



## Chapter 1- Introduction

Ian Sommerville,

*Software Engineering*, 10<sup>th</sup> Edition

Pearson Education, Addison-Wesley

Note: These are a slightly modified version of Ch1 slides available from the author's site <http://iansommerville.com/software-engineering-book/>



# 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 GDP 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*

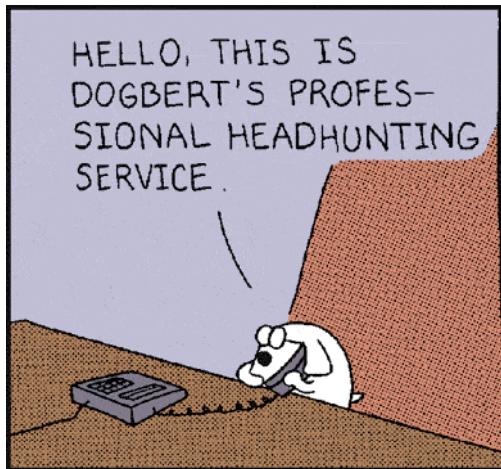
- As new software engineering techniques help us to build larger, more complex systems, the demands change. Systems have to be built and delivered more quickly; larger, even more complex systems are required; systems have to have new capabilities that were previously thought to be impossible.

## ◊ *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





## 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?

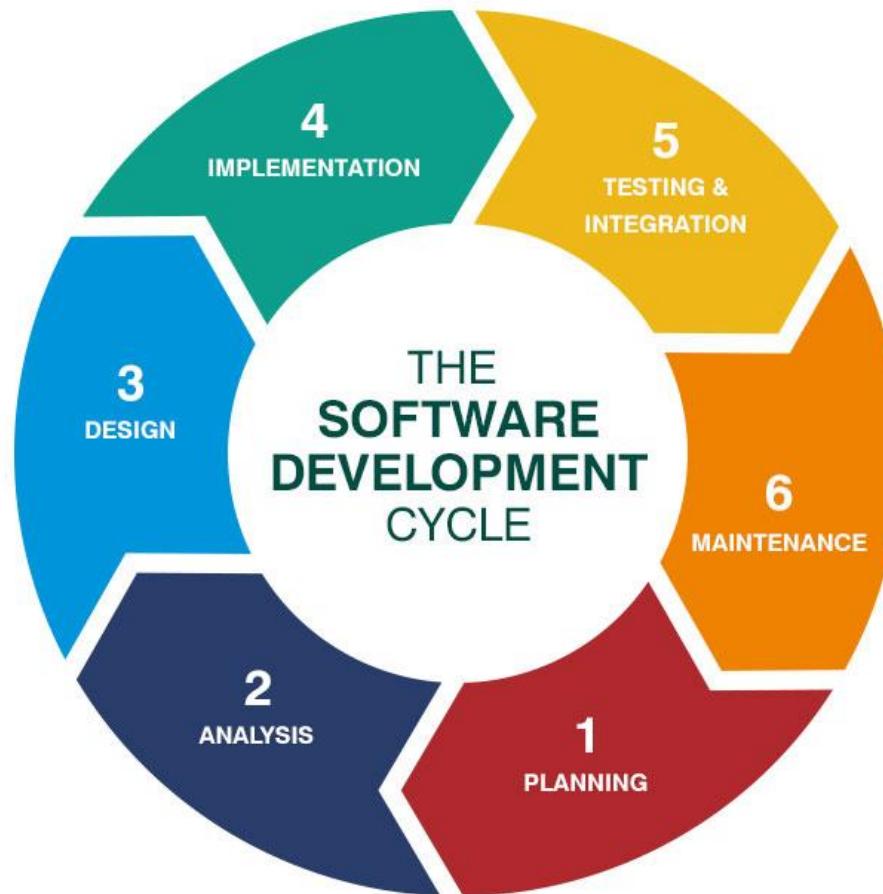
- ◇ Good software should deliver the required functionality and performance to the user and should be maintainable, dependable and usable.



# What is software engineering?



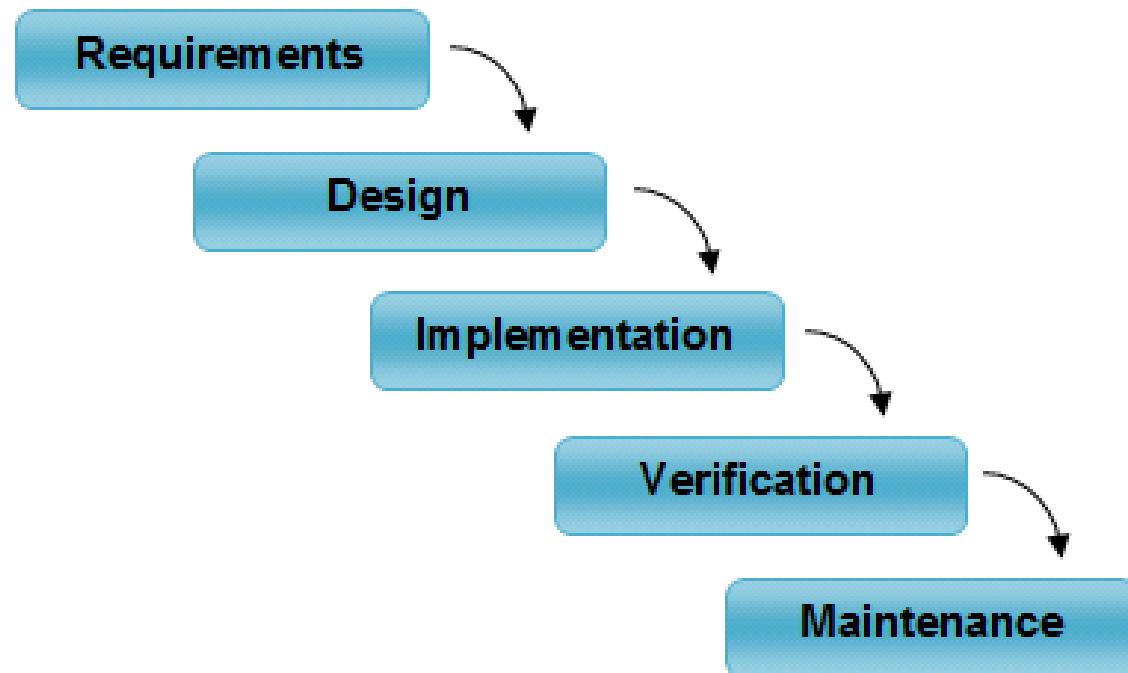
- ◊ Software engineering is an engineering discipline that is concerned with all aspects of software production



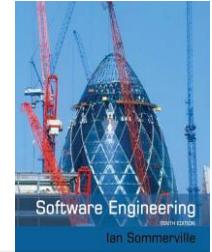
# What are the **fundamental software engineering activities**?



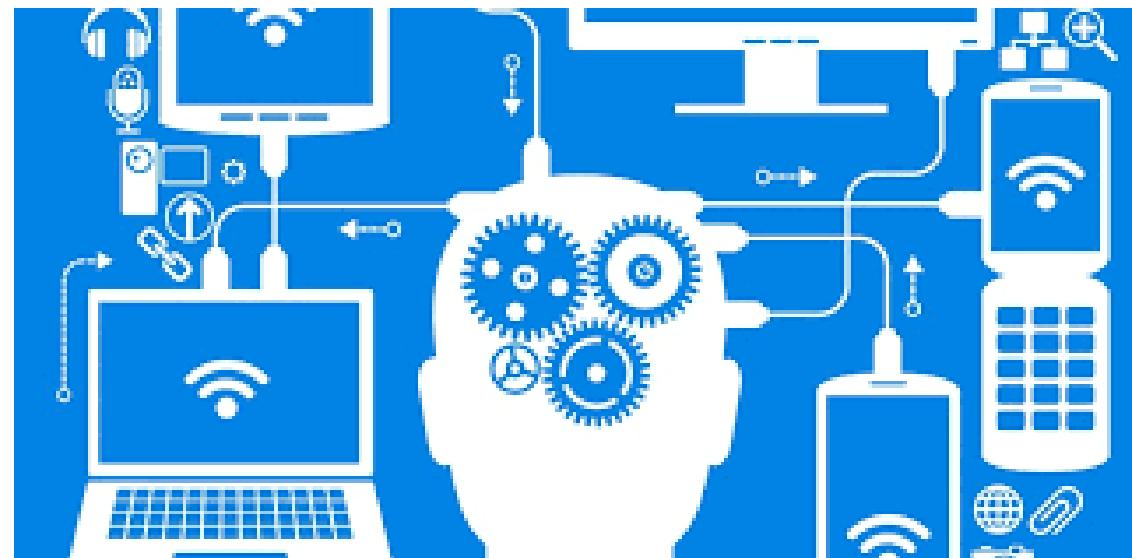
- ◊ Software specification, software development, software validation and software evolution.



# What is the difference between **software engineering** and **computer science**?



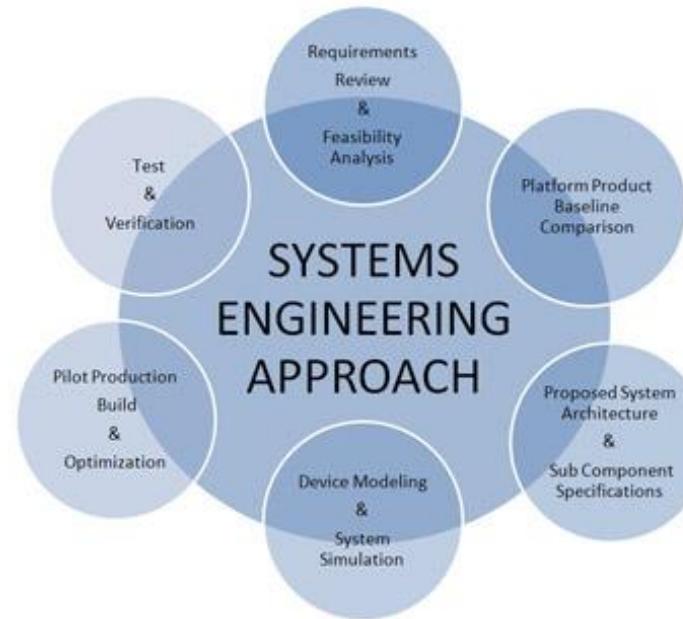
- ◊ Computer science focuses on theory and fundamentals; software engineering is concerned with the practicalities of developing and delivering useful software.



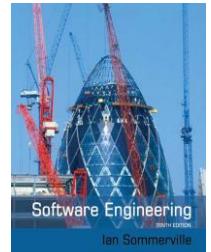
# What is the difference between **software engineering** and **system engineering**?



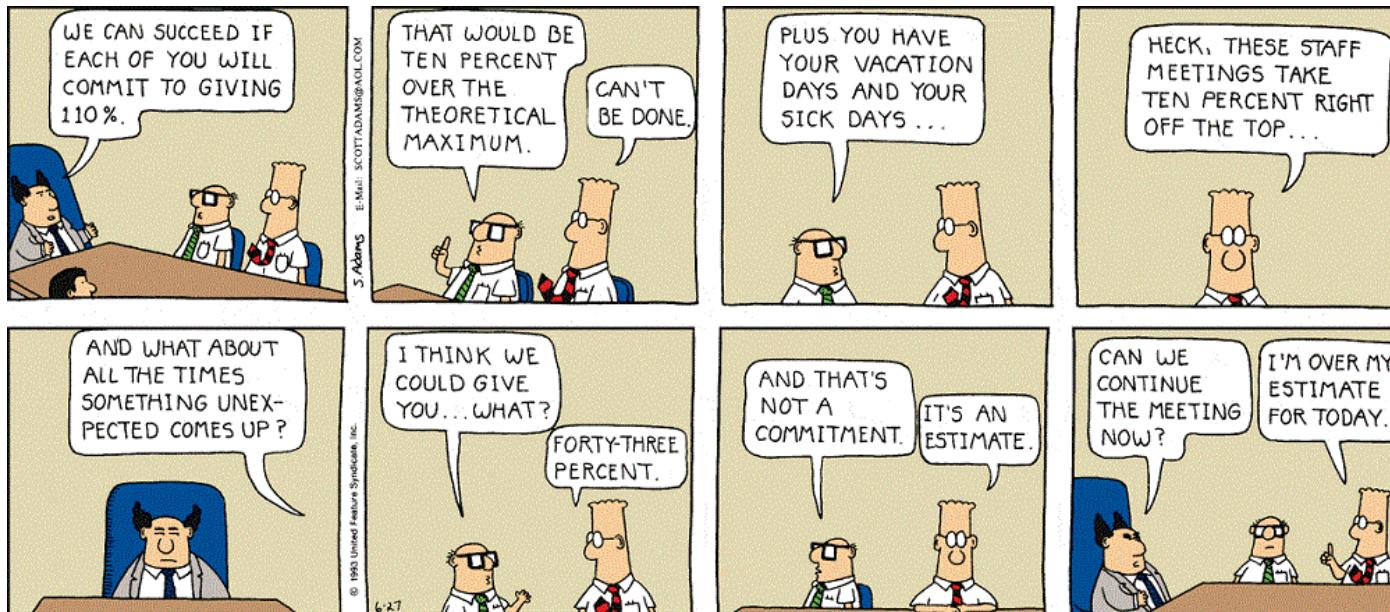
- ◊ System engineering is concerned with all aspects of computer-based systems development including hardware, software and process engineering. Software engineering is part of this more general process



# What are the **key challenges** facing software engineering?



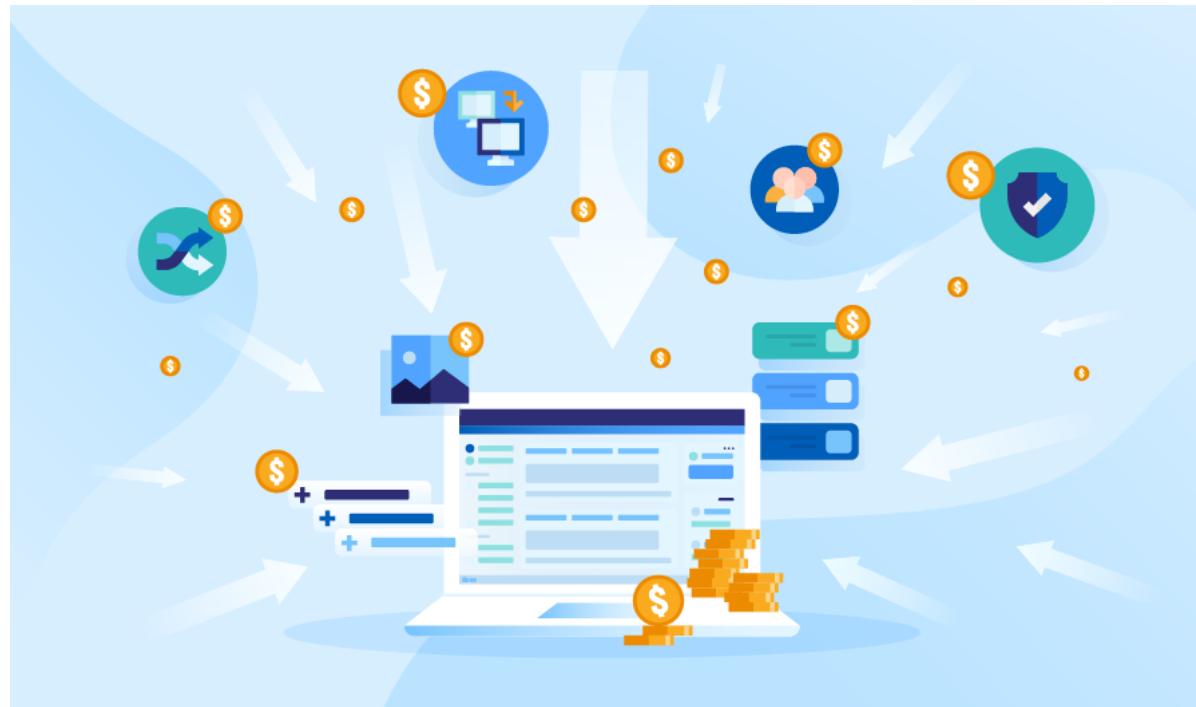
- ◊ Coping with increasing diversity, demands for reduced delivery times and developing trustworthy software.





# 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.



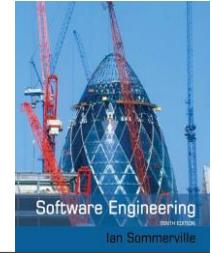
# What are the **best** software engineering techniques and methods?



- ◇ 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 analysable 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 or software product owner.

## ◇ 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

---

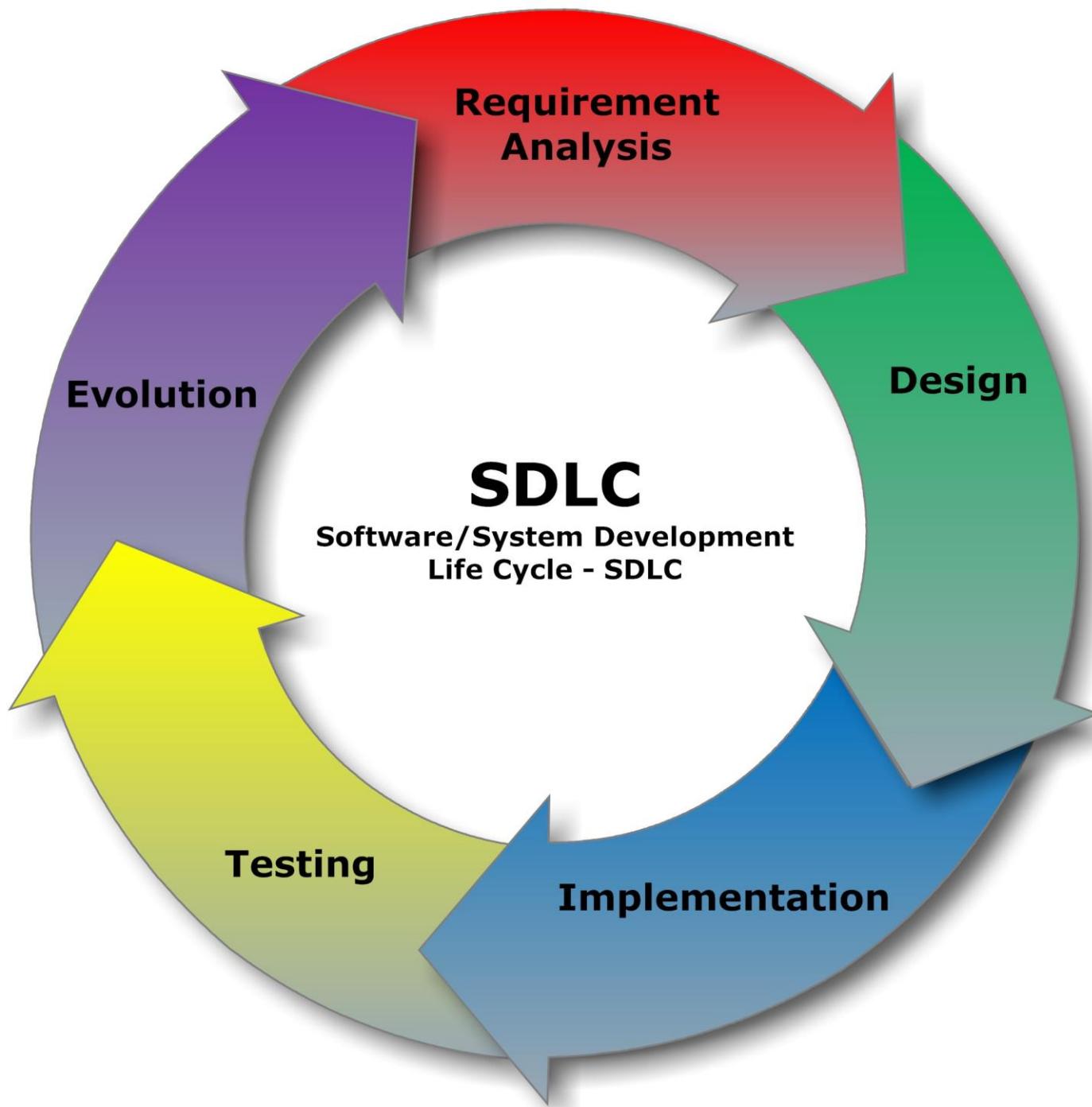


- ◊ More and more, individuals and society rely on advanced software systems. We need to be able to produce reliable and trustworthy systems economically and quickly.
- ◊ It is usually cheaper, in the long run, to use software engineering methods and techniques for software systems rather than just write 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, product managers, 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
  - Unique by company or organization
- ◊ 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 modelling 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

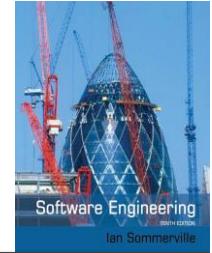


- ◊ Some **fundamental principles** apply to all types of software system, irrespective of the development techniques used:
  - Systems should be developed using a **managed and understood development process**. Of course, different processes are used for different types of software.
  - **Dependability 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.



