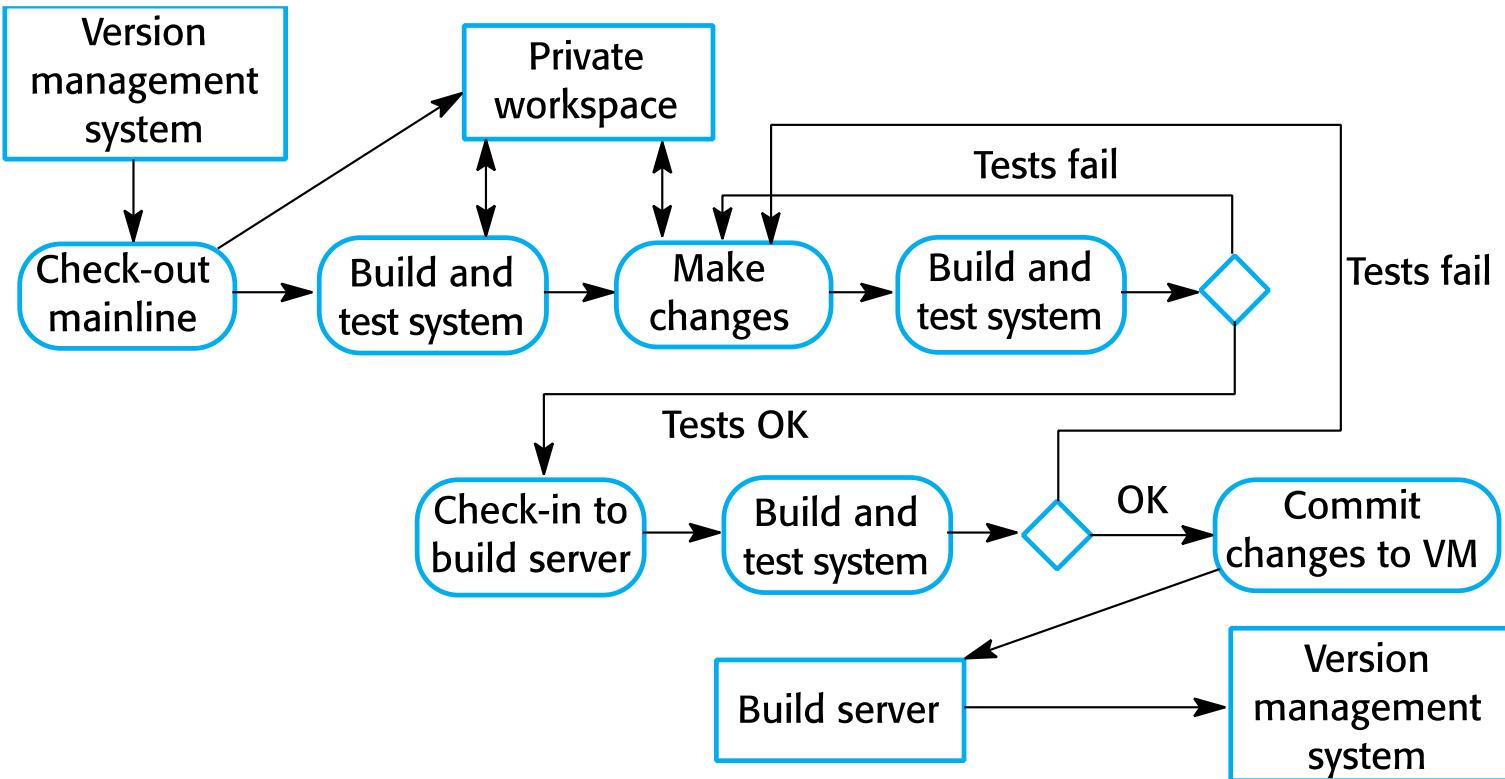
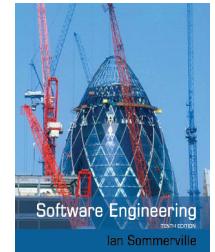
- ✧ Check out the mainline system from the version management system into the developer's private workspace.
- ✧ Build the system and run automated tests to ensure that the built system passes all tests. If not, the build is broken and you should inform whoever checked in the last baseline system. They are responsible for repairing the problem.
- ✧ Make the changes to the system components.
- ✧ Build the system in the private workspace and rerun system tests. If the tests fail, continue editing.

Agile building

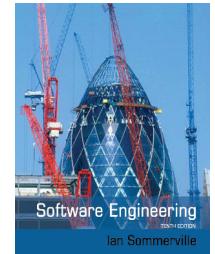


- ✧ Once the system has passed its tests, check it into the build system but do not commit it as a new system baseline.
- ✧ Build the system on the build server and run the tests. You need to do this in case others have modified components since you checked out the system. If this is the case, check out the components that have failed and edit these so that tests pass on your private workspace.
- ✧ If the system passes its tests on the build system, then commit the changes you have made as a new baseline in the system mainline.

Continuous integration



Pros and cons of continuous integration



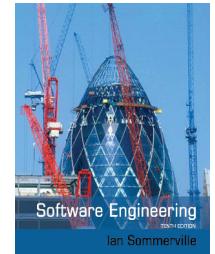
✧ Pros

- The advantage of continuous integration is that it allows problems caused by the interactions between different developers to be discovered and repaired as soon as possible.
- The most recent system in the mainline is the definitive working system.

✧ Cons

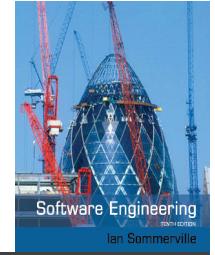
- If the system is very large, it may take a long time to build and test, especially if integration with other application systems is involved.
- If the development platform is different from the target platform, it may not be possible to run system tests in the developer's private workspace.

Daily building



- ✧ The development organization sets a delivery time (say 2 p.m.) for system components.
 - If developers have new versions of the components that they are writing, they must deliver them by that time.
 - A new version of the system is built from these components by compiling and linking them to form a complete system.
 - This system is then delivered to the testing team, which carries out a set of predefined system tests
 - Faults that are discovered during system testing are documented and returned to the system developers. They repair these faults in a subsequent version of the component.

Minimizing recompilation



- ✧ Tools to support system building are usually designed to minimize the amount of compilation that is required.
- ✧ They do this by checking if a compiled version of a component is available. If so, there is no need to recompile that component.
- ✧ A unique signature identifies each source and object code version and is changed when the source code is edited.
- ✧ By comparing the signatures on the source and object code files, it is possible to decide if the source code was used to generate the object code component.

File identification

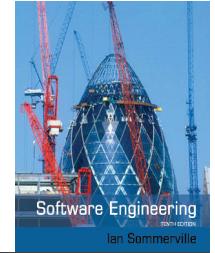


✧ Modification timestamps

- The signature on the source code file is the time and date when that file was modified. If the source code file of a component has been modified after the related object code file, then the system assumes that recompilation to create a new object code file is necessary.

✧ Source code checksums

- The signature on the source code file is a checksum calculated from data in the file. A checksum function calculates a unique number using the source text as input. If you change the source code (even by 1 character), this will generate a different checksum. You can therefore be confident that source code files with different checksums are actually different.



Timestamps vs checksums

Software Engineering
Ian Sommerville

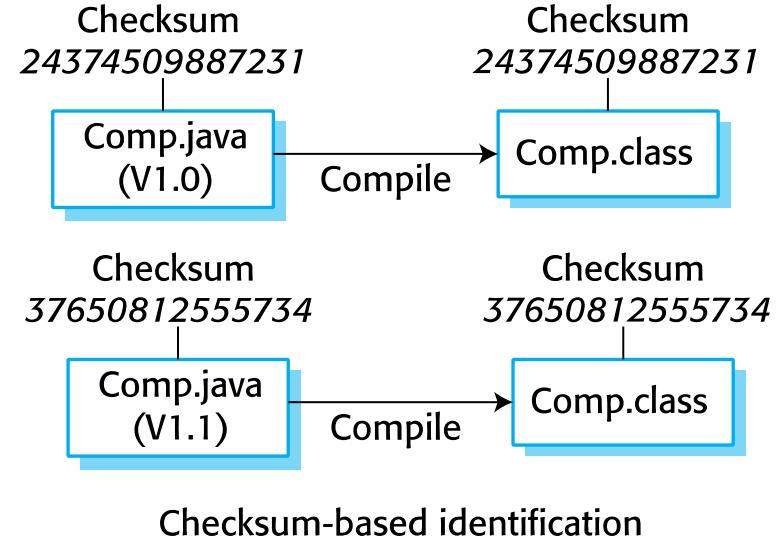
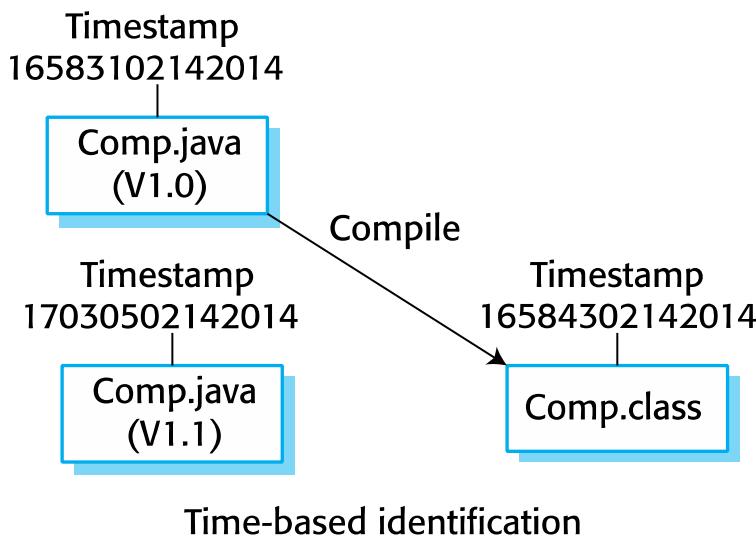
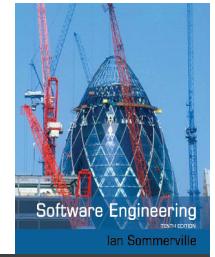
✧ Timestamps

- Because source and object files are linked by name rather than an explicit source file signature, it is not usually possible to build different versions of a source code component into the same directory at the same time, as these would generate object files with the same name.