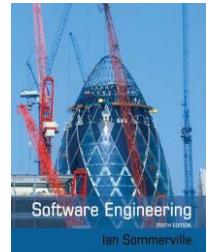
# Ethical dilemmas

- ◊ 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





# 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

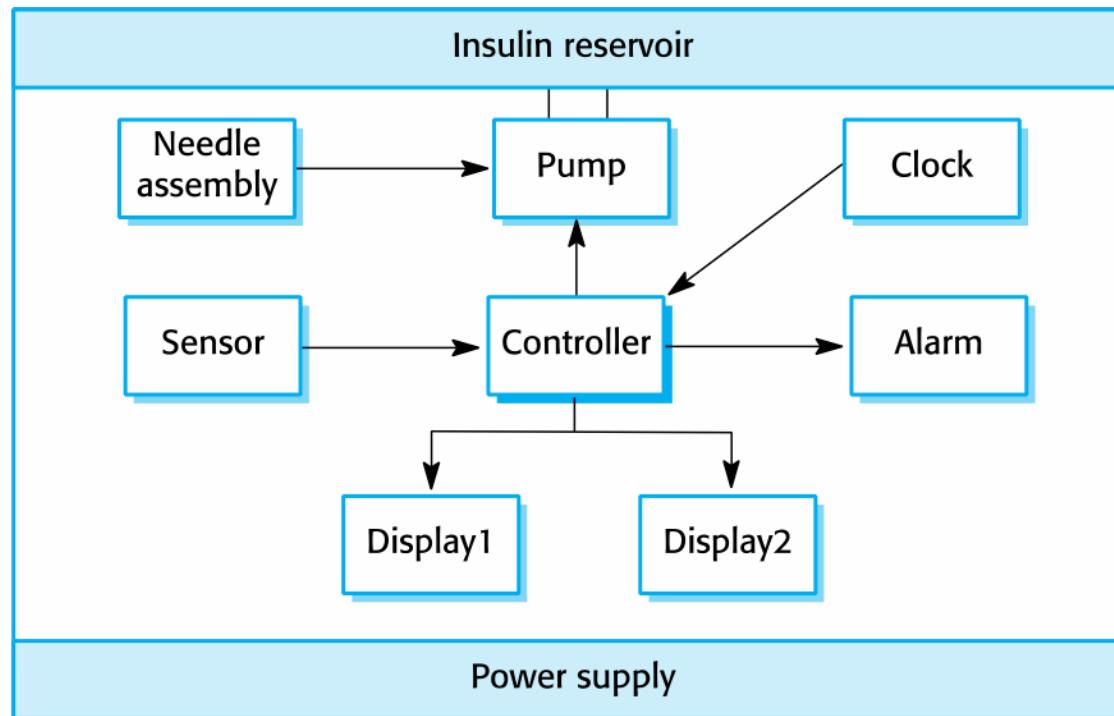


# Insulin pump control system

- ◊ 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.

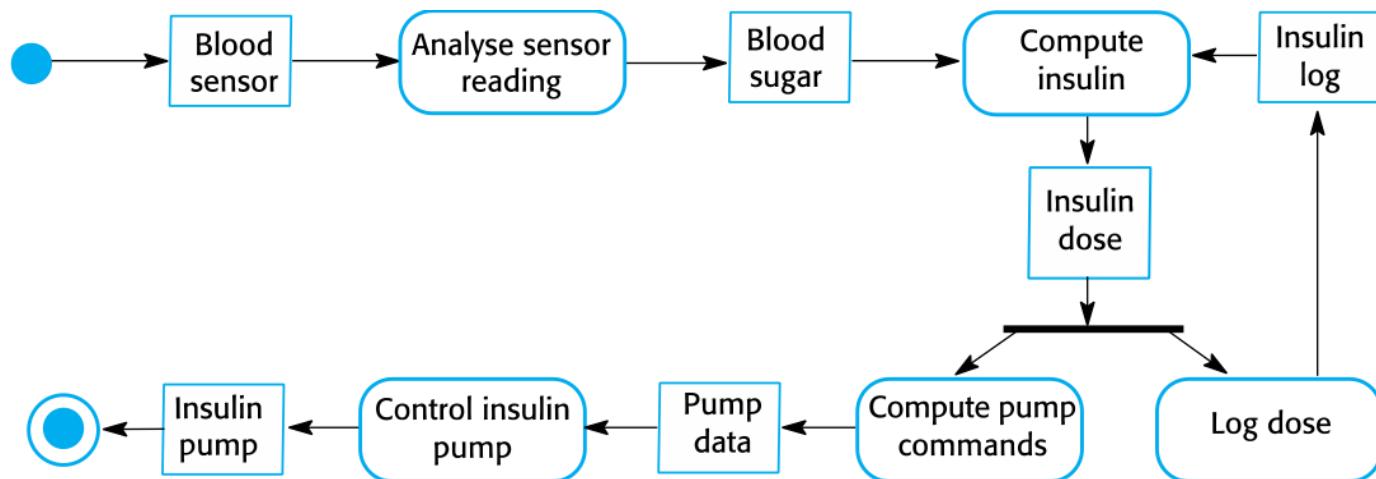


# Insulin pump hardware architecture





# 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



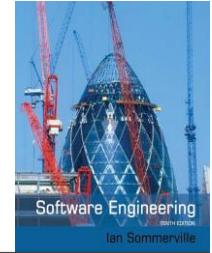
- ◊ A patient information system to support mental health care is a medical information system that **maintains information about patients** suffering from mental health problems and the **treatments** that they have received
- ◊ Most mental health patients do not require dedicated hospital treatment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

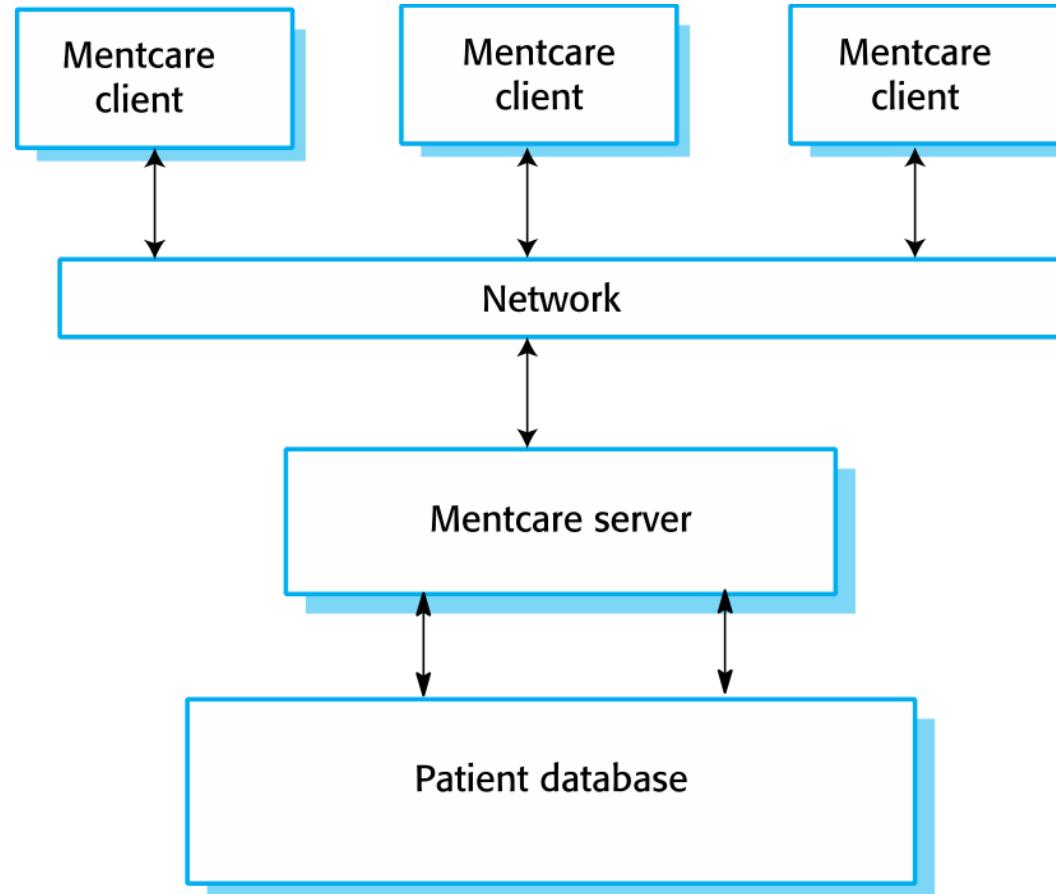


## Mentcare goals

- ◊ 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.



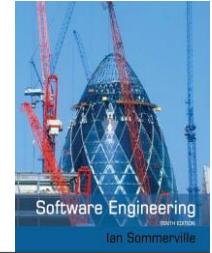
# Mentcare system concerns

## ◊ Privacy

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

## ◊ Safety

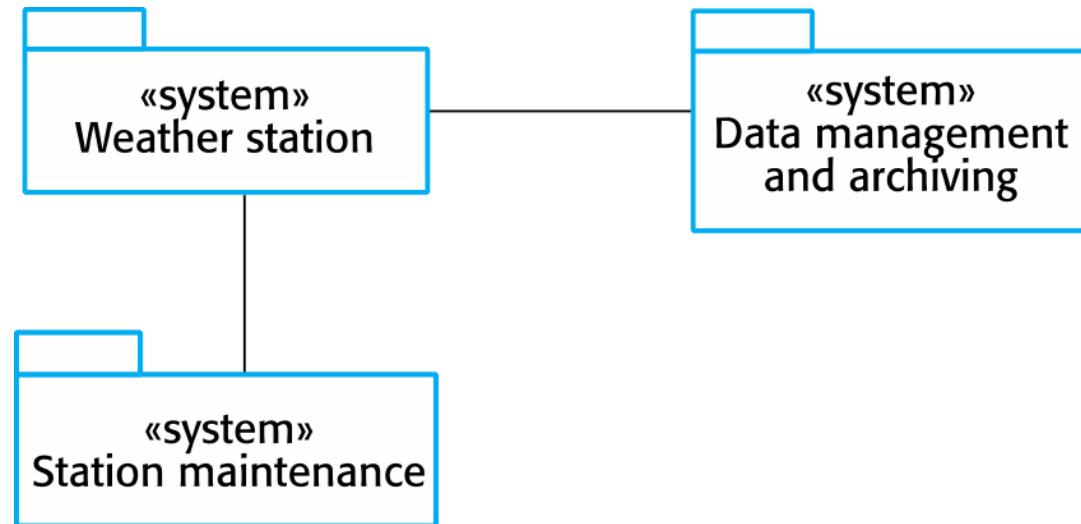
- Some mental illnesses cause patients to become suicidal or a danger to other people. Wherever possible, the system should warn medical staff about potentially suicidal or dangerous patients
- The system must be available when needed otherwise safety may be compromised and it may be impossible to prescribe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



# Wilderness weather station

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

# The weather station's environment





# Weather information system

## ◊ The weather station system

- This is responsible for collecting weather data, carrying out some initial data processing and transmitting it to the data management system

## ◊ The data management and archiving system

- This system collects the data from all of the wilderness weather stations, carries out data processing and analysis and archives the data.

## ◊ The station maintenance system

- This system can communicate by satellite with all wilderness weather stations to monitor the health of these systems and provide reports of problems



# Additional software functionality

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

# 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.

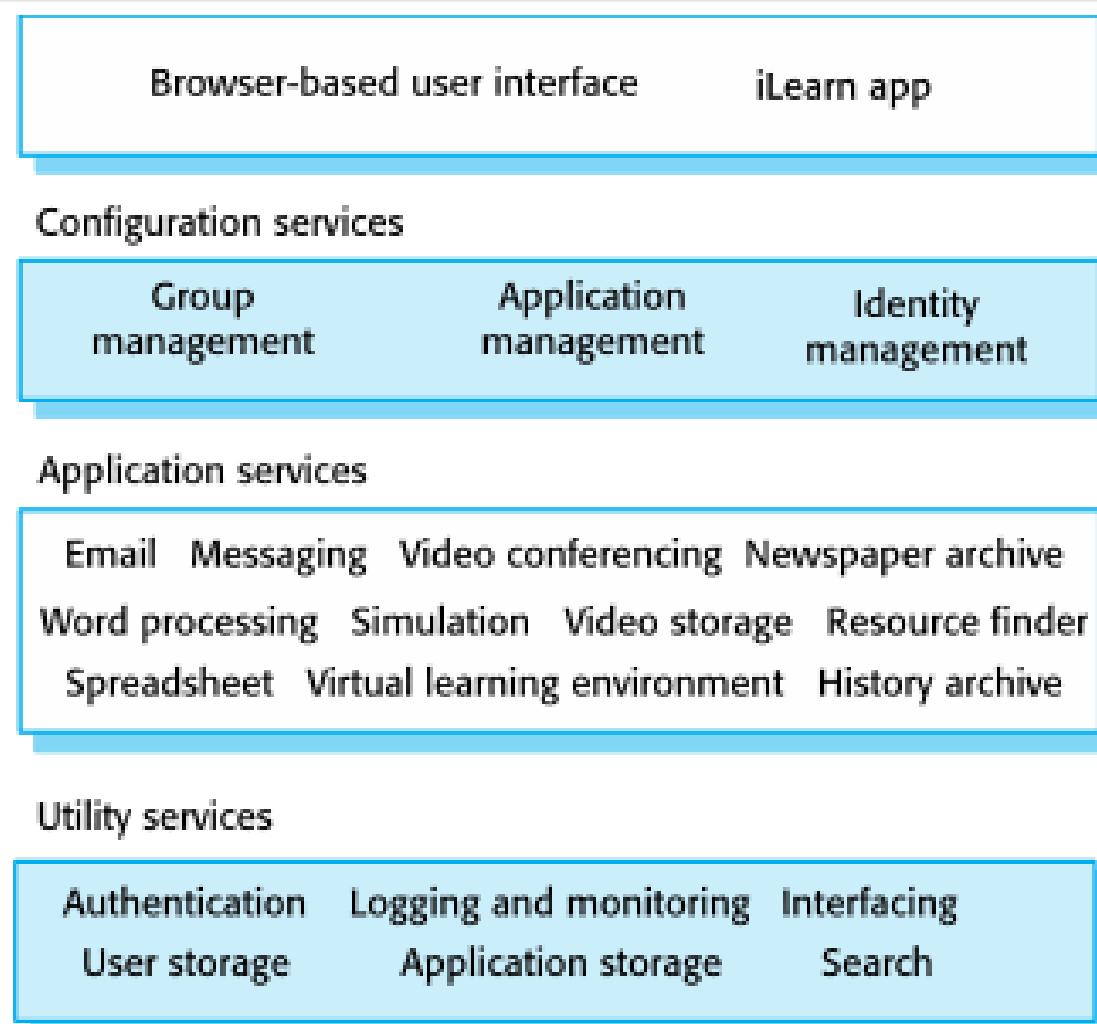
# iLearn services



- ◊ *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 & acceptability
- ◊ The high-level activities of specification, development, validation & 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

# Professional Tip of the Day: Tips for Software Job Interviews



- ◊ Know what the company does and the products they produce
  - Come up with a few questions to ask them about their company or products during the interview
- ◊ Research potential interview questions
- ◊ Know the salary range for the position you are applying for
- ◊ Make sure you can speak about or at least have knowledge about the technologies listed in the job posting
- ◊ Be honest if you don't have experience or knowledge about a topic but be willing to learn or challenge yourself



## Chapter 2 – Software Processes

Ian Sommerville,

*Software Engineering*, 10<sup>th</sup> Edition

Pearson Education, Addison-Wesley

Note: These are a slightly modified version of Chapter 2 slides available from the author's site <http://iansommerville.com/software-engineering-book/>



# Topics covered

---

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



# The software process

- ◊ **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.

Phase	Required	Nice to have
<b>INITIATION</b>		
	Document/Page Containing the following to initiate the Project:	
	Description (3 to 5 bullet Points)	Customer/market target
	Purpose (Goals and objectives)	Assumptions & exclusions
	Success Factors	Business Sponsor
	Key Resources Identified (i.e. Project manager, Dev Lead, QA Lead)	In scope and out of the scope
	Project Kick-off	
<b>PLANNING</b>		
	Requirements / Specs (1-100%)	
	Project Schedule	Key dates & process
	Work Estimates	Risk Analysis Log
	Major Milestones Identified	Project Management Plan
	Dependencies Identified	
<b>EXECUTION</b>		
	Infrastructure (1) - Development	
	<i>Core Development</i>	Tools
		Issue Management Log
	Demonstration(s)	Iterations & tempo
	Status Updates	Executive Sign-Off
<b>TESTING</b>		
	QA Cycle	
	(Risk Analysis Statement)	Customer (UAT)
	Acceptance Criteria and Process Defined	
<b>LAUNCH</b>		
	Code Freeze	
	Infrastructure (2) - PROD	Training
	Customer Training/Communication	Time-frame: begin & end
		Validation & Hot-Fixes
	<i>GO-LIVE</i>	
<b>CLOSURE</b>		
	Release Assessment	Post-mortem
		Next steps
	NSE (Source code escrow)	
	PROJECT COMPLETE	



# Sprint Example

Monthly Sprint Example 16.6 (June)





# 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 (1)

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

## ◊ Incremental development (2)

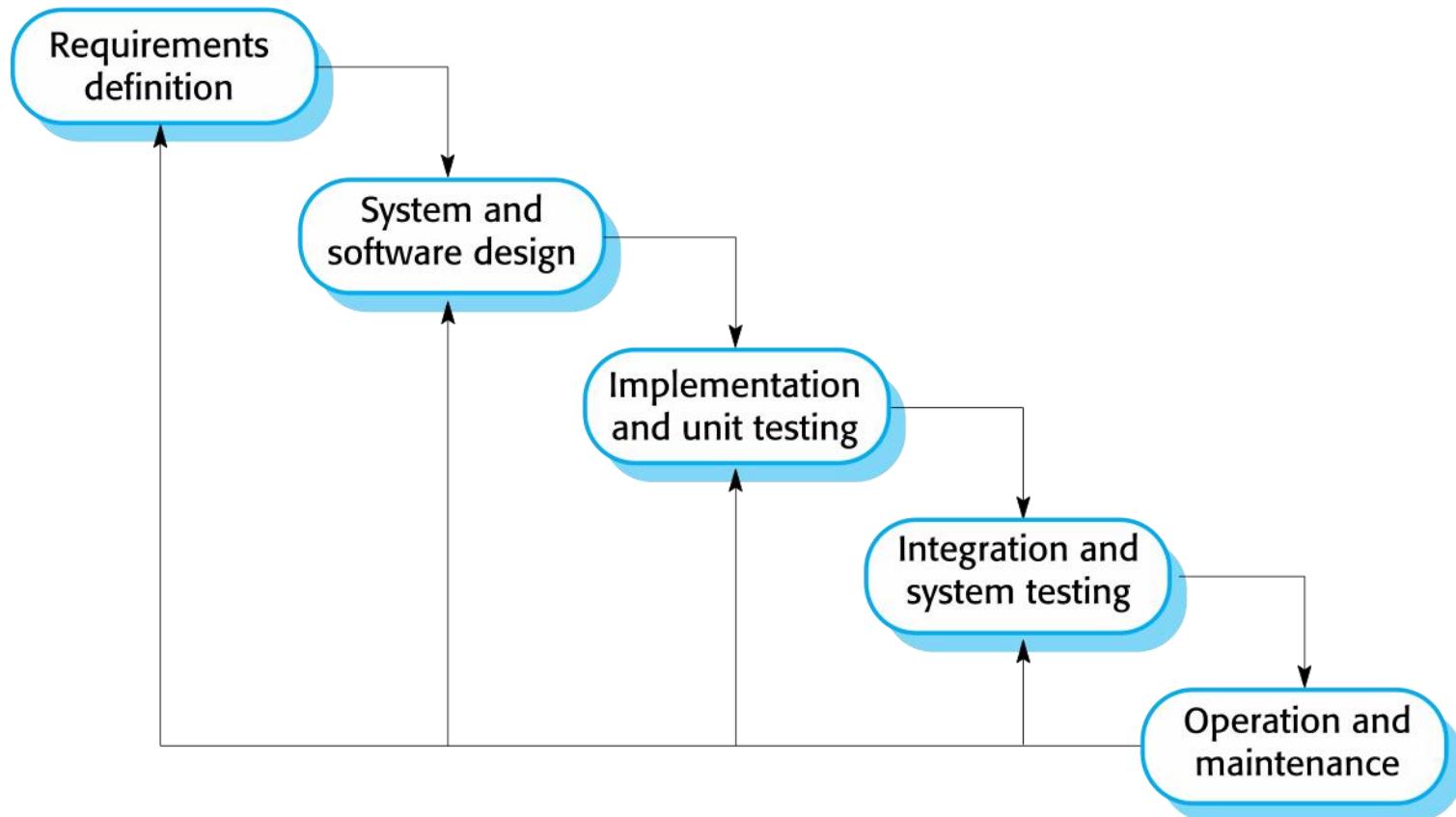
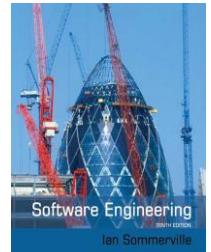
- Specification, development and validation are interleaved. May be plan-driven or agile.

## ◊ Integration and configuration (3)

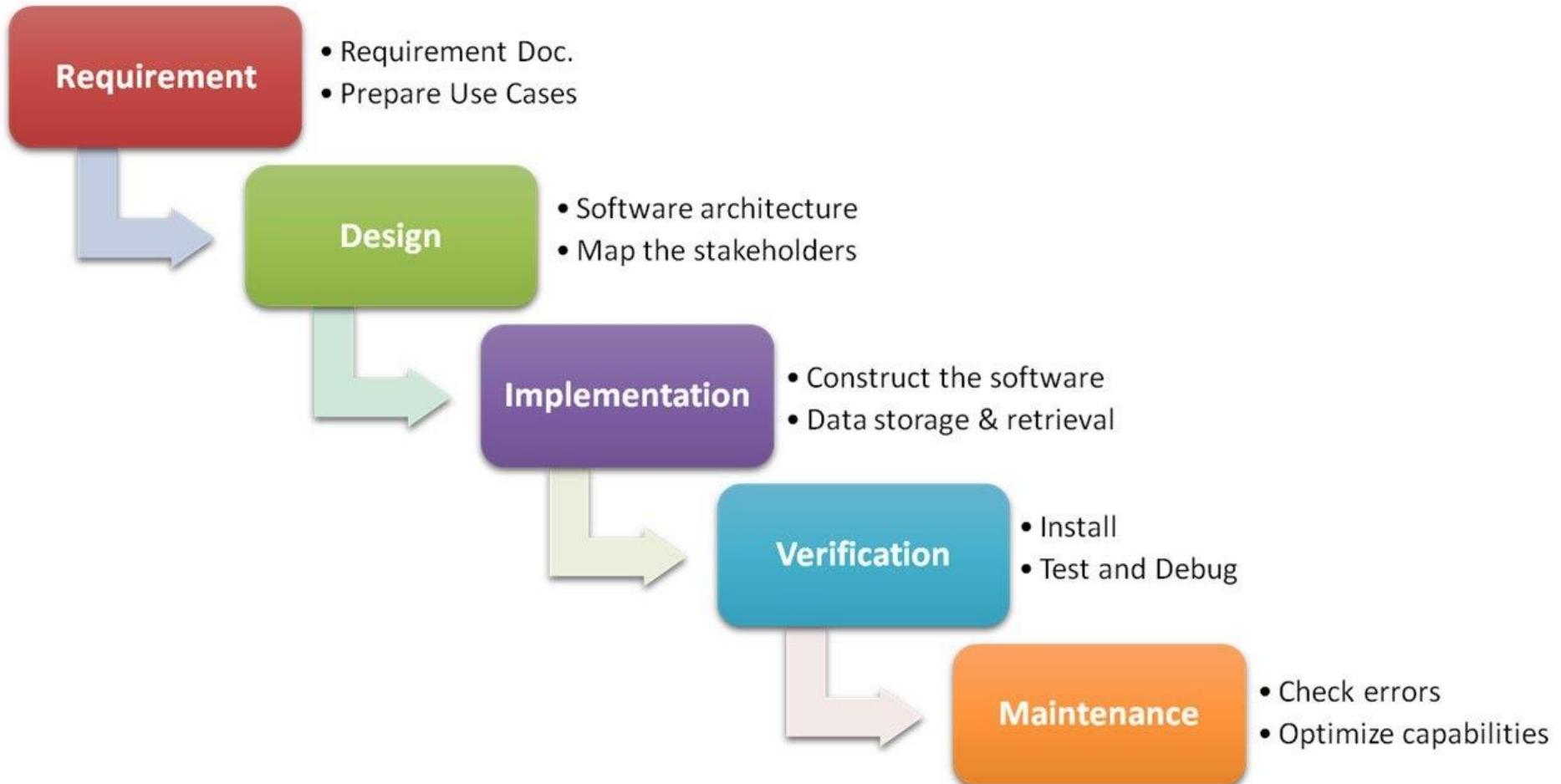
- The system is assembled from existing configurable components. May be plan-driven or agile.

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

# The waterfall model



# The waterfall model Phases

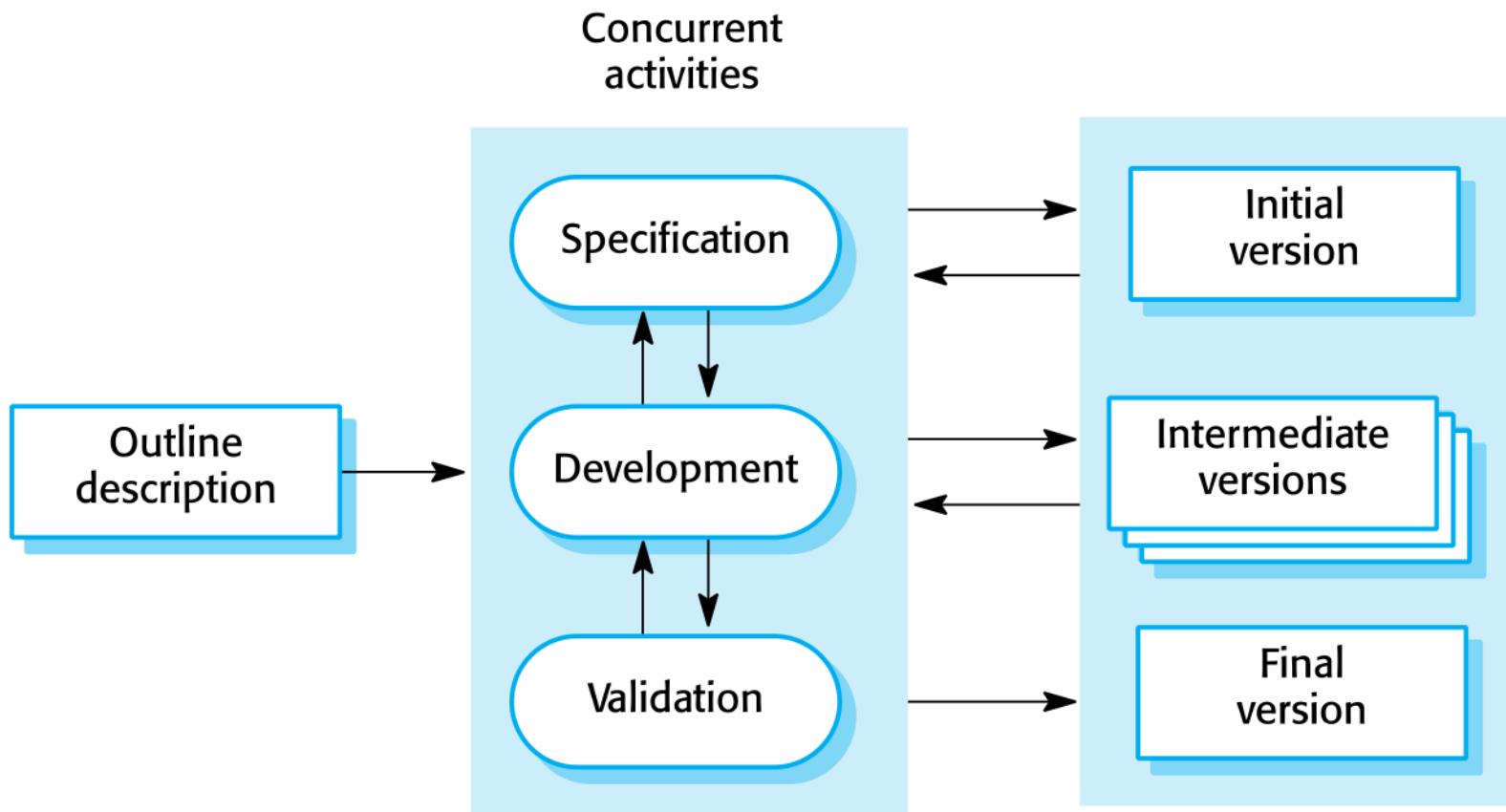
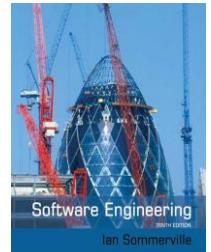




# Waterfall model problems

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

# Incremental development





# Incremental development benefits

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



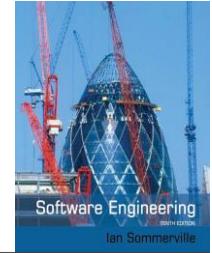
# Incremental development problems

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

# Integration and configuration



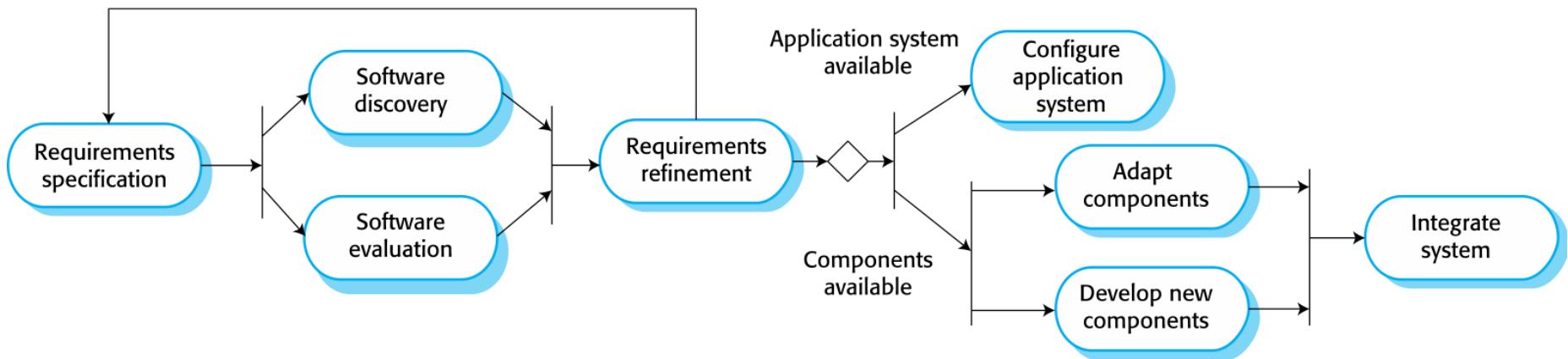
- ◊ Based on software reuse where systems are integrated from existing components or application systems (**COTS** - commercial-off-the-shelf).
- ◊ Reused elements may be configured to adapt their behaviour and functionality to a user's requirements
- ◊ Reuse is now the standard approach for building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



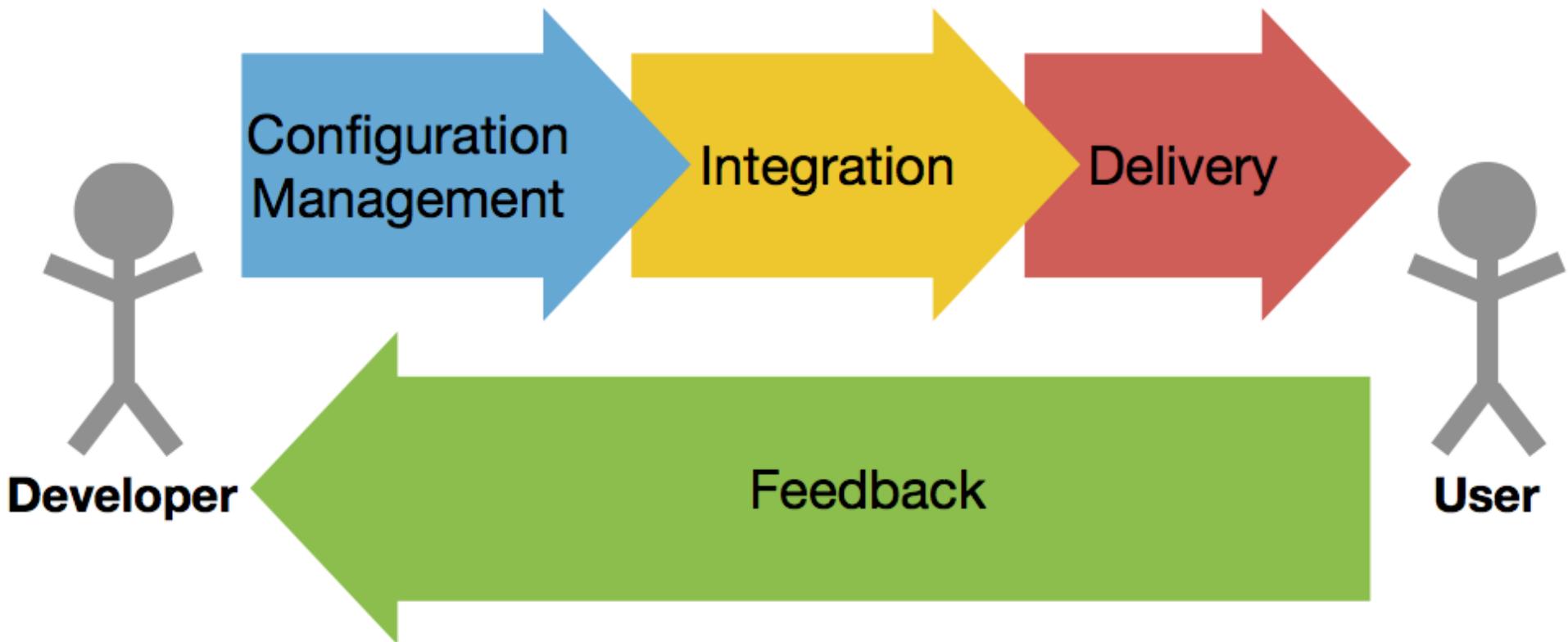
# Types of reusable software

- ◊ 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



# Integration and configuration Simplified





# Key process stages

---

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

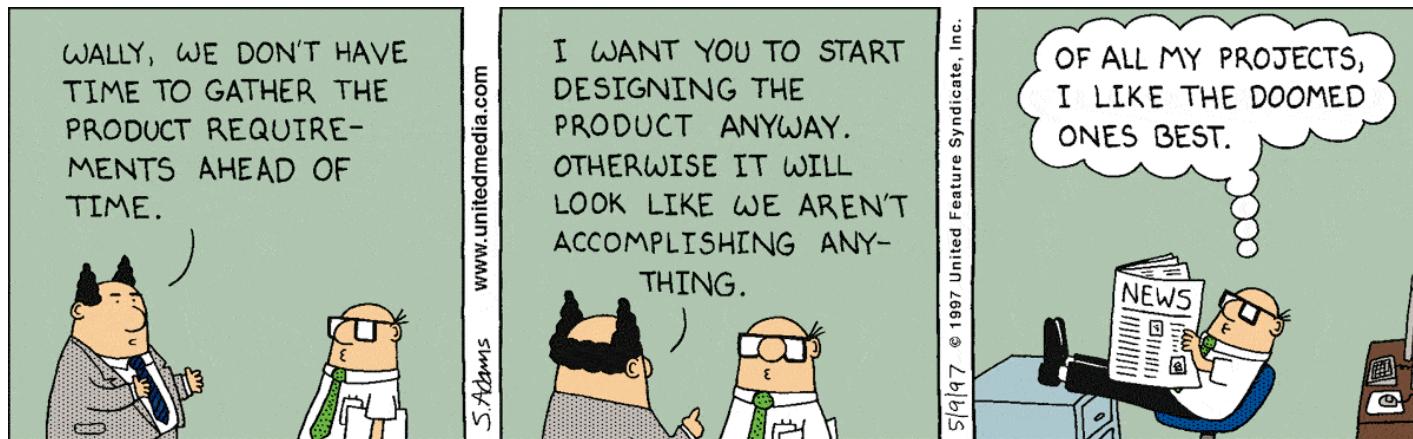


# Advantages and disadvantages

---

- ◊ 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



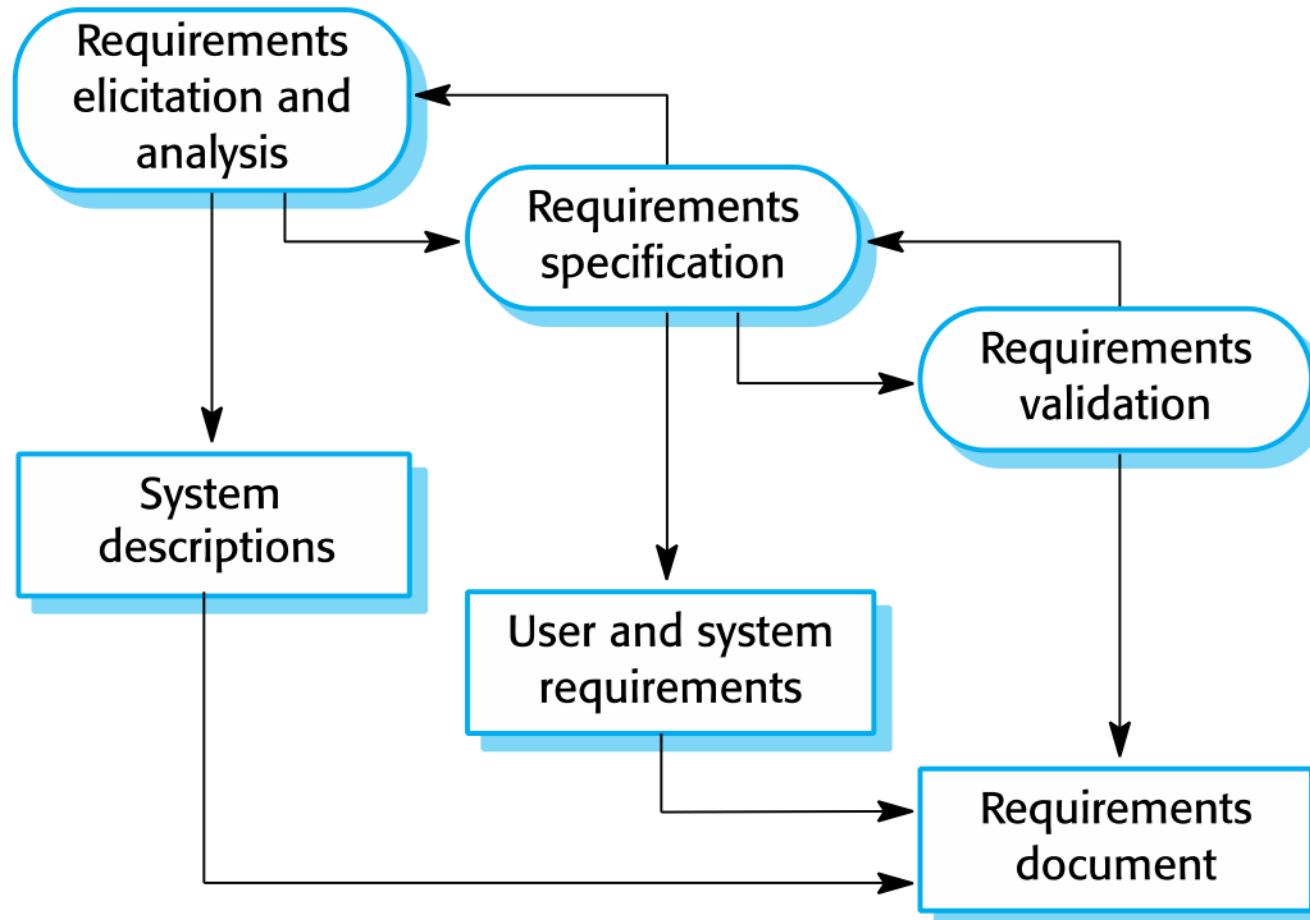


# Process activities



Four Activities of Software Process Framework

# 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

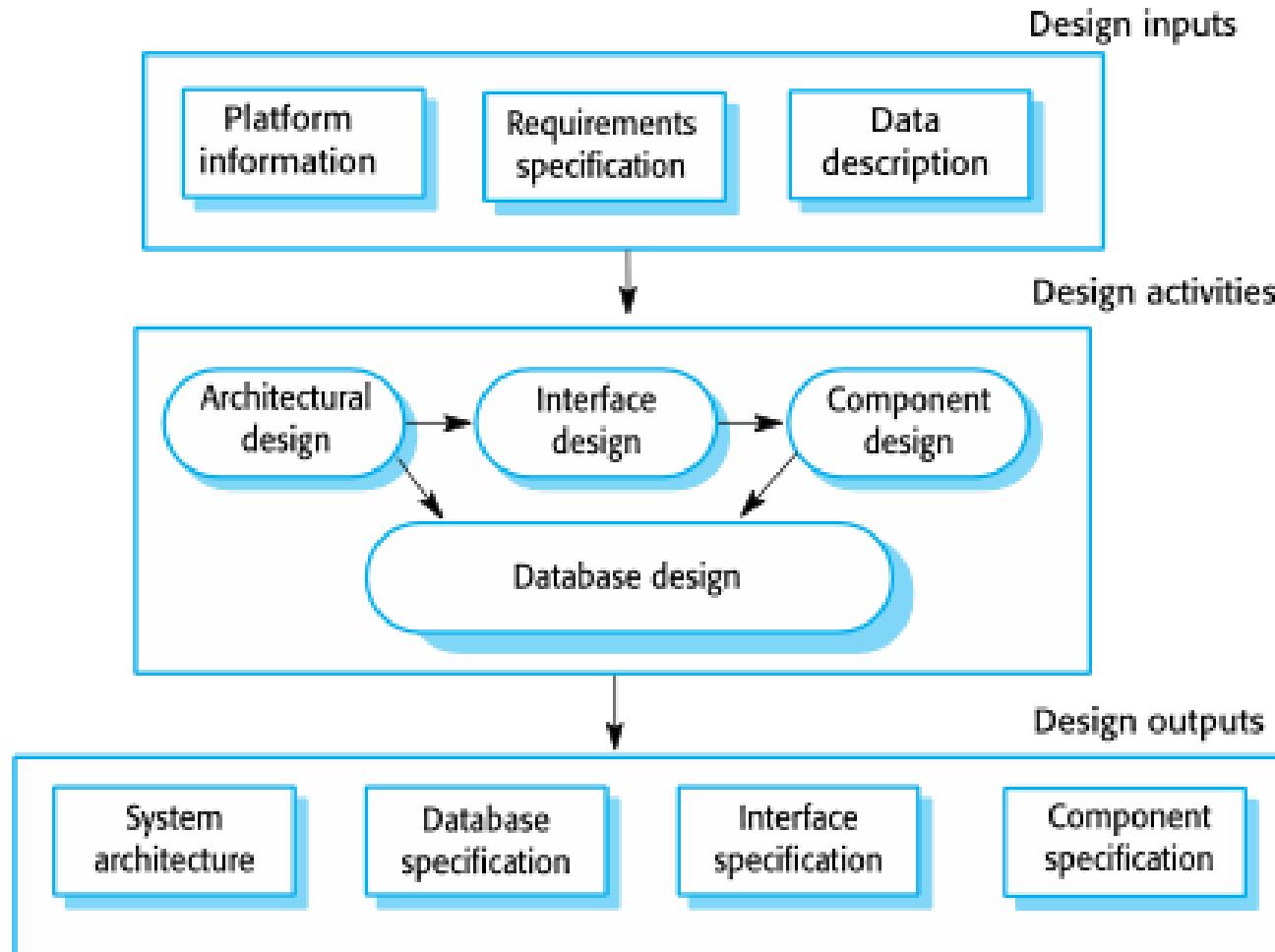
# Software **design** and **implementation**

---



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

# A general model of the design process





# Design activities

- ◊ *Architectural design*, where you identify the overall structure of the system, the principal components (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

---



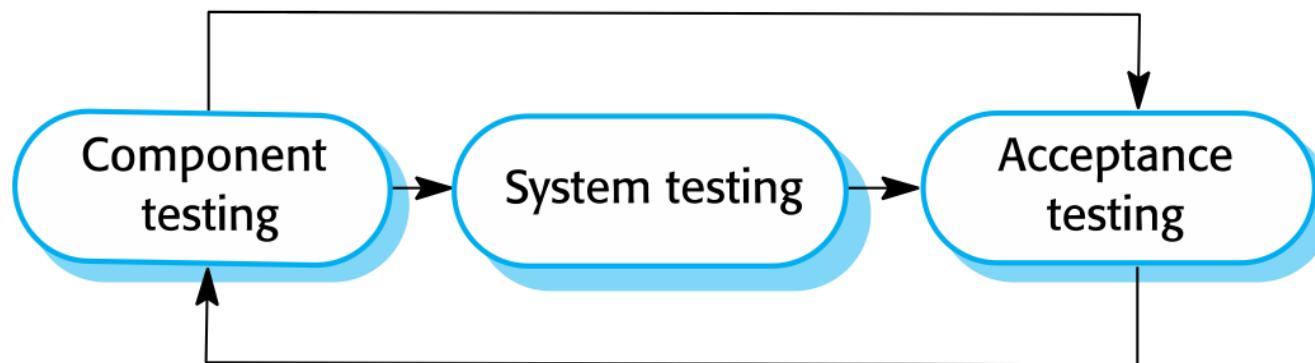
- ◊ The software is implemented either by **developing** a program or programs or by **configuring** an application system
- ◊ **Design and implementation** are interleaved activities for most types of software system
- ◊ **Programming** is an individual activity with no standard process
- ◊ **Debugging** 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