✧ Checksums

- When you recompile a component, it does not overwrite the object code, as would normally be the case when the timestamp is used. Rather, it generates a new object code file and tags it with the source code signature. Parallel compilation is possible and different versions of a component may be compiled at the same time.

Linking source and object code



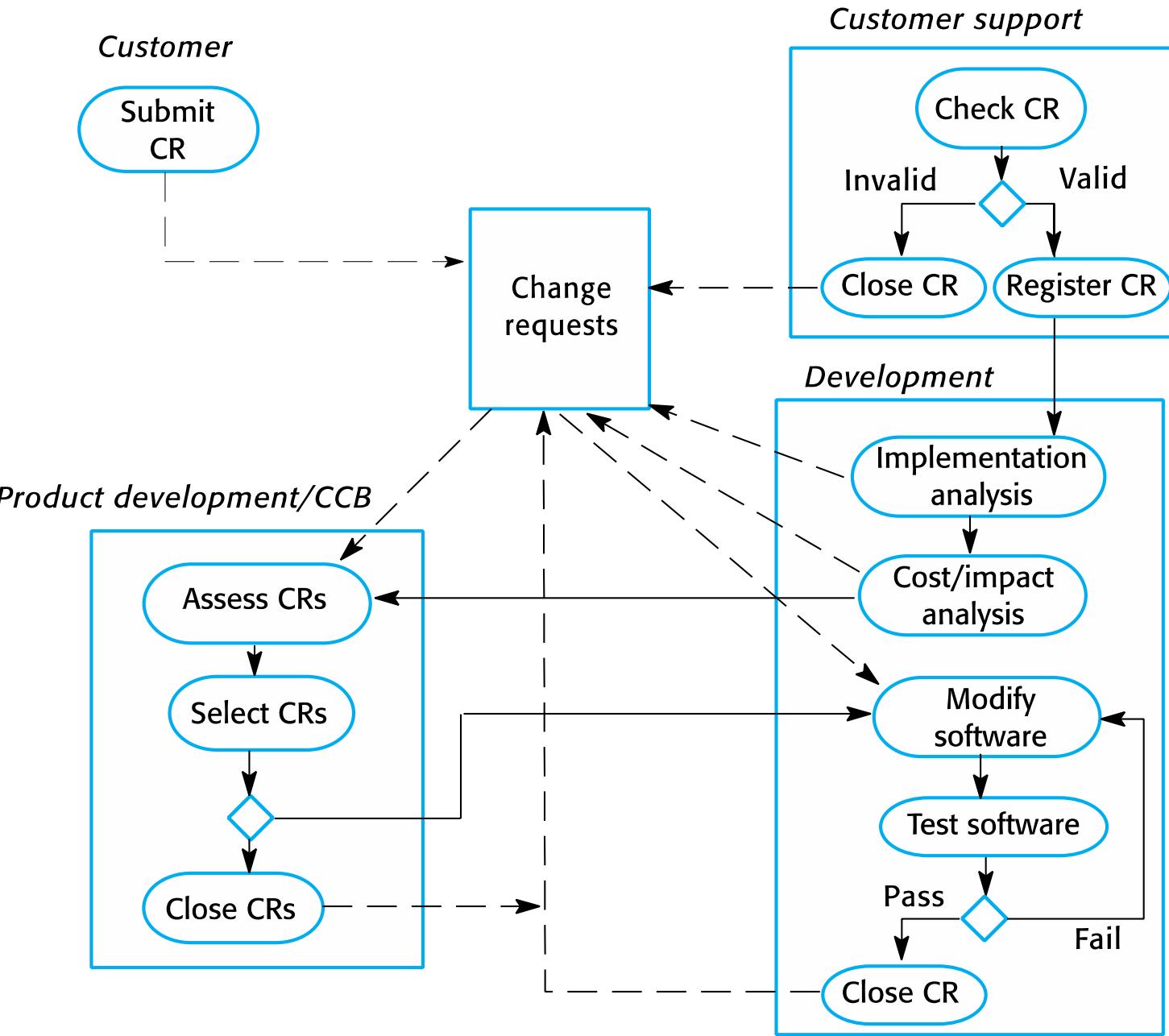


Change management

Change management

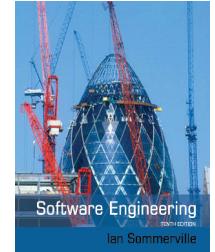


- ✧ Organizational needs and requirements change during the lifetime of a system, bugs have to be repaired and systems have to adapt to changes in their environment.
- ✧ Change management is intended to ensure that system evolution is a managed process and that priority is given to the most urgent and cost-effective changes.
- ✧ The change management process is concerned with analyzing the costs and benefits of proposed changes, approving those changes that are worthwhile and tracking which components in the system have been changed.



The change management process

A partially completed change request form (a)



Change Request Form

Project: SICSA/AppProcessing

Number: 23/02

Change requester: I. Sommerville

Date: 20/07/12

Requested change: The status of applicants (rejected, accepted, etc.) should be shown visually in the displayed list of applicants.

Change analyzer: R. Looek

Analysis date: 25/07/12

Components affected: ApplicantListDisplay, StatusUpdater

Associated components: StudentDatabase

A partially completed change request form (b)



Change Request Form

Change assessment: Relatively simple to implement by changing the display color according to status. A table must be added to relate status to colors. No changes to associated components are required.

Change priority: Medium

Change implementation:

Estimated effort: 2 hours

Date to SGA app. team: 28/07/12

CCB decision date: 30/07/12

Decision: Accept change. Change to be implemented in Release 1.2

Change implementor:

Date of change:

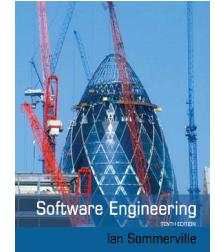
Date submitted to QA:

QA decision:

Date submitted to CM:

Comments:

Factors in change analysis



- ✧ The consequences of not making the change
- ✧ The benefits of the change
- ✧ The number of users affected by the change
- ✧ The costs of making the change
- ✧ The product release cycle

Derivation history



// SICSA project (XEP 6087)

//

// APP-SYSTEM/AUTH/RBAC/USER_ROLE

//

// Object: currentRole

// Author: R. Looek

// Creation date: 13/11/2012

//

// © St Andrews University 2012

//

// Modification history

// Version Modifier Date

Change

Reason

// 1.0 J. Jones 11/11/2009

Add header

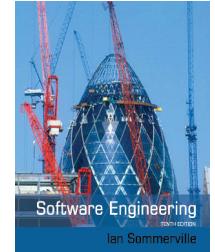
Submitted to CM

// 1.1 R. Looek 13/11/2012

New field

Change req. R07/02

Change management and agile methods

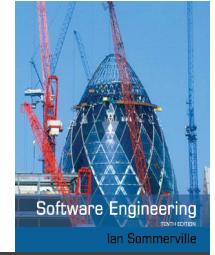


- ✧ In some agile methods, customers are directly involved in change management.
- ✧ They propose a change to the requirements and work with the team to assess its impact and decide whether the change should take priority over the features planned for the next increment of the system.
- ✧ Changes to improve the software improvement are decided by the programmers working on the system.
- ✧ Refactoring, where the software is continually improved, is not seen as an overhead but as a necessary part of the development process.



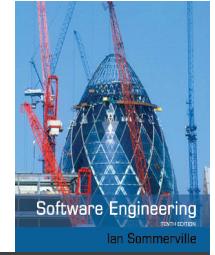
Release management

Release management



- ✧ A system release is a version of a software system that is distributed to customers.
- ✧ For mass market software, it is usually possible to identify two types of release: major releases which deliver significant new functionality, and minor releases, which repair bugs and fix customer problems that have been reported.
- ✧ For custom software or software product lines, releases of the system may have to be produced for each customer and individual customers may be running several different releases of the system at the same time.

Release components



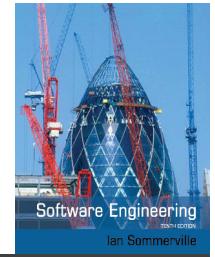
- ✧ As well as the executable code of the system, a release may also include:
 - configuration files defining how the release should be configured for particular installations;
 - data files, such as files of error messages, that are needed for successful system operation;
 - an installation program that is used to help install the system on target hardware;
 - electronic and paper documentation describing the system;
 - packaging and associated publicity that have been designed for that release.

Factors influencing system release planning

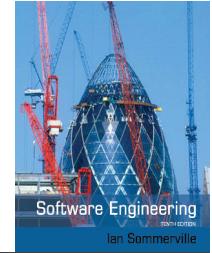


Factor	Description
Competition	For mass-market software, a new system release may be necessary because a competing product has introduced new features and market share may be lost if these are not provided to existing customers.
Marketing requirements	The marketing department of an organization may have made a commitment for releases to be available at a particular date.
Platform changes	You may have to create a new release of a software application when a new version of the operating system platform is released.
Technical quality of the system	If serious system faults are reported which affect the way in which many customers use the system, it may be necessary to issue a fault repair release. Minor system faults may be repaired by issuing patches (usually distributed over the Internet) that can be applied to the current release of the system.