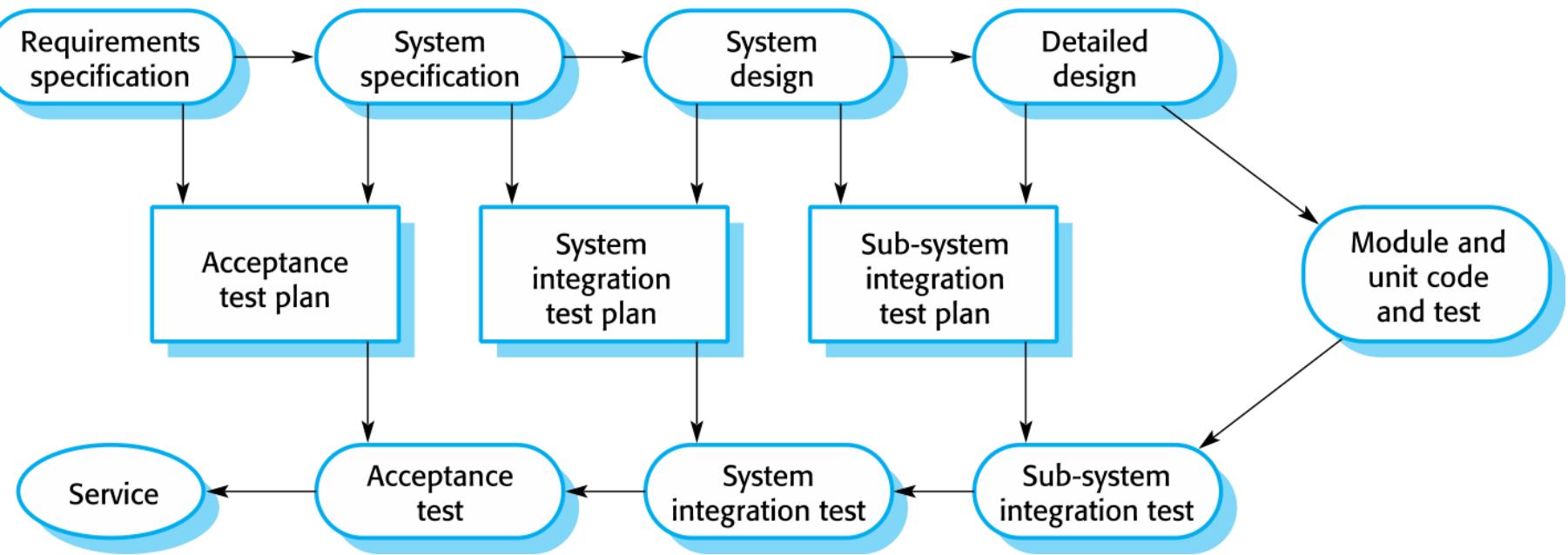
## ◊ System testing

- 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)

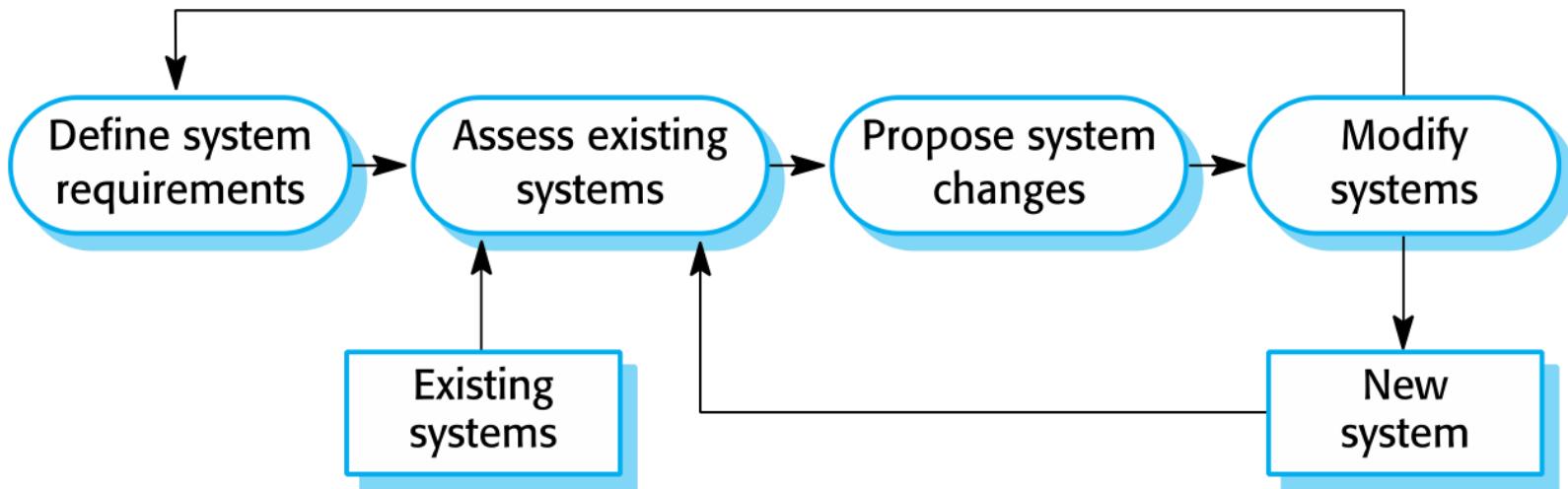
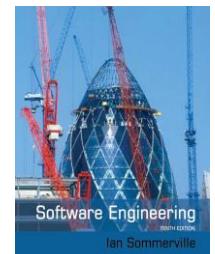


# Software evolution



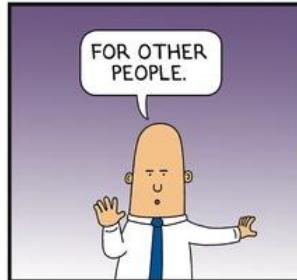
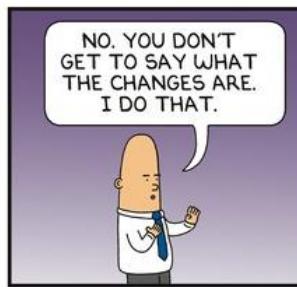
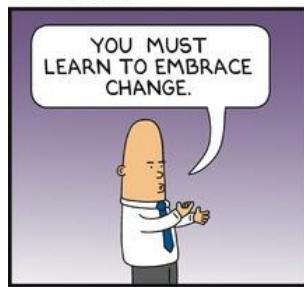
- ◊ Software is inherently flexible and can change
- ◊ As requirements change through changing business circumstances, the software that supports the business must also evolve and change
- ◊ Although there has been a demarcation between development and evolution (maintenance) this is increasingly irrelevant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

# System evolution



## Coping with change

DILBERT



BY SCOTT ADAMS



# 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-analyzing 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 been 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

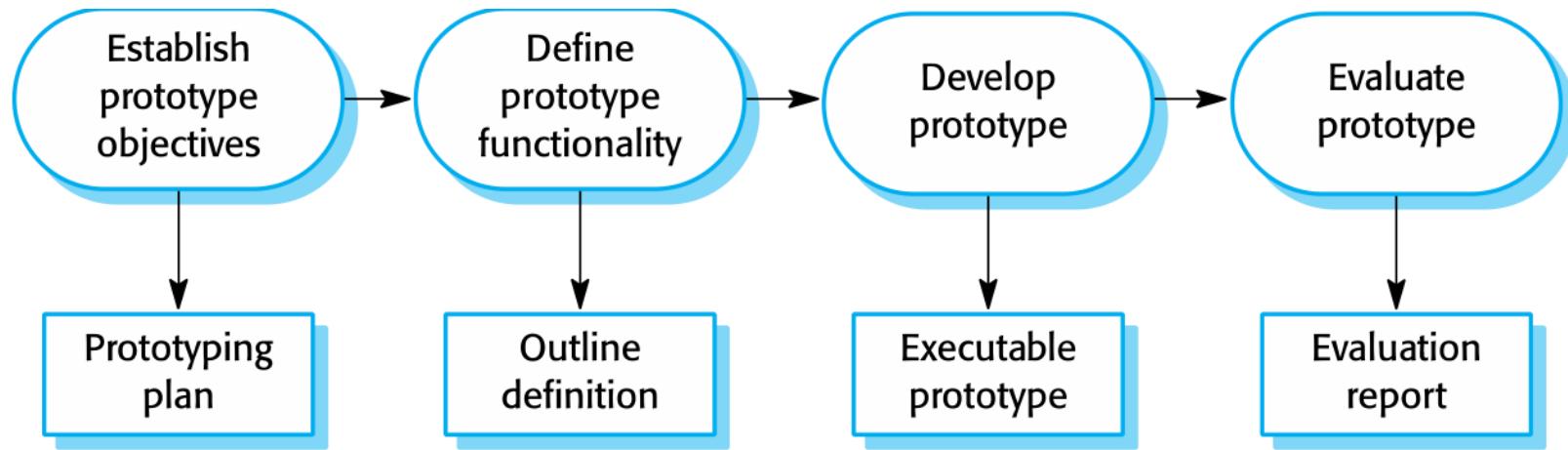


# Benefits of prototyping

---

- ◊ 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 organizational quality standards



## Incremental delivery

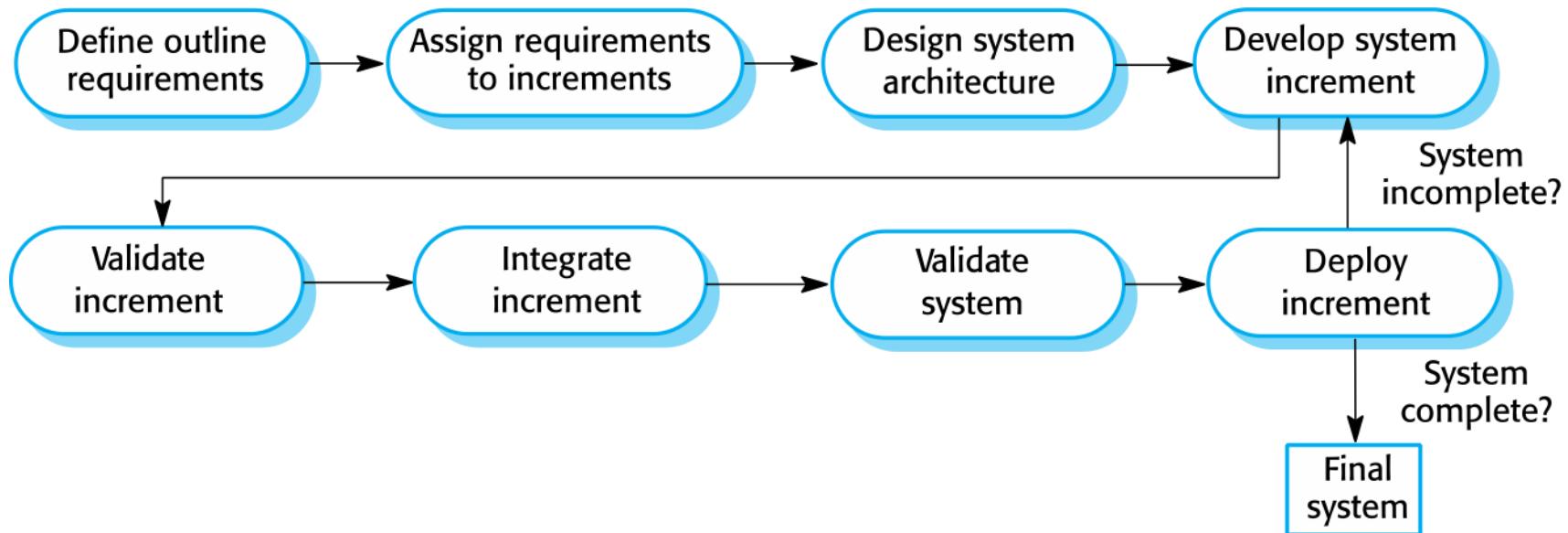
- ◊ Rather than deliver the system as a single delivery, the development and delivery is broken down into increments with each increment delivering part of the required functionality
- ◊ User requirements are prioritized and the highest priority requirements are included in early increments
- ◊ Once 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 (see earlier Software Process Model #3 – slides 10 and 14-16)
  - Develop the system in versions
  - Can work on parallel versions
  - Improved user/customer involvement (as compared with waterfall)
- ◊ Incremental delivery (considered Software Process Model #4)
  - 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
  - Used in agile development

# Incremental delivery





# Incremental delivery advantages

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

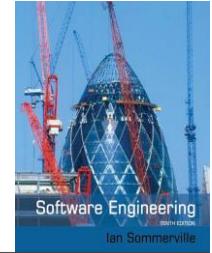


# Incremental delivery problems

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

## 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

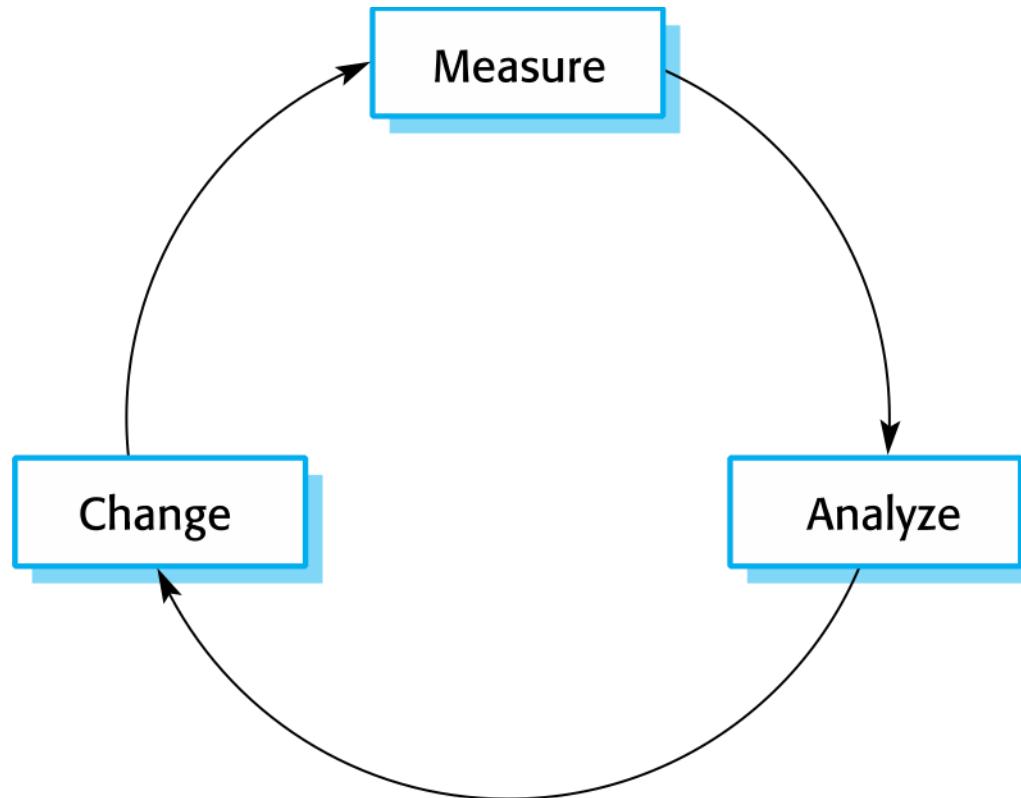


# Approaches to improvement

- ◊ 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.

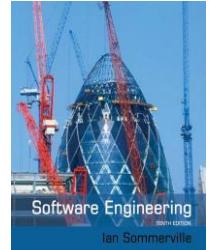
## ◊ *Process change*

- 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 organizations 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.

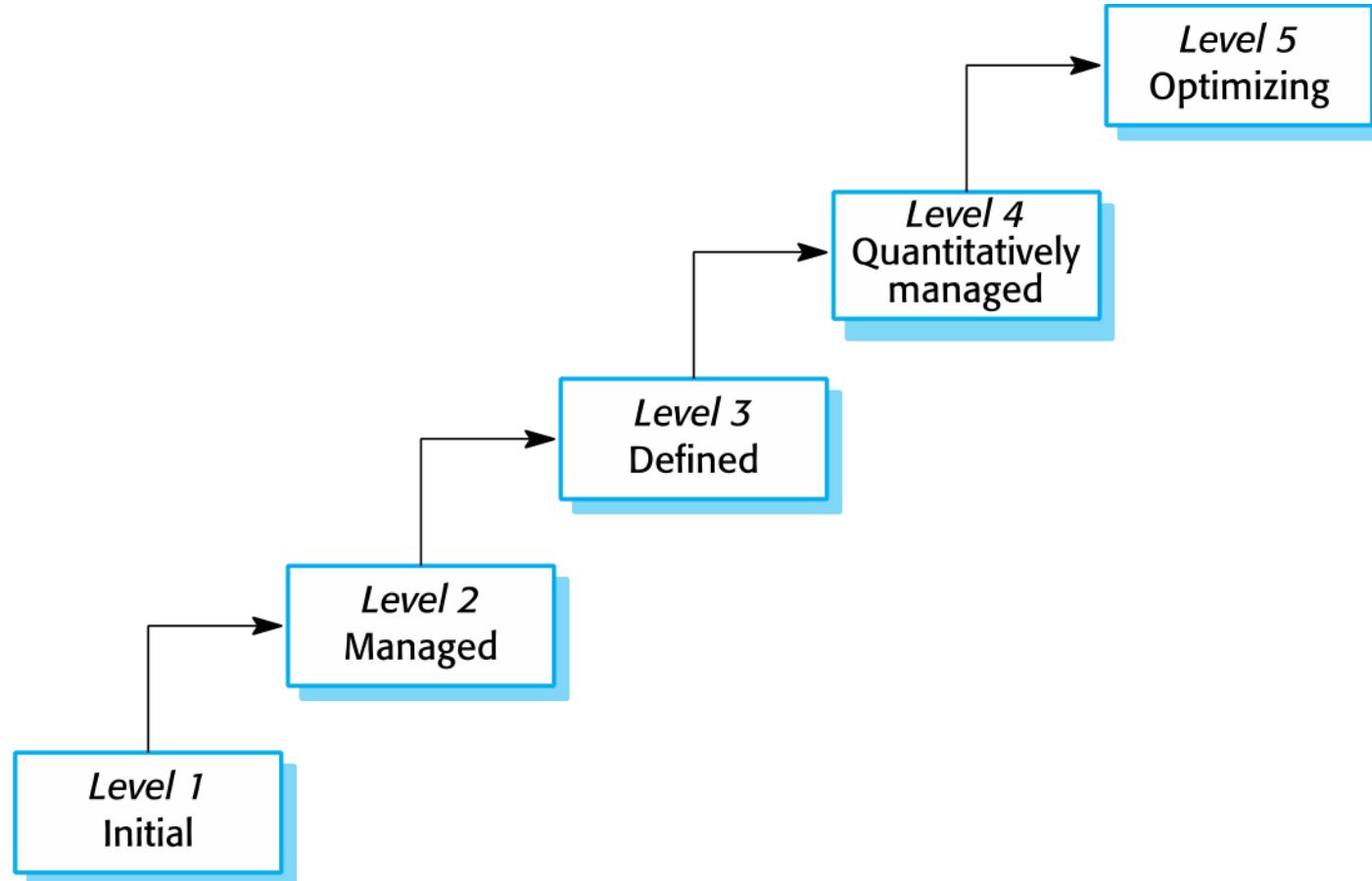


# Process metrics

- ◊ **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
- ◊ Optimizing
  - 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**, **reuse-oriented development**, and **incremental delivery**
- ◊ **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

Ian Sommerville,

*Software Engineering*, 10<sup>th</sup> Edition

Pearson Education, Addison-Wesley

Note: These are a slightly modified version of Chapter 3 slides available from the author's site <http://iansommerville.com/software-engineering-book/>



# 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 environment 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; their aim was to radically reduce the delivery time for working software systems



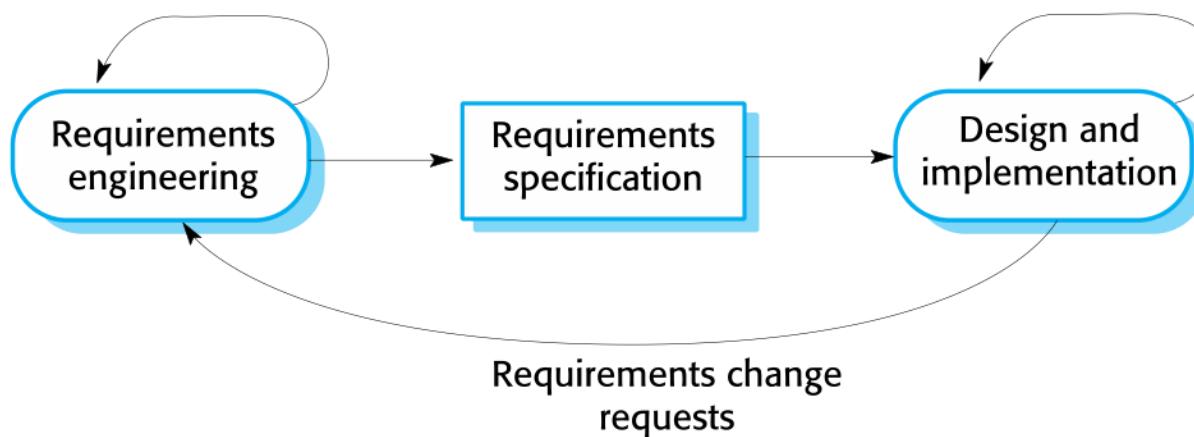
# Agile development

- ◊ 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



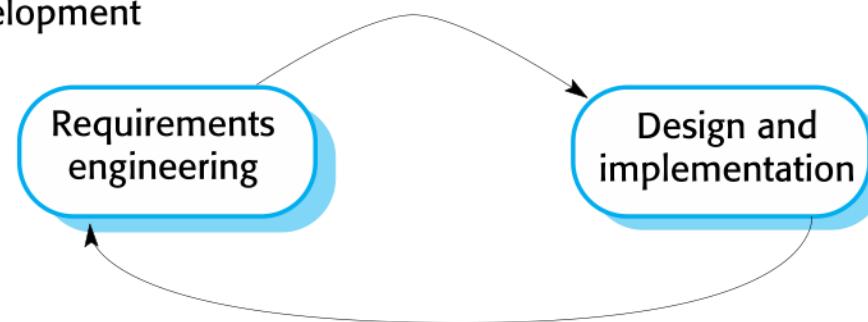
# Plan-driven and agile development

Plan-based development



Requirements change  
requests

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 also possible
- Iteration occurs within activities