Release creation



- ✧ The executable code of the programs and all associated data files must be identified in the version control system.
- ✧ Configuration descriptions may have to be written for different hardware and operating systems.
- ✧ Update instructions may have to be written for customers who need to configure their own systems.
- ✧ Scripts for the installation program may have to be written.
- ✧ Web pages have to be created describing the release, with links to system documentation.
- ✧ When all information is available, an executable master image of the software must be prepared and handed over for distribution to customers or sales outlets.



Release tracking

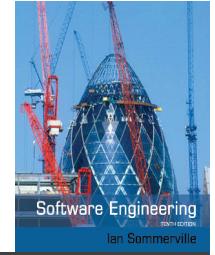
- ✧ In the event of a problem, it may be necessary to reproduce exactly the software that has been delivered to a particular customer.
- ✧ When a system release is produced, it must be documented to ensure that it can be re-created exactly in the future.
- ✧ This is particularly important for customized, long-lifetime embedded systems, such as those that control complex machines.
 - Customers may use a single release of these systems for many years and may require specific changes to a particular software system long after its original release date.

Release reproduction



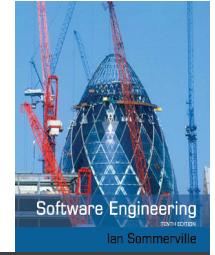
- ✧ To document a release, you have to record the specific versions of the source code components that were used to create the executable code.
- ✧ You must keep copies of the source code files, corresponding executables and all data and configuration files.
- ✧ You should also record the versions of the operating system, libraries, compilers and other tools used to build the software.

Release planning



- ✧ As well as the technical work involved in creating a release distribution, advertising and publicity material have to be prepared and marketing strategies put in place to convince customers to buy the new release of the system.
- ✧ Release timing
 - If releases are too frequent or require hardware upgrades, customers may not move to the new release, especially if they have to pay for it.
 - If system releases are too infrequent, market share may be lost as customers move to alternative systems.

Software as a service

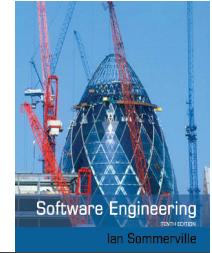


- ✧ Delivering software as a service (SaaS) reduces the problems of release management.
- ✧ It simplifies both release management and system installation for customers.
- ✧ The software developer is responsible for replacing the existing release of a system with a new release and this is made available to all customers at the same time.



Key points

- ✧ Configuration management is the management of an evolving software system. When maintaining a system, a CM team is put in place to ensure that changes are incorporated into the system in a controlled way and that records are maintained with details of the changes that have been implemented.
- ✧ The main configuration management processes are concerned with version management, system building, change management, and release management.
- ✧ Version management involves keeping track of the different versions of software components as changes are made to them.



Key points

- ✧ System building is the process of assembling system components into an executable program to run on a target computer system.
- ✧ Software should be frequently rebuilt and tested immediately after a new version has been built. This makes it easier to detect bugs and problems that have been introduced since the last build.
- ✧ Change management involves assessing proposals for changes from system customers and other stakeholders and deciding if it is cost-effective to implement these in a new version of a system.
- ✧ System releases include executable code, data files, configuration files and documentation. Release management involves making decisions on system release dates, preparing all information for distribution and documenting each system release.



Chapter 4 – Requirements Engineering

30/10/2014 Chapter 4 Requirements Engineering 1



Topics covered

- ✧ Functional and non-functional requirements
- ✧ Requirements engineering processes
- ✧ Requirements elicitation
- ✧ Requirements specification
- ✧ Requirements validation
- ✧ Requirements change

30/10/2014 Chapter 4 Requirements Engineering 2



Requirements engineering

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

30/10/2014 Chapter 4 Requirements Engineering 3



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.

30/10/2014 Chapter 4 Requirements Engineering 4



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."

30/10/2014 Chapter 4 Requirements Engineering 5



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.

30/10/2014 Chapter 4 Requirements Engineering 6

User and system requirements

User requirements definition

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

System requirements specification

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

30/10/2014 Chapter 4 Requirements Engineering 7

Readers of different types of requirements specification

```

graph LR
    UR[User requirements] --> CM[Client managers  
System end-users  
Client engineers  
Contractor managers  
System architects]
    SR[System requirements] --> SEU[System end-users  
Client engineers  
System architects  
Software developers]
  
```

30/10/2014 Chapter 4 Requirements Engineering 8

System stakeholders

- ✧ Any person or organization who is affected by the system in some way and so who has a legitimate interest
- ✧ Stakeholder types
 - End users
 - System managers
 - System owners
 - External stakeholders

30/10/2014 Chapter 4 Requirements Engineering 9

Stakeholders in the Mentcare system

- ✧ 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.

30/10/2014 Chapter 4 Requirements Engineering 10

Stakeholders in the Mentcare system

- ✧ 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.

30/10/2014 Chapter 4 Requirements Engineering 11

Agile methods and requirements

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

30/10/2014 Chapter 4 Requirements Engineering 12



Functional and non-functional requirements

30/10/2014 Chapter 4 Requirements Engineering 13

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

30/10/2014

Chapter 4 Requirements Engineering

14



Functional requirements

30/10/2014 Chapter 4 Requirements Engineering 15

Mentcare system: functional requirements

- ❖ 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.