## ◊ Agile development

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

# Agile methods

**DILBERT**



**BY SCOTT ADAMS**





# 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 rapidly 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



# The 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



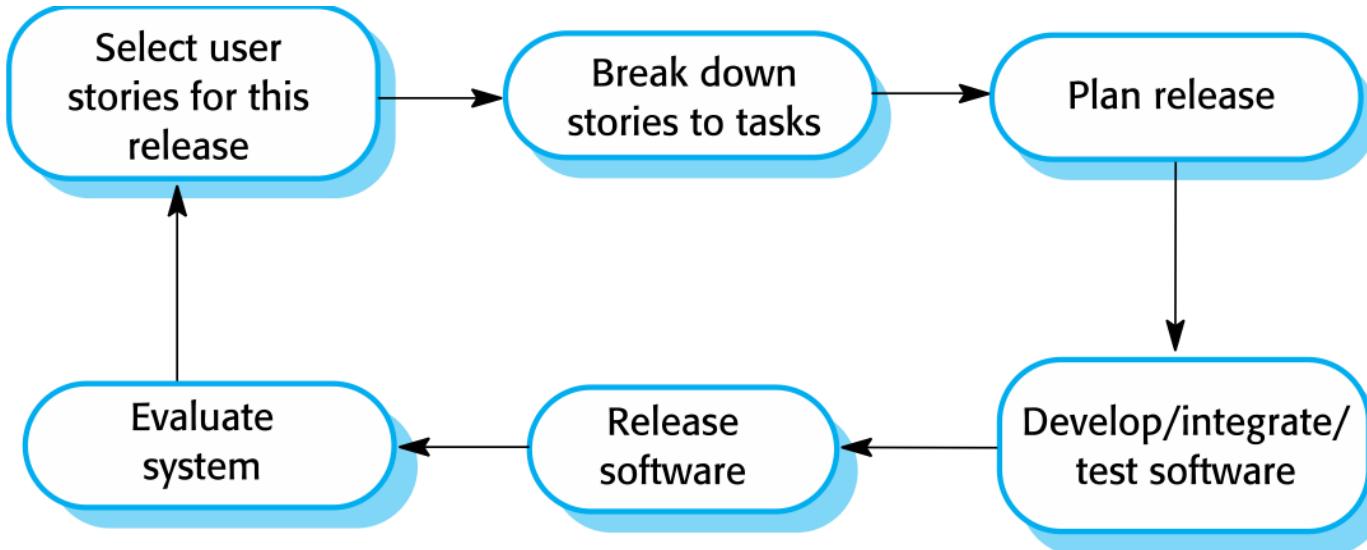


# Extreme programming

- ◊ 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



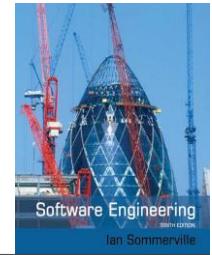
# The extreme programming release cycle





# 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



## 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.

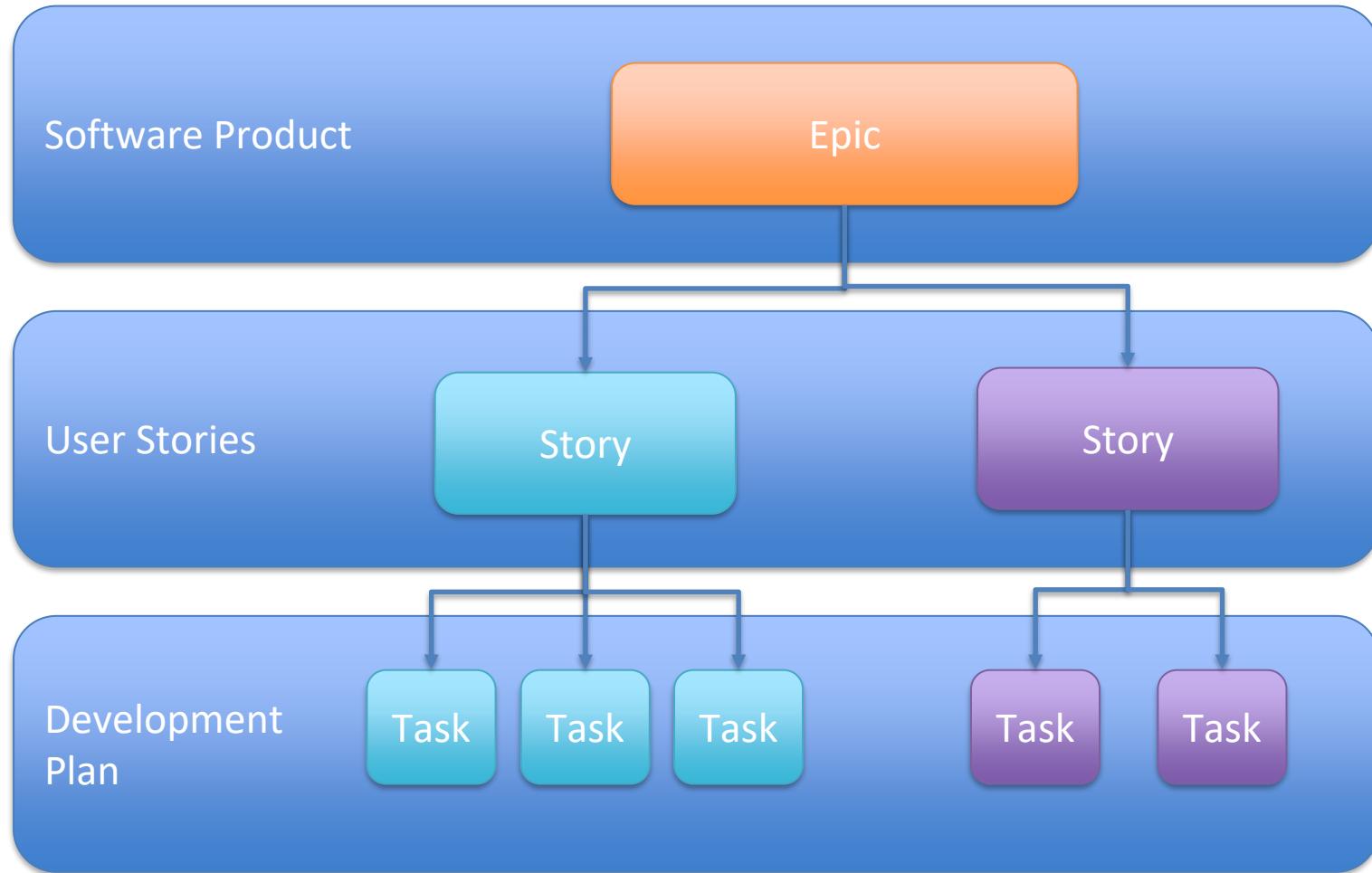


# Sample User Story

- ◊ As a *<type of user>*,  
*I want <to perform some task>*  
*so that I can <achieve some goal/benefit/value>.*
  
- ◊ As a *Amazon Customer*,  
*I want to be able to search for a product*  
*So that I can purchase the product I need.*

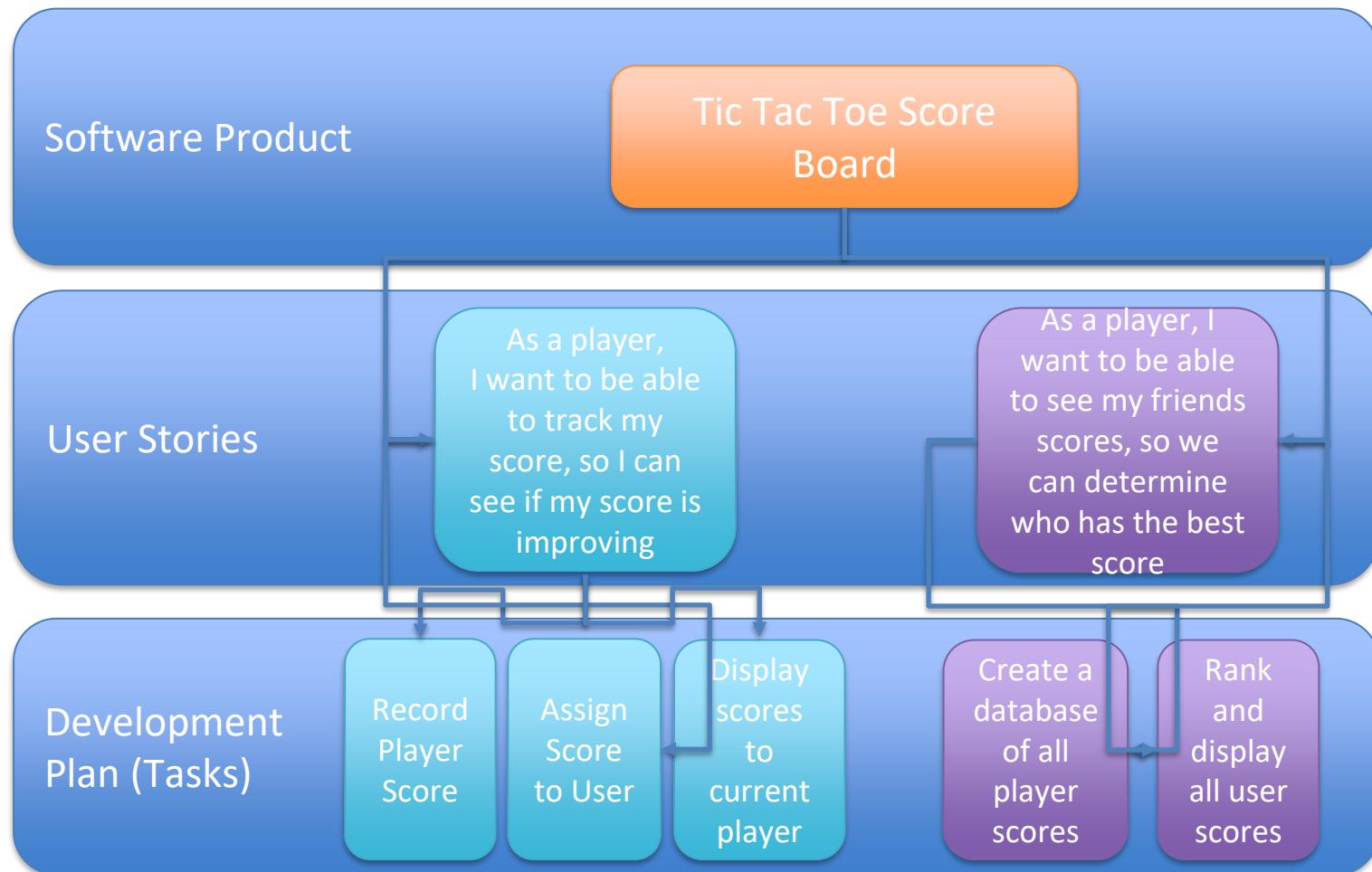


# Structure of Stories and Tasks





# Structure of Stories and Tasks





# 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



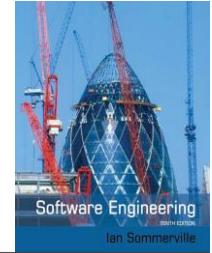
## Examples of refactoring

- ◊ 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, 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.
- ◊ 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.

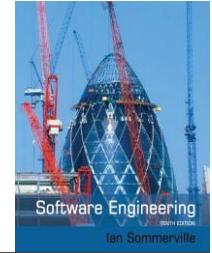
### Output:

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



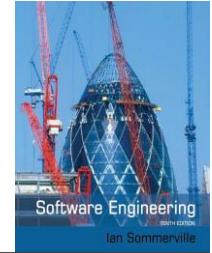
# Test automation

- ◊ 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 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 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 one 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



# Agile Vs Scrum

---

- ◊ Agile is a methodology based on iterative development
- ◊ Scrum is a type of agile approach that is used in software development

## For your Resume:

Familiar with Agile methodologies and working in a Scrum development environment

## Do Not Put:

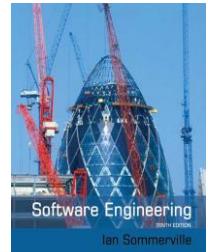
Familiar with Scrum methodologies



# 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's **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

# Intro to SCURM in Under 10 Minutes



A screenshot of a video player interface. The main title is "Scrum in 10 minutes". Below it, a subtitle reads: "Great for introduction to the Scrum process or for a quick refresher." To the left of the main video frame, there are four smaller thumbnail images showing various Scrum-related scenes: a person working at a desk, a person standing in a field, a group of people in a meeting, and a person sitting at a desk. On the right side, there are two more thumbnail images: one showing a person in a field and another showing a group of people in a meeting. The overall background is dark.



Software Engineering  
Ian Sommerville

# Scrum terminology (a)

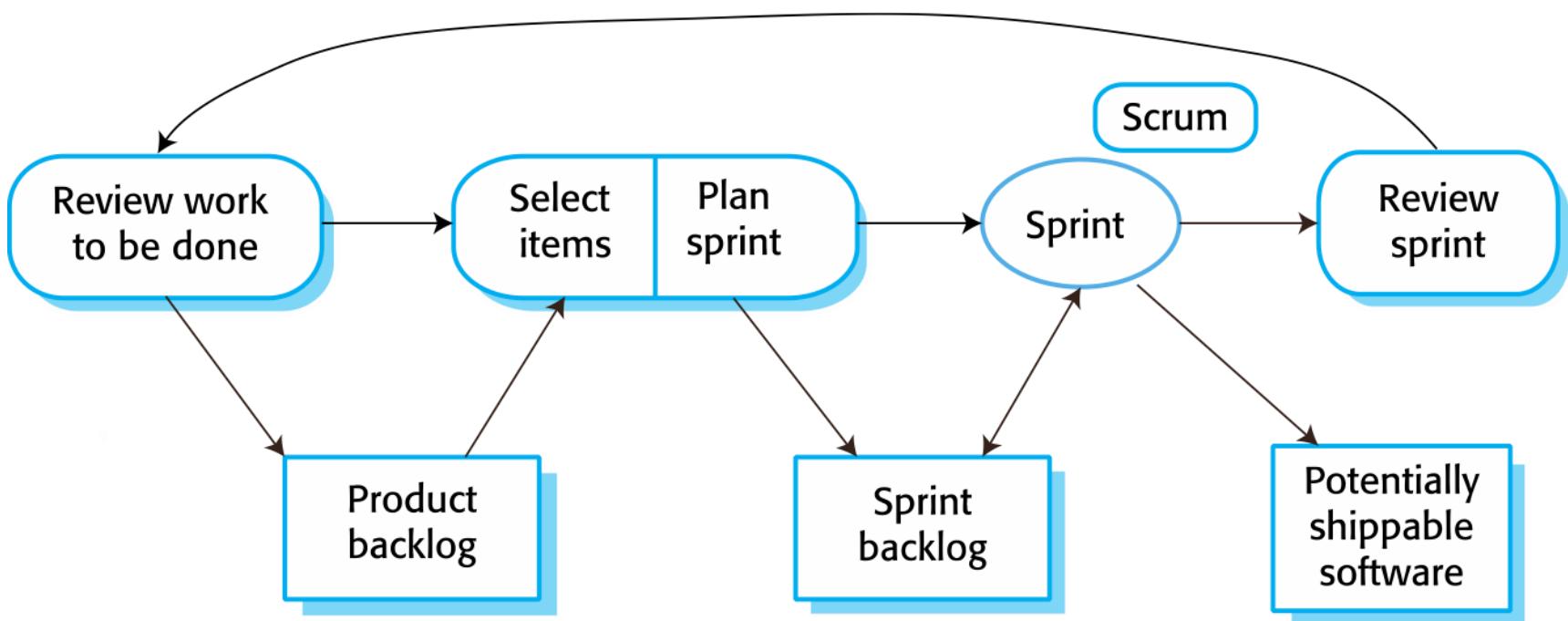
Scrum term	Definition
Development team	A self-organizing group of software developers, which should be no more than 7 people. They are responsible for developing the software and other essential project documents.
Potentially shippable product increment	The software increment that is delivered from a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