30/10/2014

Chapter 4 Requirements Engineering

16



Requirements imprecision

30/10/2014 Chapter 4 Requirements Engineering 17

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, because of system and environmental complexity, it is impossible to produce a complete and consistent requirements document.

30/10/2014

Chapter 4 Requirements Engineering

18

Non-functional requirements

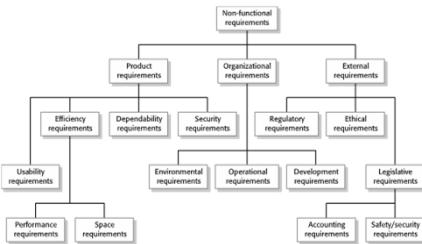
- ◊ These define system properties and constraints e.g. reliability, response time and storage requirements. Constraints are I/O device capability, system representations, etc.
- ◊ Process requirements may also be specified mandating a particular IDE, programming language or development method.
- ◊ Non-functional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

30/10/2014

Chapter 4 Requirements Engineering

19

Types of nonfunctional requirement



30/10/2014

Chapter 4 Requirements Engineering

20

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.

30/10/2014

Chapter 4 Requirements Engineering

21

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.

30/10/2014

Chapter 4 Requirements Engineering

22

Examples of nonfunctional requirements in the Mentcare system

Product requirement

The Mentcare system 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 Mentcare 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.

30/10/2014

Chapter 4 Requirements Engineering

23

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.

30/10/2014

Chapter 4 Requirements Engineering

24

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)

30/10/2014

Chapter 4 Requirements Engineering

25

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

30/10/2014

Chapter 4 Requirements Engineering

26

Requirements engineering processes



30/10/2014

Chapter 4 Requirements Engineering

27

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.

30/10/2014

Chapter 4 Requirements Engineering

28

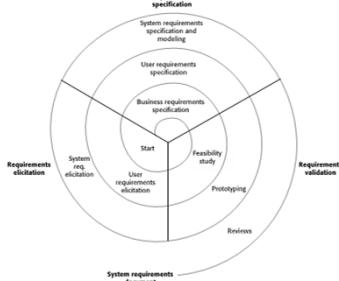
A spiral view of the requirements engineering process



30/10/2014

Chapter 4 Requirements Engineering

29



Requirements elicitation



30/10/2014

Chapter 4 Requirements Engineering

30

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**.

30/10/2014 Chapter 4 Requirements Engineering 31

Requirements elicitation



30/10/2014 Chapter 4 Requirements Engineering 32

Requirements elicitation



◊ 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.

30/10/2014 Chapter 4 Requirements Engineering 33

Problems of requirements elicitation



◊ Stakeholders don't know what they really want.

◊ Stakeholders express requirements in their own terms.

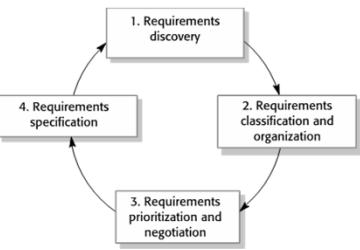
◊ Different stakeholders may have conflicting requirements.

◊ Organisational and political factors may influence the system requirements.

◊ The requirements change during the analysis process. New stakeholders may emerge and the business environment may change.

30/10/2014 Chapter 4 Requirements Engineering 34

The requirements elicitation and analysis process

30/10/2014 Chapter 4 Requirements Engineering 35

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.

30/10/2014 Chapter 4 Requirements Engineering 36

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.

30/10/2014

Chapter 4 Requirements Engineering

37

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.

30/10/2014

Chapter 4 Requirements Engineering

38

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.
- ✧ Interviewers need to be open-minded without pre-conceived ideas of what the system should do
- ✧ You need to prompt the user to talk about the system by suggesting requirements rather than simply asking them what they want.

30/10/2014

Chapter 4 Requirements Engineering

39

Problems with interviews



- ✧ Application specialists may use language to describe their work that isn't easy for the requirements engineer to understand.
- ✧ 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.

30/10/2014

Chapter 4 Requirements Engineering

40

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.

30/10/2014

Chapter 4 Requirements Engineering

41

Scope of ethnography



- ✧ Requirements that are derived from the way that people actually work rather than the way I 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.

30/10/2014

Chapter 4 Requirements Engineering

42

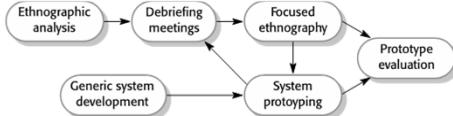
Focused ethnography



❖ 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.

30/10/2014 Chapter 4 Requirements Engineering 43

Ethnography and prototyping for requirements analysis

30/10/2014 Chapter 4 Requirements Engineering 44

Stories and scenarios



❖ Scenarios and user stories are real-life examples of how a system can be used.
 ❖ Stories and scenarios are a description of how a system may be used for a particular task.
 ❖ Because they are based on a practical situation, stakeholders can relate to them and can comment on their situation with respect to the story.

30/10/2014 Chapter 4 Requirements Engineering 45

Photo sharing in the classroom (iLearn)



❖ Jack is a primary school teacher in Ullapool (a village in northern Scotland). He has decided that a class project should be focused around the fishing industry in the area, looking at the history, development and economic impact of fishing. As part of this, pupils are asked to gather and share reminiscences from relatives, use newspaper archives and collect old photographs related to fishing and fishing communities in the area. Pupils use an iLearn wiki to gather together fishing stories and SCARAN (a history resources site) to access newspaper archives and photographs. However, Jack also needs a photo sharing site as he wants pupils to take and comment on each others photos and to upload scans of old photographs that they may have in their families.

Jack sends an email to a primary school teachers group, which he is a member of to see if anyone can recommend an appropriate system. Two teachers reply and both suggest that he uses KidsTakePics, a photo sharing site that allows teachers to check and moderate content. As KidsTakePics is not integrated with the iLearn authentication service, he sets up a teacher and a class account. He uses the iLearn setup service to add KidsTakePics to the services seen by the pupils in his class so that when they log in, they can immediately use the system to upload photos from their mobile devices and class computers.

30/10/2014 Chapter 4 Requirements Engineering 46

Scenarios



❖ A structured form of user story
 ❖ Scenarios 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.

30/10/2014 Chapter 4 Requirements Engineering 47

Uploading photos iLearn



❖ **Initial assumption:** A user or a group of users have one or more digital photographs to be uploaded to the picture sharing site. These are saved on either a tablet or laptop computer. They have successfully logged on to KidsTakePics.

❖ **Normal:** The user chooses upload photos and they are prompted to select the photos to be uploaded on their computer and to select the project name under which the photos will be stored. They should also be given the option of inputting keywords that should be associated with each uploaded photo. Uploaded photos are named by creating a conjunction of the user name with the filename of the photo on the local computer.

❖ On completion of the upload, the system automatically sends an email to the project moderator asking them to check new content and generates an on-screen message to the user that this has been done.

30/10/2014 Chapter 4 Requirements Engineering 48

Uploading photos

Software Engineering

- ❖ **What can go wrong:**
- ❖ No moderator is associated with the selected project. An email is automatically generated to the school administrator asking them to nominate a project moderator. Users should be informed that there could be a delay in making their photos visible.
- ❖ Photos with the same name have already been uploaded by the same user. The user should be asked if they wish to re-upload the photos with the same name, rename the photos or cancel the upload. If they chose to re-upload the photos, the originals are overwritten. If they chose to rename the photos, a new name is automatically generated by adding a number to the existing file name.
- ❖ **Other activities:** The moderator may be logged on to the system and may approve photos as they are uploaded.
- ❖ **System state on completion:** User is logged on. The selected photos have been uploaded and assigned a status 'awaiting moderation'. Photos are visible to the moderator and to the user who uploaded them.

30/10/2014 Chapter 4 Requirements Engineering 49

Requirements specification

Software Engineering

30/10/2014 Chapter 4 Requirements Engineering 50

Requirements specification

Software Engineering

- ❖ 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.

30/10/2014 Chapter 4 Requirements Engineering 51

Ways of writing a system requirements specification

Software Engineering

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.

30/10/2014 Chapter 4 Requirements Engineering 52

Requirements and design

Software Engineering

- ❖ 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.

30/10/2014 Chapter 4 Requirements Engineering 53

Natural language specification

Software Engineering

- ❖ 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.

30/10/2014 Chapter 4 Requirements Engineering 54

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.

30/10/2014

Chapter 4 Requirements Engineering

55

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.

30/10/2014

Chapter 4 Requirements Engineering

56

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.*)

30/10/2014

Chapter 4 Requirements Engineering

57

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.

30/10/2014

Chapter 4 Requirements Engineering

58

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.

30/10/2014

Chapter 4 Requirements Engineering

59

A structured specification of a requirement for an insulin pump



Insulin Pump/Control Software/SRS/3.3.2

Function Compute insulin dose: safe sugar level.

Description

Computes the dose of insulin to be delivered when the current measured sugar level is in the safe zone between 3 and 7 units.

Inputs Current sugar reading (r2); the previous two readings (r0 and r1).

Source Current sugar reading from sensor. Other readings from memory.

Outputs CompDose—the dose in insulin to be delivered.

Destination Main control loop.

30/10/2014

Chapter 4 Requirements Engineering

60

A structured specification of a requirement for an insulin pump

Action

CompDose is zero if the sugar level is stable or falling or if the level is increasing but the rate of increase is decreasing. If the level is increasing and the rate of increase is increasing, then CompDose is computed by dividing the difference between the current sugar level and the previous level by 4 and rounding the result. If the result, is rounded to zero then CompDose is set to the minimum dose that can be delivered.

Requirements

Two previous readings so that the rate of change of sugar level can be computed.