# Scrum terminology (b)

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

# Scrum sprint cycle



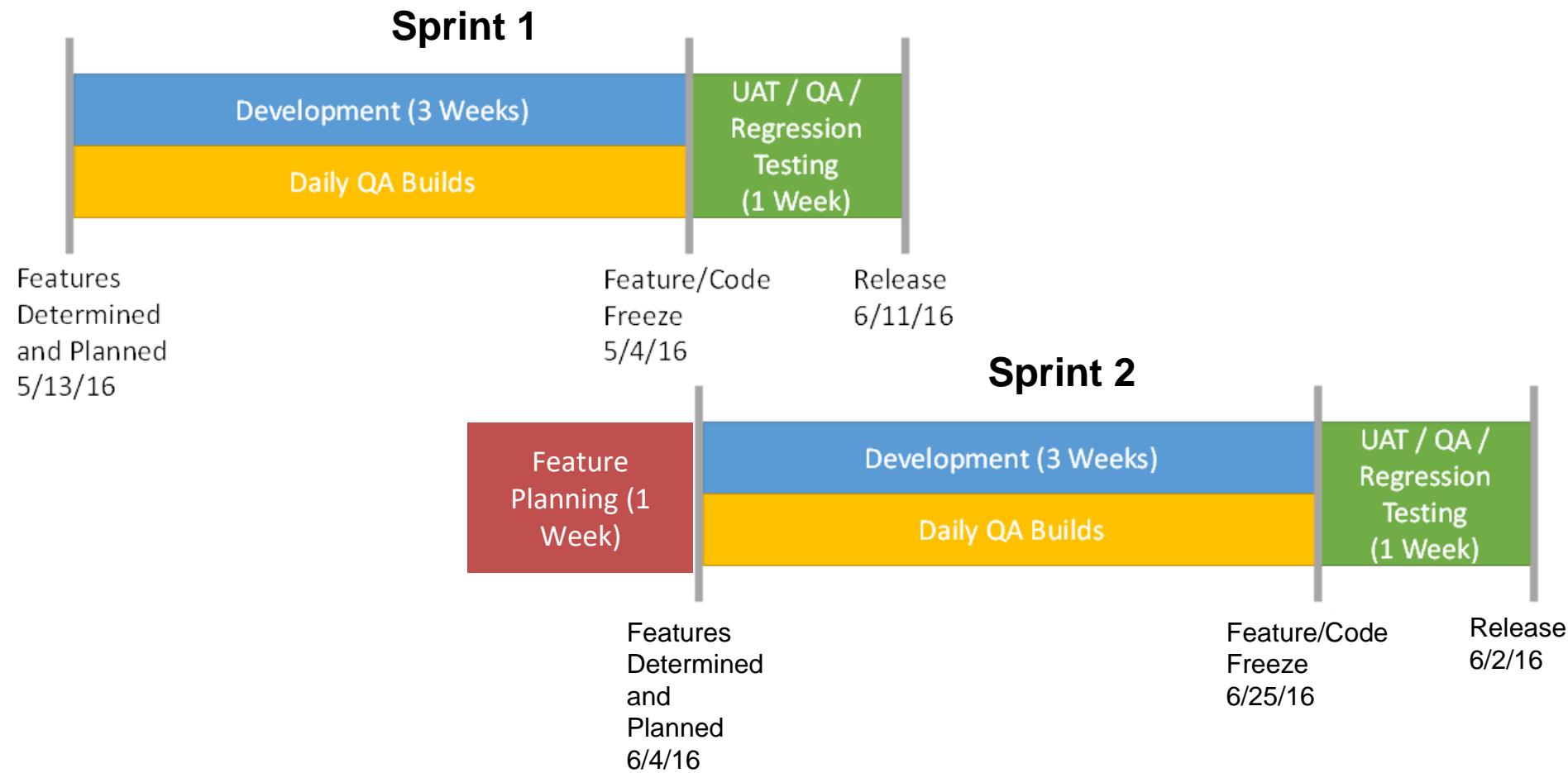


# The Scrum sprint cycle

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



# Examples of a SCRUM Sprint Cycle





# The Sprint cycle

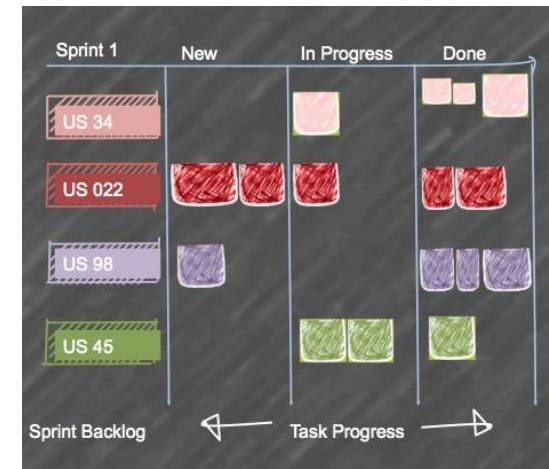
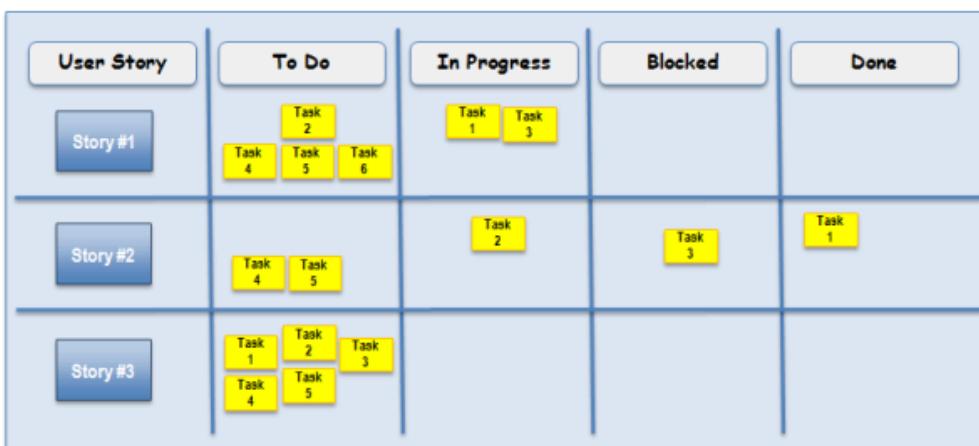
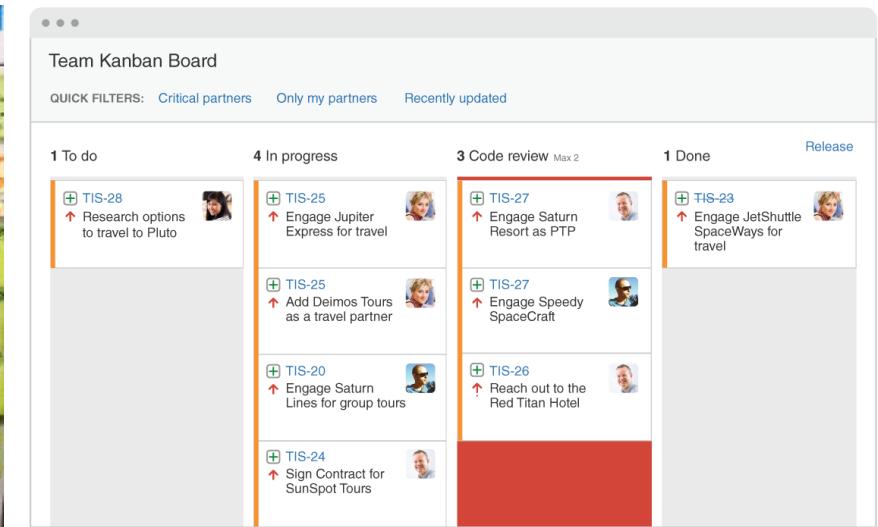
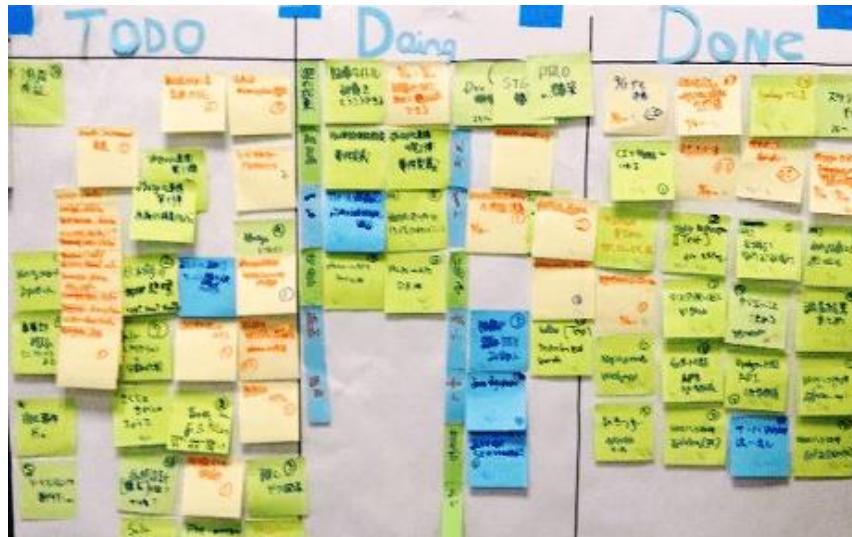
- ◊ Once these are agreed, the team organize themselves to develop the software
- ◊ During this stage the team is isolated from the customer and the organization, with all communications channelled through the '**Scrum master**'
- ◊ The role of the Scrum master is to protect the development team from 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 Tracking

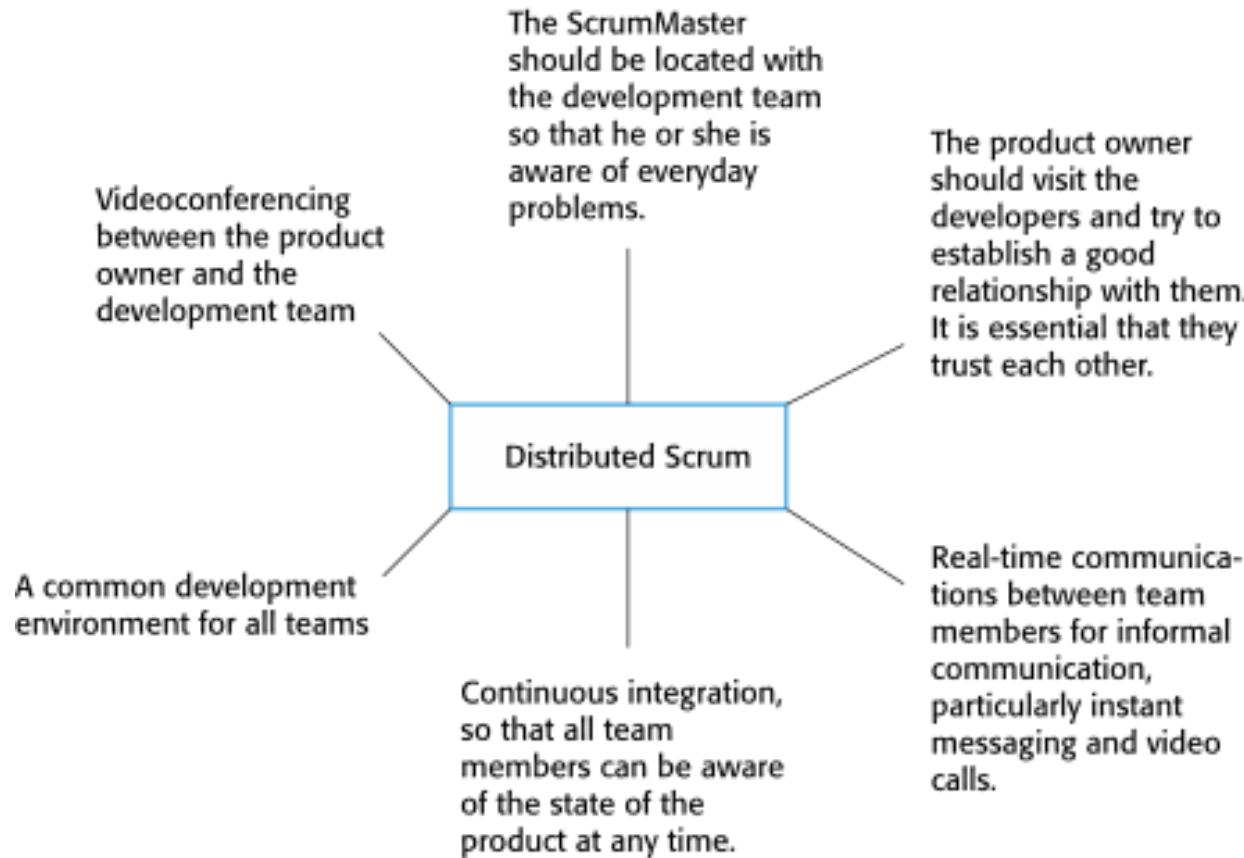




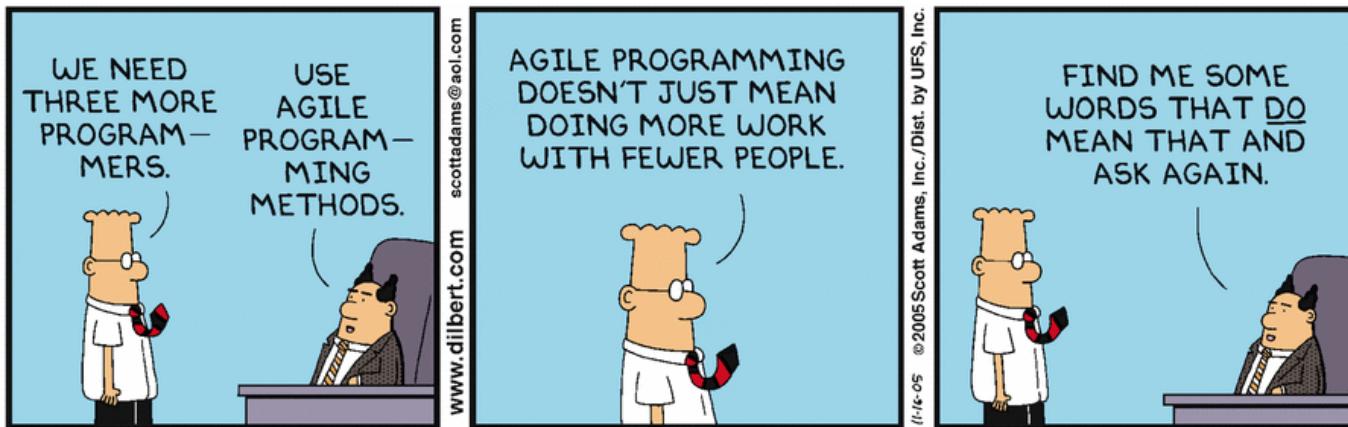
## Scrum benefits

- ◊ 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 by many legal departments because what has to be delivered cannot be guaranteed

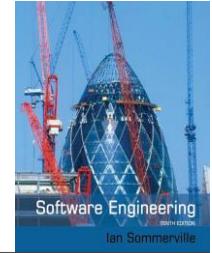
# Agile methods and software maintenance



- ◊ 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



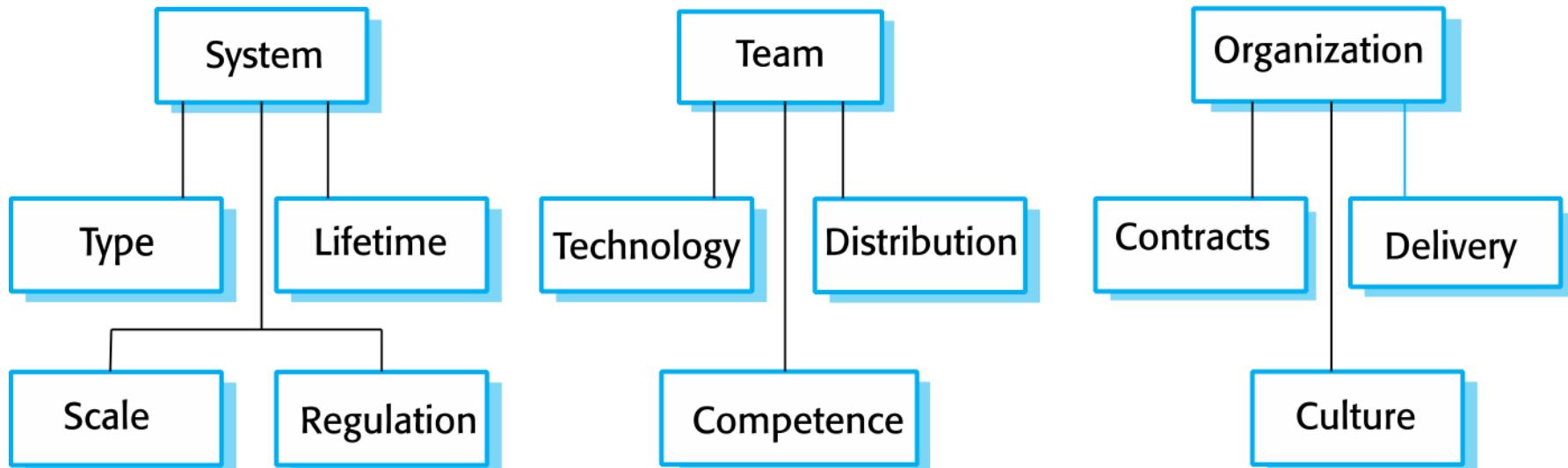
Principle	Practice
Customer involvement	<p>This depends on having a customer who is willing and able to spend time with the development team and who can represent all system stakeholders. Often, customer representatives have other demands on their time and cannot play a full part in the software development.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



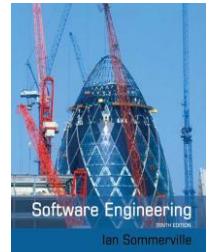
# People and teams

- ◊ 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 visualization and program analysis is essential if design documentation is not available

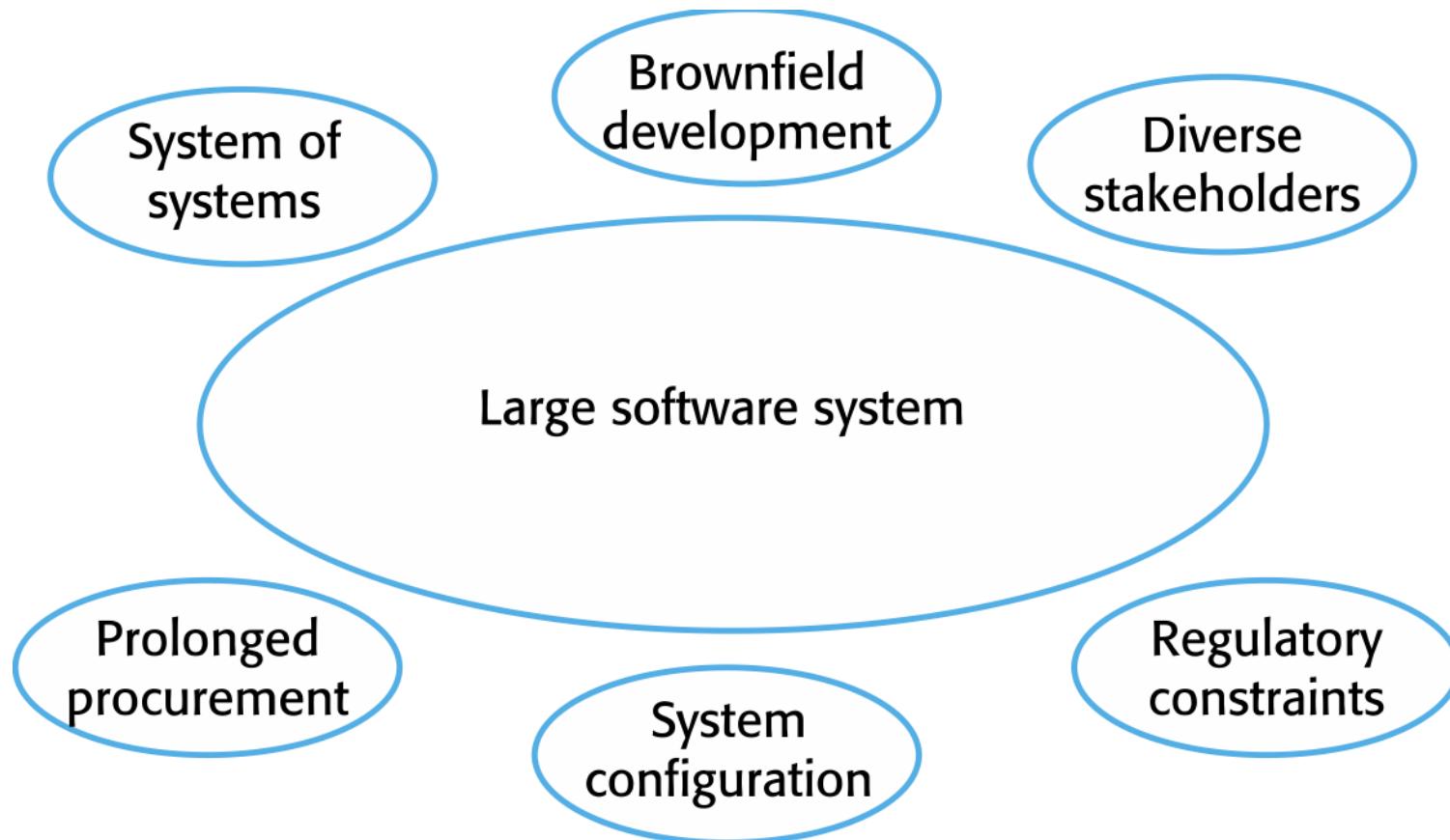


# Organizational issues

- ◊ 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?



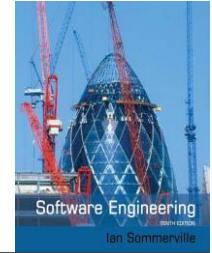
# Factors in large systems





# Agile methods for large systems

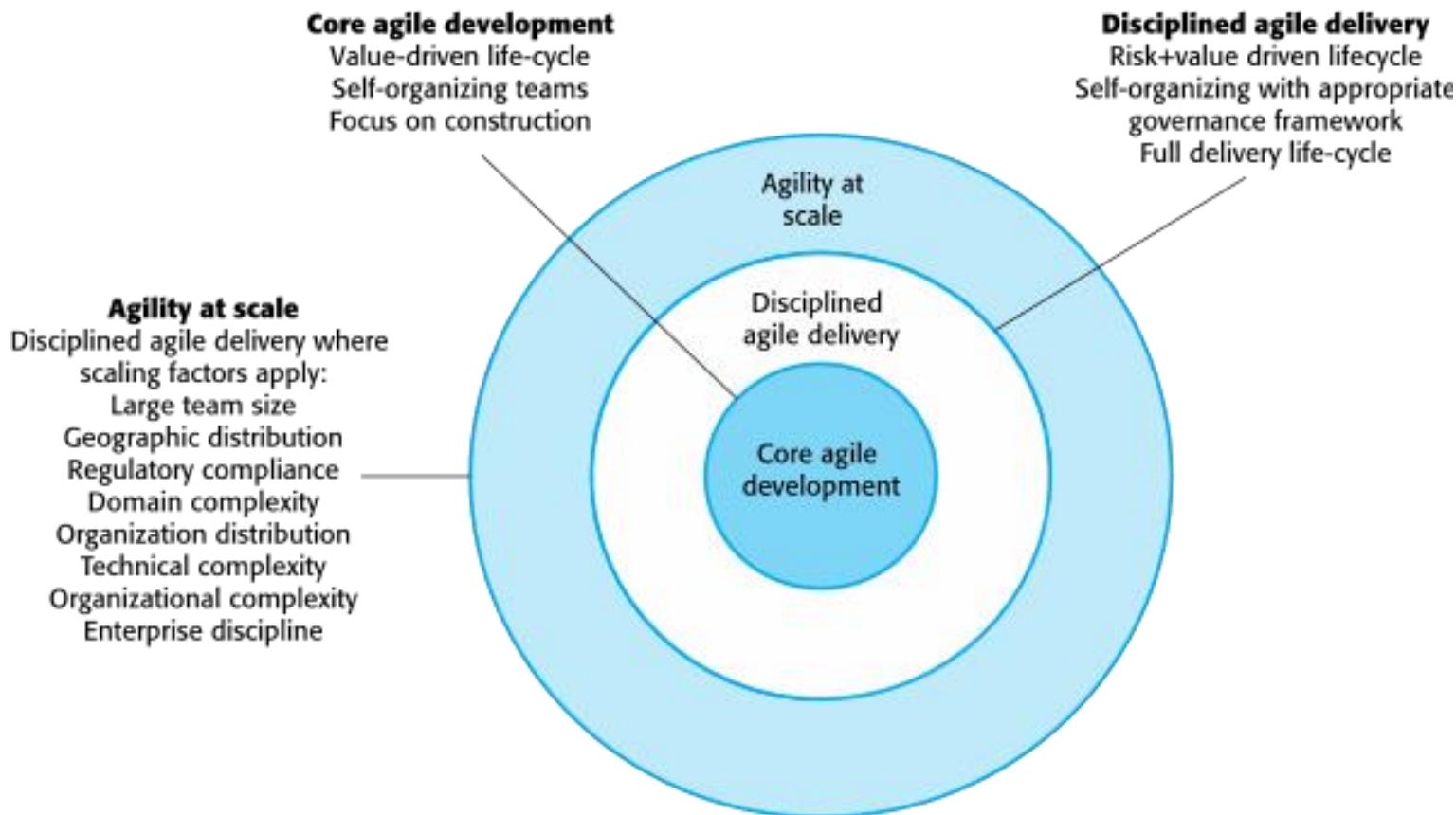
- ◊ Large systems are usually collections of separate, communicating systems, where separate teams develop each system. Frequently, these teams are working in different places, sometimes in different time zones.
- ◊ Large systems are ‘brownfield systems’, that is they include and interact with a number of existing systems. Many of the system requirements are concerned with this interaction and so they do not 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.

# IBM's agility at scale model





# Scaling up to large systems

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



# Multi-team Scrum

## ◊ *Role replication*

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



## Professional Tip of the Day

- ◊ How to ask for a raise or promotion?
  - Understand your leveling or promotion system
  - Contact HR for the job description or details
  - Ensure you meet the requirements and qualifications for the promotion
  - Document how you qualify for the promotion
  - Ask for projects that expand your development skills
  - Approach your boss or manager
  - Look for a new job



## Chapter 4 – Requirements Engineering

Ian Sommerville,

*Software Engineering*, 10<sup>th</sup> Edition

Pearson Education, Addison-Wesley

Note: These are a slightly modified version of Chapter 4 slides available from the author's site

<http://iansommerville.com/software-engineering-book/>



# Topics covered

---

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



# 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.
- ◊ Translation – Figure out **what** a system needs to do, write it down.



# What is a requirement?

- ◊ Some **function** or **characteristic** that must exist in a project
- ◊ It **may range** from a high-level abstract statement to an extremely detailed description of a function



# Types of requirement

## ◊ User requirements

- Statements in natural language plus diagrams of the services the system provides and its operational constraints. Written for customers.
  - High level, easy to understand, potentially verbose

## ◊ 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.
  - Lower level, more complex, quite detailed



# 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.

# Agile methods and requirements



- ◊ Many agile methods argue that producing detailed system requirements is a waste of time. Requirements change quickly, thus the requirements document is always out of date.
- ◊ This is problematic for systems that require pre-delivery analysis or systems developed by several teams.





# Functional and non-functional requirements

# 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 (**map directly to some code execution parts**)
  - What the user sees, how they interact with the system, what the system does
- May state what the system should not do

## ◊ Non-functional requirements

- **Constraints on (or characteristics of)** the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.
  - Behind the scenes, inform functional requirements
- Often apply to **the system as a whole** rather than individual features or services



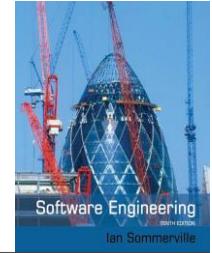
# Functional requirements

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



# Requirements imprecision

- ◊ Problems arise when functional requirements are not precisely stated
- ◊ Ambiguous requirements may be interpreted in different ways by developers and users



# Requirements completeness and consistency

- ◊ In principle, requirements should be both complete and consistent
- ◊ Complete
  - They should include descriptions of all functions required
- ◊ Consistent
  - There should be no conflicts or contradictions in the descriptions of the system facilities

# Requirements completeness and consistency



- ◊ At what point does working towards completeness take away from the project?



*"Good news! He said he only needs a few more weeks to finish the first draft of the Requirements Document."*



# Non-functional requirements

- ◊ These define **system properties and constraints** e.g., reliability, response time, storage requirements, platform
- ◊ Process requirements may also be specified mandating a particular IDE, programming language or development method
- ◊ While functional requirements map directly to running code, non-functional requirements **may be more critical than functional requirements**. If these are not met, the system may be useless.
- ◊ They may also **force functional requirements**. For instance a mobile application will require a touch based UI.

# 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



# Non-functional classifications

## ◊ Product requirements

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

## ◊ Organizational requirements

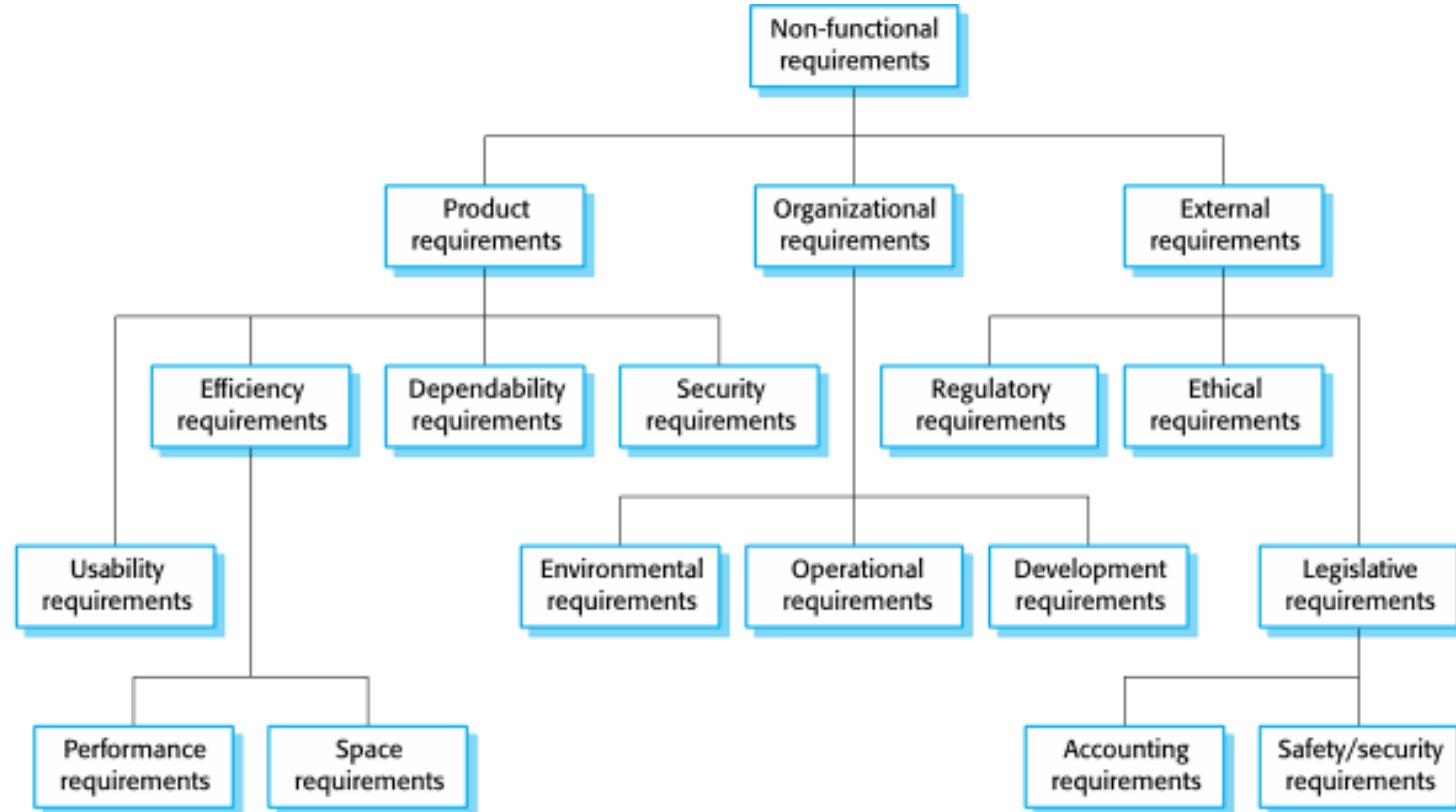
- Requirements which are a consequence of organizational policies and procedures, e.g. process standards used or implementation requirements

## ◊ External requirements

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



# Non-functional classifications



# 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. (Some nonfunctional requirements inform the need for specific functional requirements)

## External requirement

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



# 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.

# Metrics for specifying non-functional requirements



Software Engineering  
Ian Sommerville

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

# Requirements elicitation





# 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, and other. These are called **stakeholders**.
- ◊ Find requirements!



# 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.
- ◊ **Stages** include:
  - Requirements **discovery**
  - Requirements **classification and organization**
  - Requirements **prioritization and negotiation**
  - Requirements **specification**

# 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
- ◊ Organizational 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.

# Problems of requirements elicitation



## ◊ Reality versus Perception

- Customer may not understand why a computer can't do what a child can



IN CS, IT CAN BE HARD TO EXPLAIN  
THE DIFFERENCE BETWEEN THE EASY  
AND THE VIRTUALLY IMPOSSIBLE.



# Requirements discovery - 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 be 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



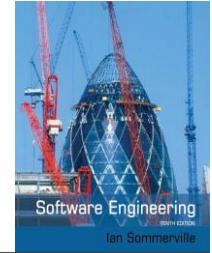
# Problems with interviews

- ◊ Application specialists may use language to describe their work that isn't easy for the requirements engineer to understand.
  - Every profession has its own language
- ◊ Interviews are not good for understanding domain requirements
  - Requirements engineers cannot understand specific domain terminology (see above)
  - Some domain knowledge is so familiar that people find it hard to articulate or think that it isn't worth articulating
  - The odd case that is unlikely to come up in an interview



## Requirements discovery - Ethnography

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



# 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 practical situations, stakeholders can relate to them and can comment on their situation with respect to the story.



# 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



# Requirements specification



# Requirements specification

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

# Ways of writing a system requirements specification



Notation	Description
<b>Natural language</b>	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. (almost pseudocode)
Graphical notations	Graphical models, supplemented by text annotations, are used to define the functional requirements for the system; UML use case and sequence diagrams are commonly used.
Mathematical specifications	These notations are based on mathematical concepts such as finite-state machines or sets. Although these unambiguous specifications can reduce the ambiguity in a requirements document, most customers don't understand a formal specification. They cannot check that it represents what they want and are reluctant to accept it as a system contract



# Requirements and design

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



# Natural language specification

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



# 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

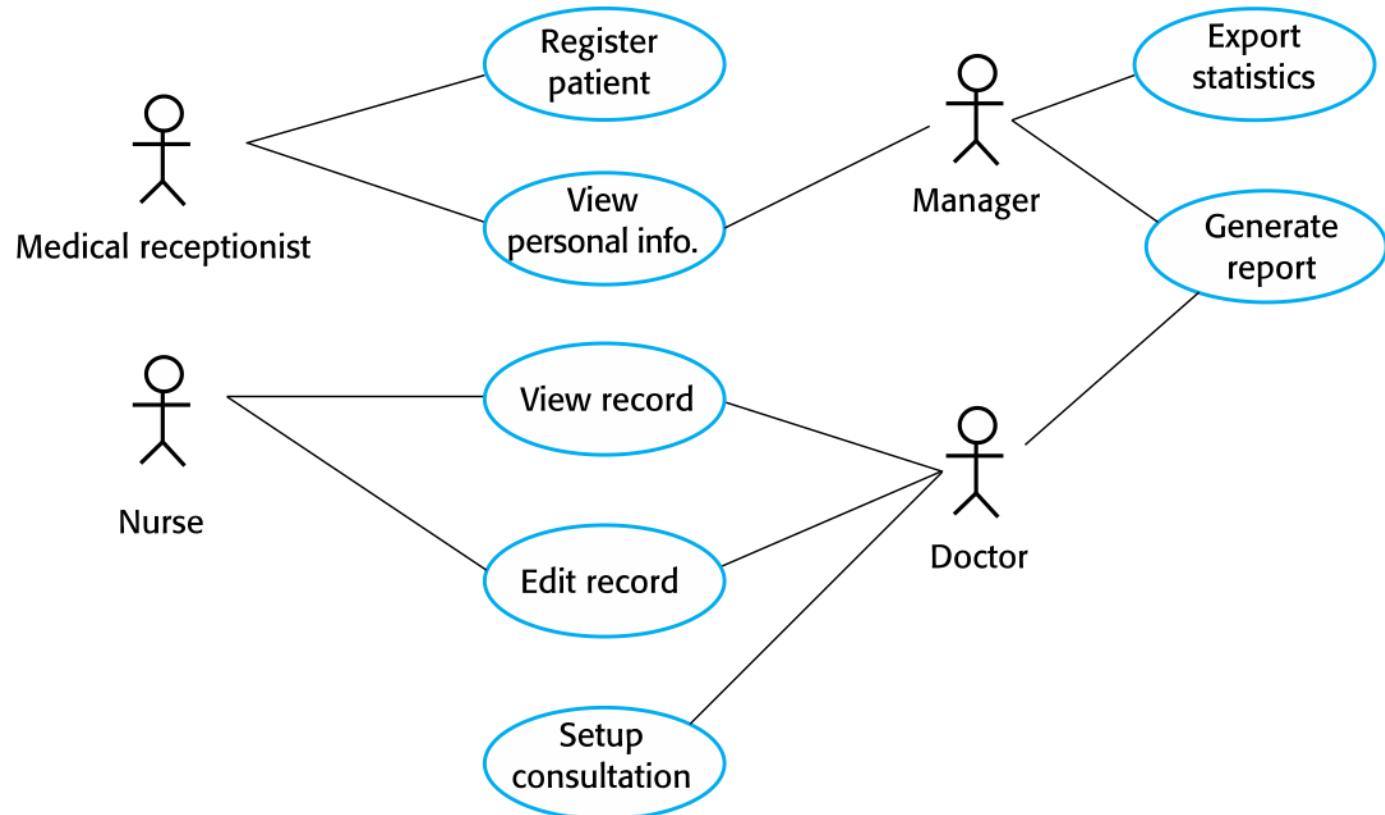


## Use cases

- ◊ Use-cases are a kind of scenarios included in 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 models 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





# 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.
- ◊ See general structure of a “**req spec**” document next, *but note that we use a “streamlined” version in Project Part #2*

# The structure of a requirements document



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

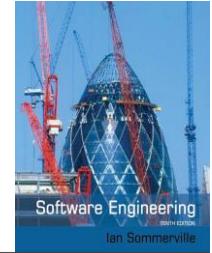
# The structure of a requirements document



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



# Requirements validation



# Requirements validation

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



# Requirements checking

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

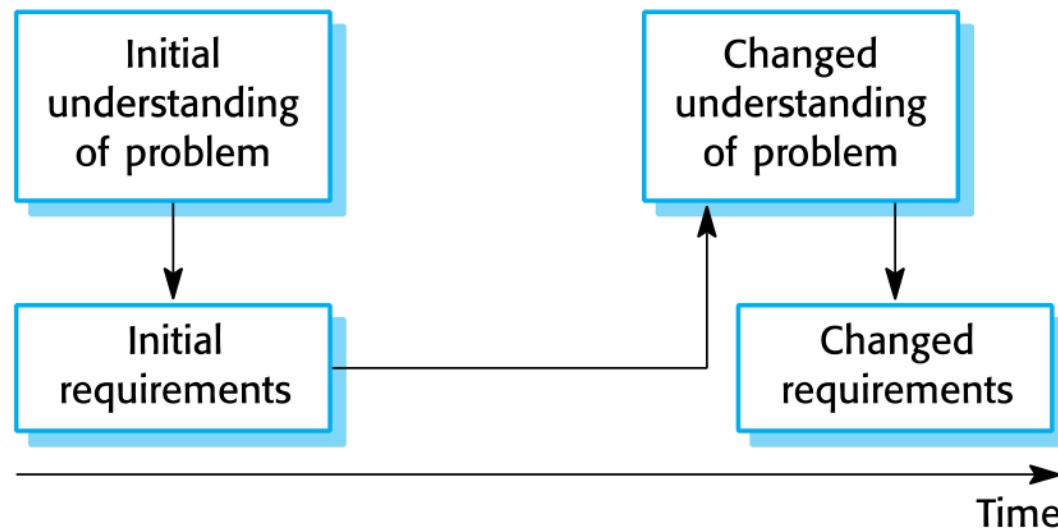


# Requirements change



# Requirements evolution

Software Engineering  
Ian Sommerville





# 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.



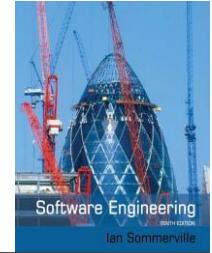
# Key points

- ◊ 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



# Key points

- ◊ The **requirements engineering process** is an iterative process that includes **requirements elicitation, specification, and validation**
- ◊ You can use a range of techniques for requirements elicitation including **interviews** and **ethnography**. **User stories** and **scenarios** may be used to facilitate discussions.
- ◊ **Requirements specification** is the process of formally documenting the user and system requirements and creating a software requirements document



# Key points

- ◊ The **software requirements document** is an agreed statement of the system requirements. It should be organized so that both system customers and software developers can use it.
- ◊ **Requirements validation** is the process of checking the requirements for validity, consistency, completeness, realism and verifiability
- ◊ Business, organizational and technical changes inevitably lead to changes to the requirements for a software system. **Requirements management** is the process of managing and controlling these changes.



## Chapter 5 – System Modeling

Ian Sommerville,

*Software Engineering*, 10<sup>th</sup> Edition

Pearson Education, Addison-Wesley

Note: These are a slightly modified version of Chapter 5 slides available from the author's site <http://iansommerville.com/software-engineering-book/>



# 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 modeling** 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 **lead to requirements** for the new system.
- ◊ Models of the new system are used during requirements engineering to help **explain the proposed requirements** to other system **stakeholders**. Engineers use these models to discuss **design proposals** and to document the system for implementation.
- ◊ In a **model-driven engineering process**, it is possible to **generate a complete or partial system implementation** from the system model.



# System perspectives

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



# UML diagram types

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

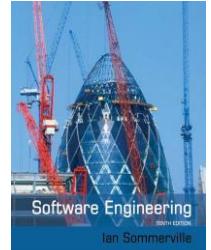


# Use of graphical models

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



# Context models



# Context models

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

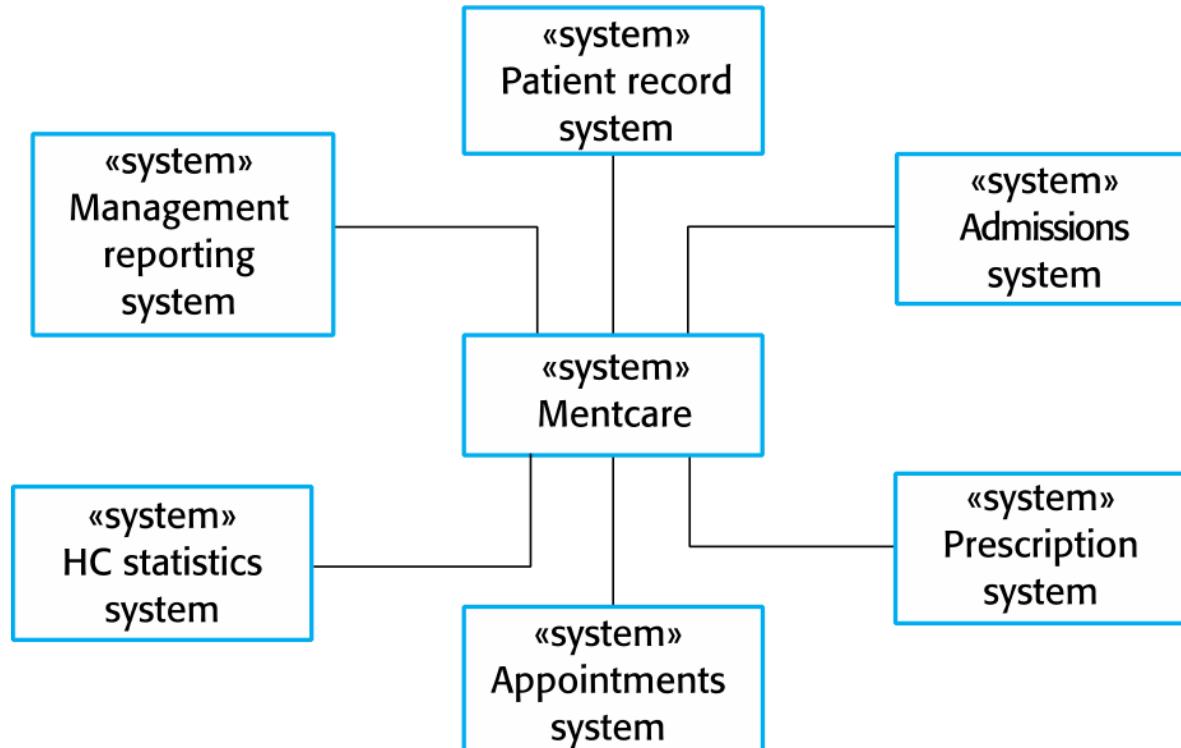


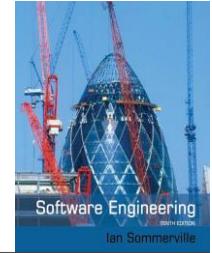
# System boundaries

- ◊ **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 may be 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



# 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 are incorporated into the UML
- ◊ Each use case represents a discrete task that involves external interaction with an actor
- ◊ Actors in a use case may be people or other systems



# Transfer-data use case

- ◊ A use case in the Mentcare system



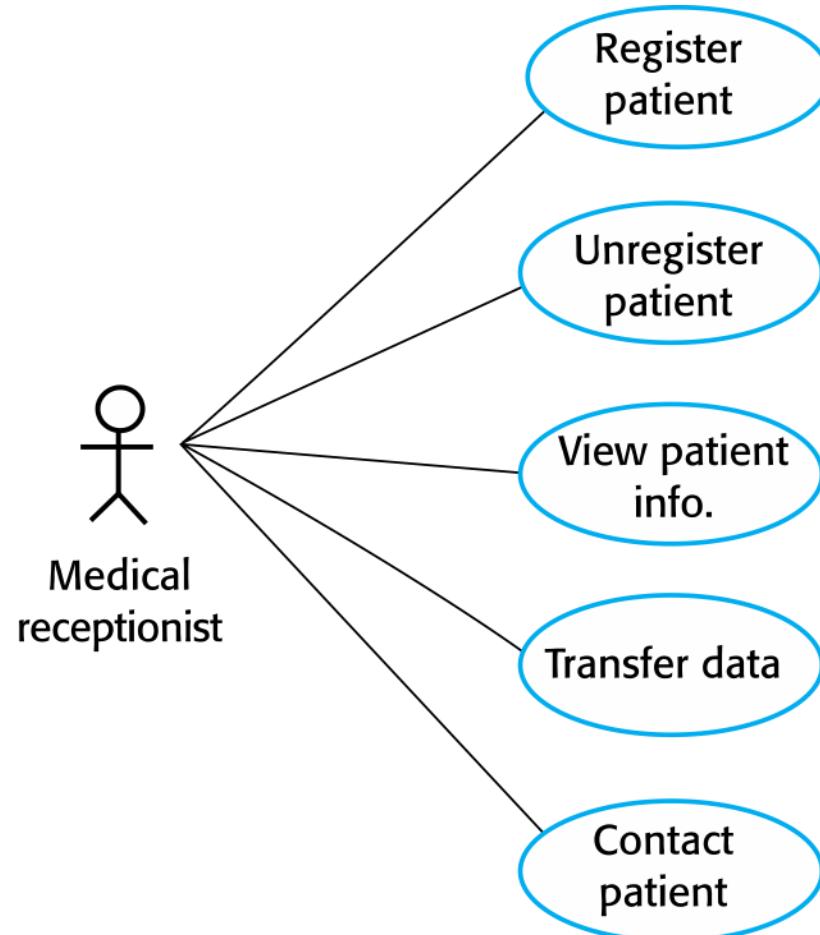
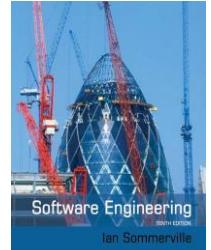
# Tabular description of the ‘Transfer data’ use-case