Pre-condition

The insulin reservoir contains at least the maximum allowed single dose of insulin.

Post-condition r0 is replaced by r1 then r1 is replaced by r2.

Side effects None.

30/10/2014

Chapter 4 Requirements Engineering

61

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 ($r2 < r1$)	CompDose = 0
Sugar level stable ($r2 = r1$)	CompDose = 0
Sugar level increasing and rate of increase decreasing $((r2 - r1) < (r1 - r0))$	CompDose = 0
Sugar level increasing and rate of increase stable or increasing $((r2 - r1) \geq (r1 - r0))$	$\text{CompDose} = \text{round}((r2 - r1)/4)$ If rounded result = 0 then $\text{CompDose} = \text{MinimumDose}$

30/10/2014

Chapter 4 Requirements Engineering

63

Use cases

- Use-cases are a kind of scenario that are included in the UML.

- Use cases identify the actors in an interaction and which describe the interaction itself.

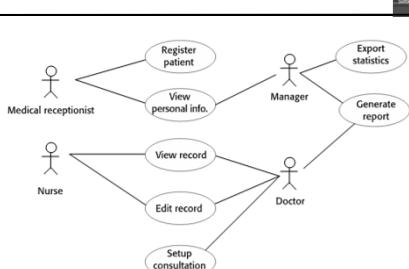
- A set of use cases should describe all possible interactions with the system.

- High-level graphical model supplemented by more detailed tabular description (see Chapter 5).

- UML 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 Mentcare system



30/10/2014

Chapter 4 Requirements Engineering

65

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.



66

Users of a requirements document

```

graph LR
    SC[System customers] --> S1[Specify the requirements and read them to check that they meet their needs. Customers specify changes to the requirements.]
    M[Managers] --> S2[Use the requirements document to plan a bid for the system and to plan the system development process.]
    SE[System engineers] --> S3[Use the requirements to understand what system is to be developed.]
    STE[System test engineers] --> S4[Use the requirements to develop validation tests for the system.]
    SME[System maintenance engineers] --> S5[Use the requirements to understand the system and the relationships between its parts.]
  
```

30/10/2014 Chapter 4 Requirements Engineering 67

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.

30/10/2014 Chapter 4 Requirements Engineering 68

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.

30/10/2014 Chapter 4 Requirements Engineering 69

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.

30/10/2014 Chapter 4 Requirements Engineering 70

Requirements validation

30/10/2014 Chapter 4 Requirements Engineering 71

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.

30/10/2014 Chapter 4 Requirements Engineering 72

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?

30/10/2014

Chapter 4 Requirements Engineering

73

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.

30/10/2014

Chapter 4 Requirements Engineering

74

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.

30/10/2014

Chapter 4 Requirements Engineering

75

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?

30/10/2014

Chapter 4 Requirements Engineering

76

Requirements change



30/10/2014

Chapter 4 Requirements Engineering

77

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.

30/10/2014

Chapter 4 Requirements Engineering

78

Changing requirements



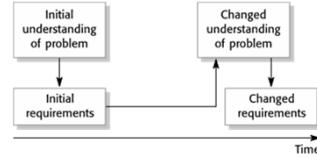
- ✧ Large systems usually have a diverse user community, with many users having different requirements and priorities that may be conflicting or contradictory.
- The final system requirements are inevitably a compromise between them and, with experience, it is often discovered that the balance of support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

30/10/2014

Chapter 4 Requirements Engineering

79

Requirements evolution



30/10/2014

Chapter 4 Requirements Engineering

80

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.

30/10/2014

Chapter 4 Requirements Engineering

81

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.

30/10/2014

Chapter 4 Requirements Engineering

82

Requirements change management



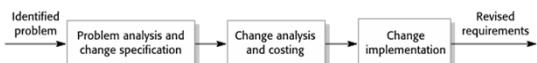
- ✧ Deciding if a requirements change should be accepted
 - *Problem analysis and change specification*
 - During this stage, the problem or the change proposal is analyzed to check that it is valid. This analysis is fed back to the change requestor who may respond with a more specific requirements change 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.

30/10/2014

Chapter 4 Requirements Engineering

83

Requirements change management



30/10/2014

Chapter 4 Requirements Engineering

84

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.

30/10/2014

Chapter 4 Requirements Engineering

85

Key points

- ✧ The requirements engineering process is an iterative process that includes requirements elicitation, specification and validation.
- ✧ Requirements elicitation is an iterative process that can be represented as a spiral of activities – requirements discovery, requirements classification and organization, requirements negotiation and requirements documentation.
- ✧ You can use a range of techniques for requirements elicitation including interviews and ethnography. User stories and scenarios may be used to facilitate discussions.

30/10/2014

Chapter 4 Requirements Engineering

86

Key points

- ✧ Requirements specification is the process of formally documenting the user and system requirements and creating a software requirements document.
- ✧ 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.

30/10/2014

Chapter 4 Requirements Engineering

87

Key points

- ✧ 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.

30/10/2014

Chapter 4 Requirements Engineering

88