## 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’



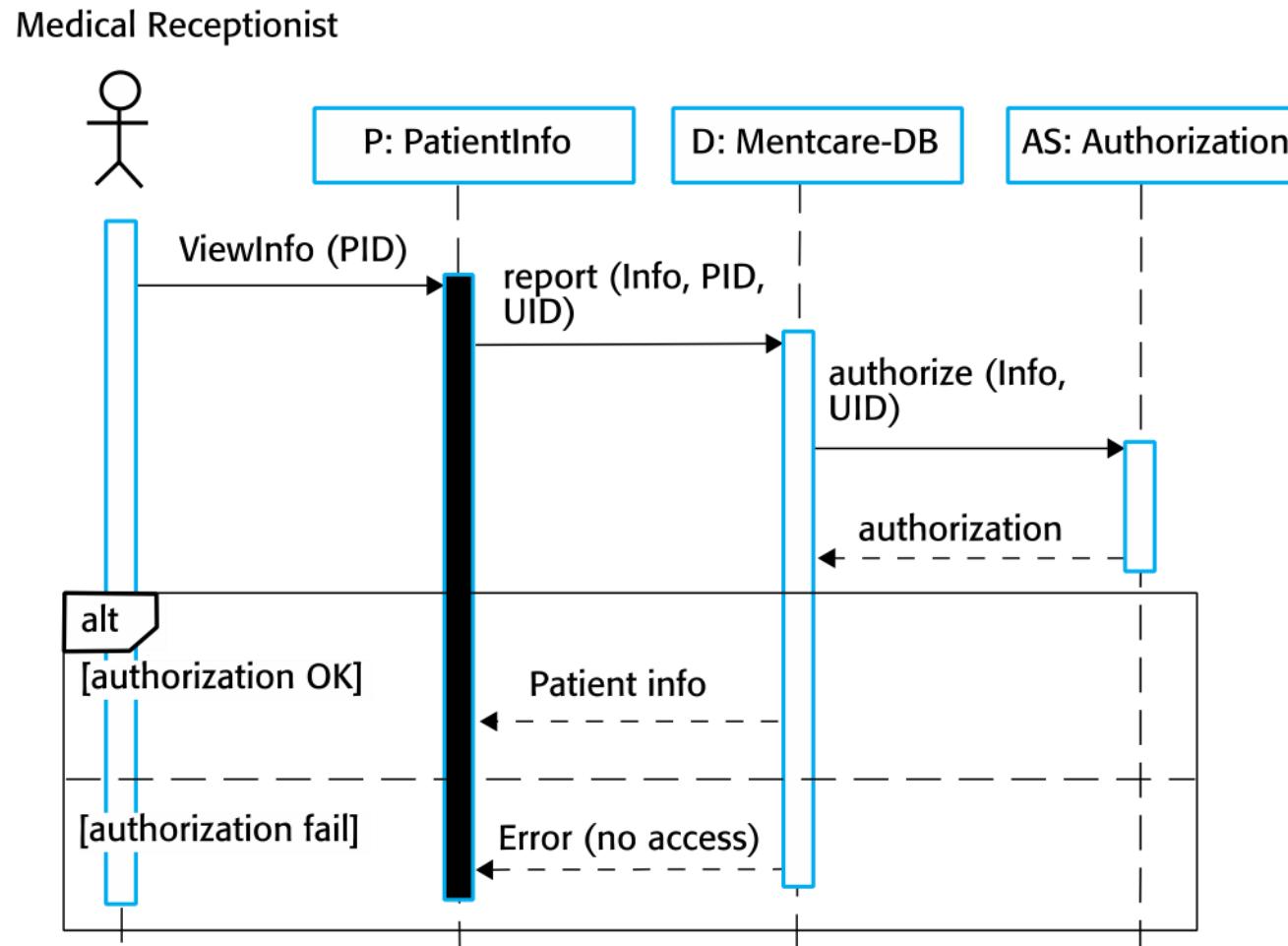


# Sequence diagrams

- ◊ 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





# Structural models



# Structural models

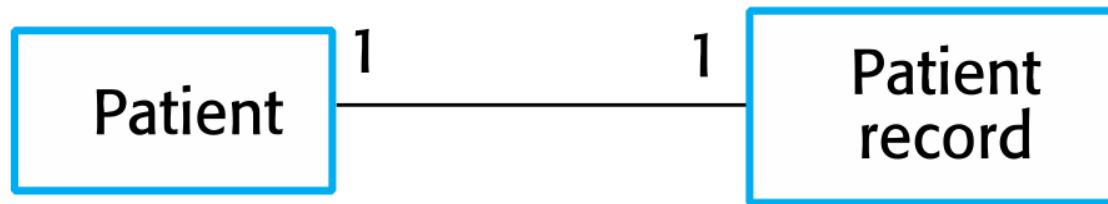
- ◊ **Structural models** of software display the organization of a system in terms of the components that make up that system and their relationships
- ◊ You create structural models of a system when you are discussing and designing the **system architecture**



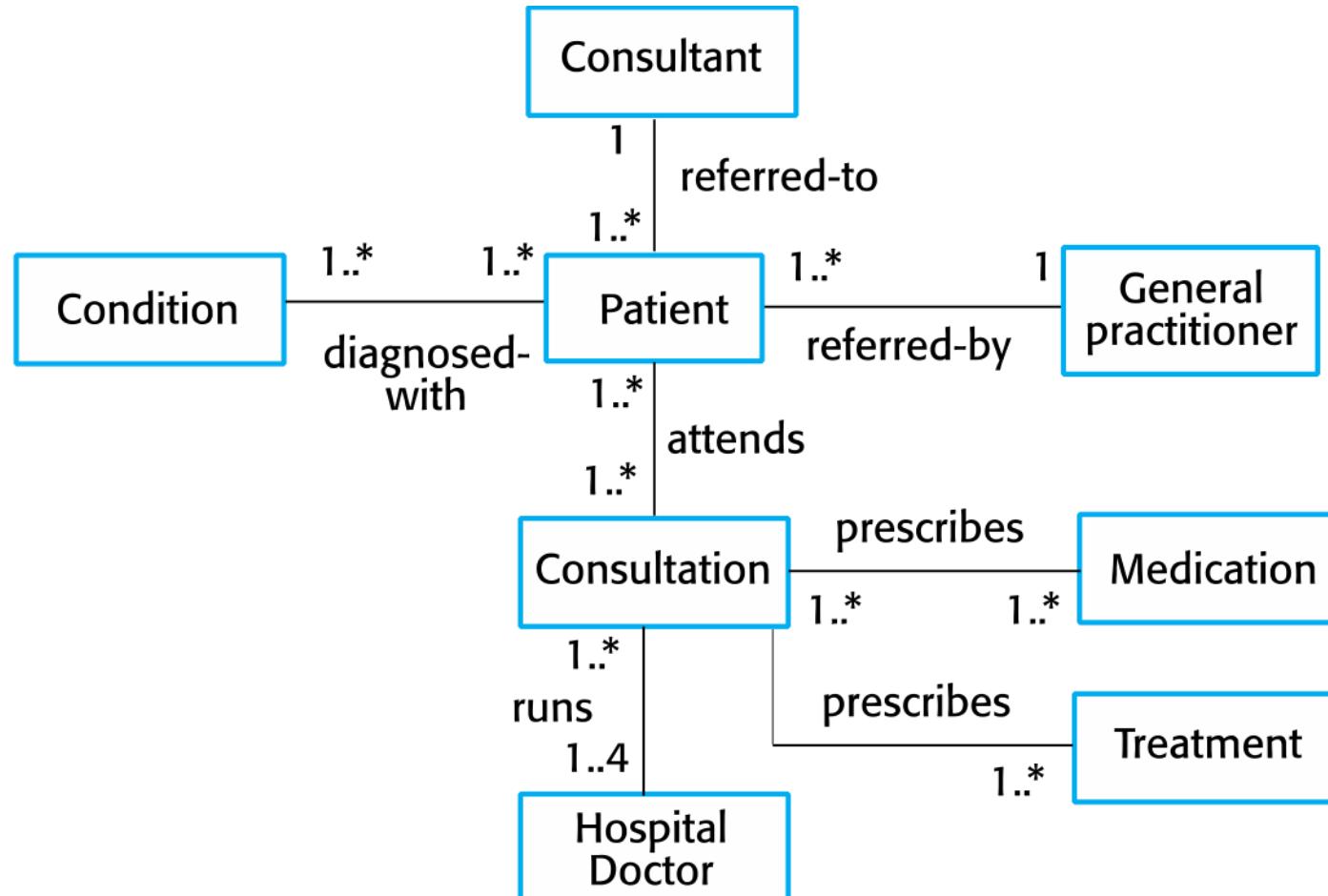
# Class diagrams

- ◊ **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.

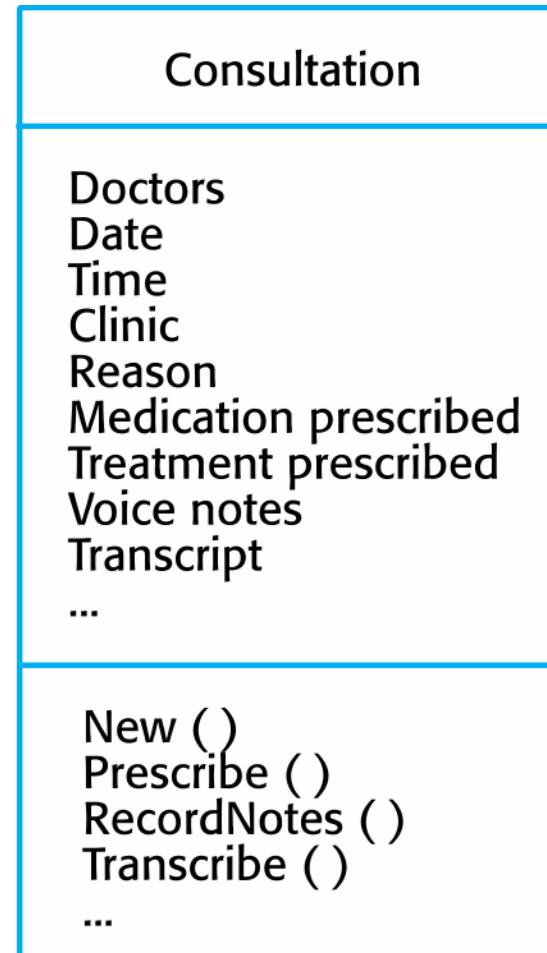
# UML classes and association



# 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

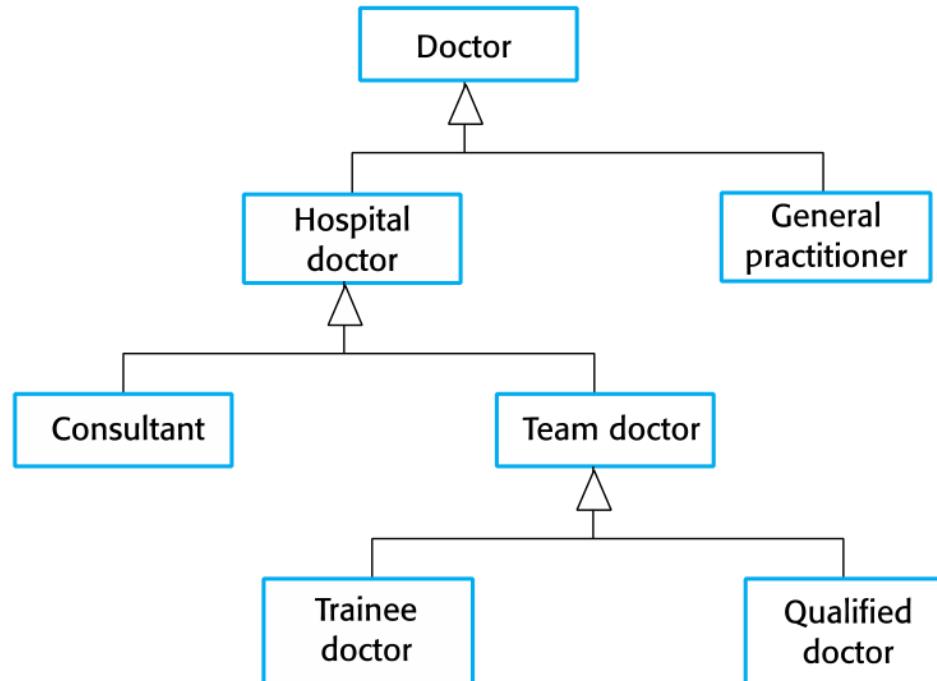


# Generalization

- ◊ 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 sub-classes that **inherit** the attributes and operations from their super-classes. These lower-level classes then *add more specific attributes and operations*.

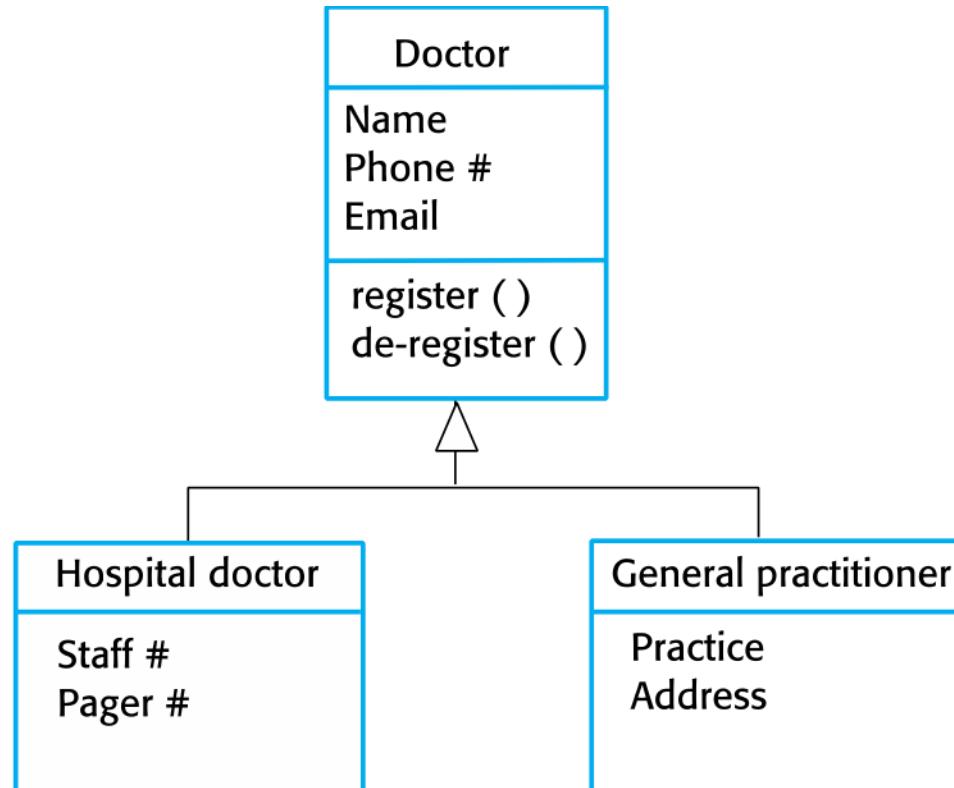


# A generalization hierarchy





# A generalization hierarchy with added detail



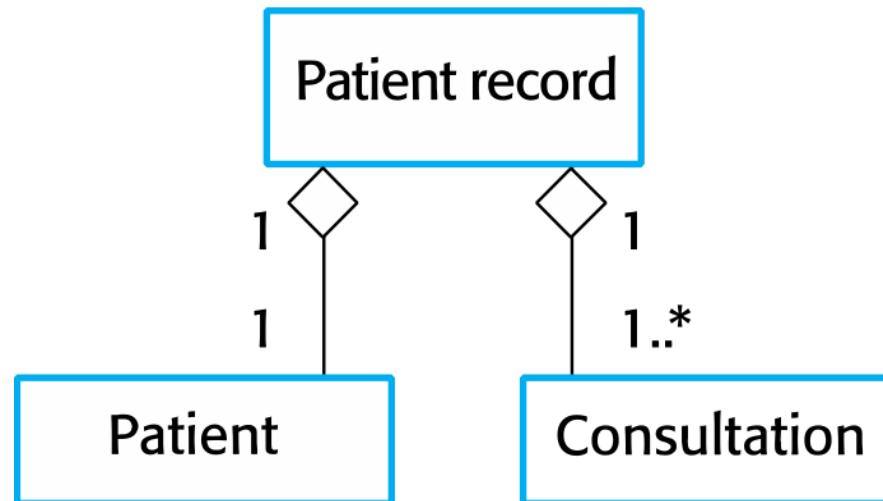


# Object class aggregation models

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



# 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.

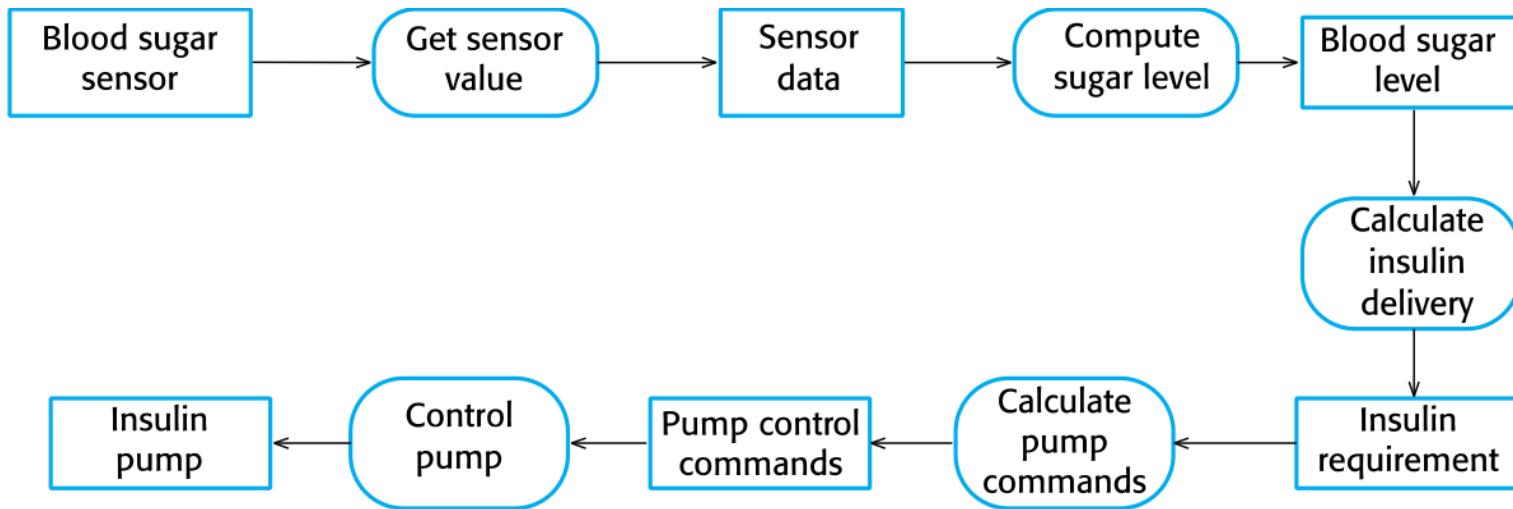


# Data-driven modeling

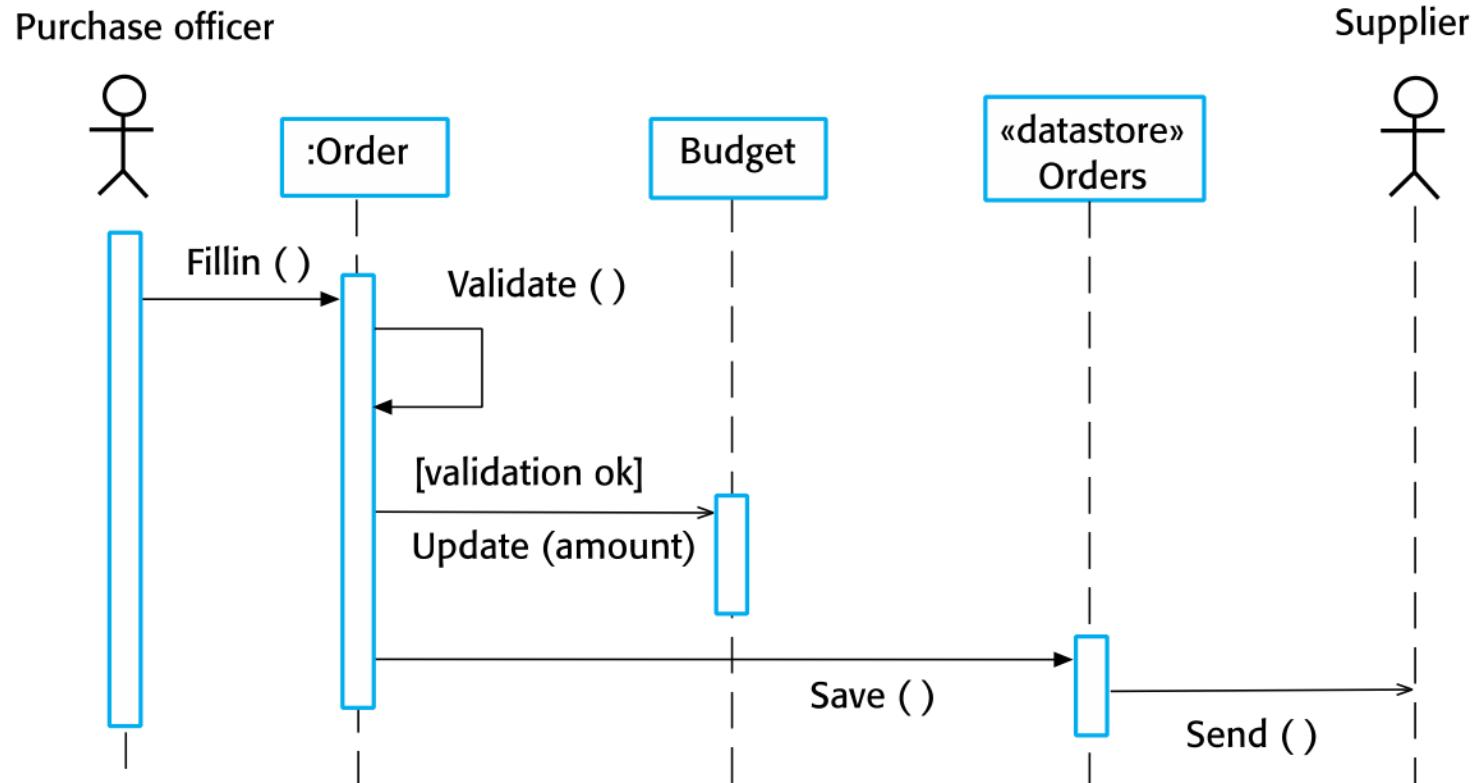
- ◊ Many business systems are data-processing systems that are primarily driven by data. They are **controlled by the data input to the system**, with relatively little external event processing.
- ◊ **Data-driven models** show the sequence of actions involved in processing input data and generating 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 (process models)**



# An activity model of the insulin pump's operation



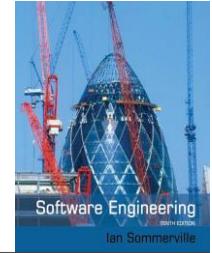
# Order processing





# 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

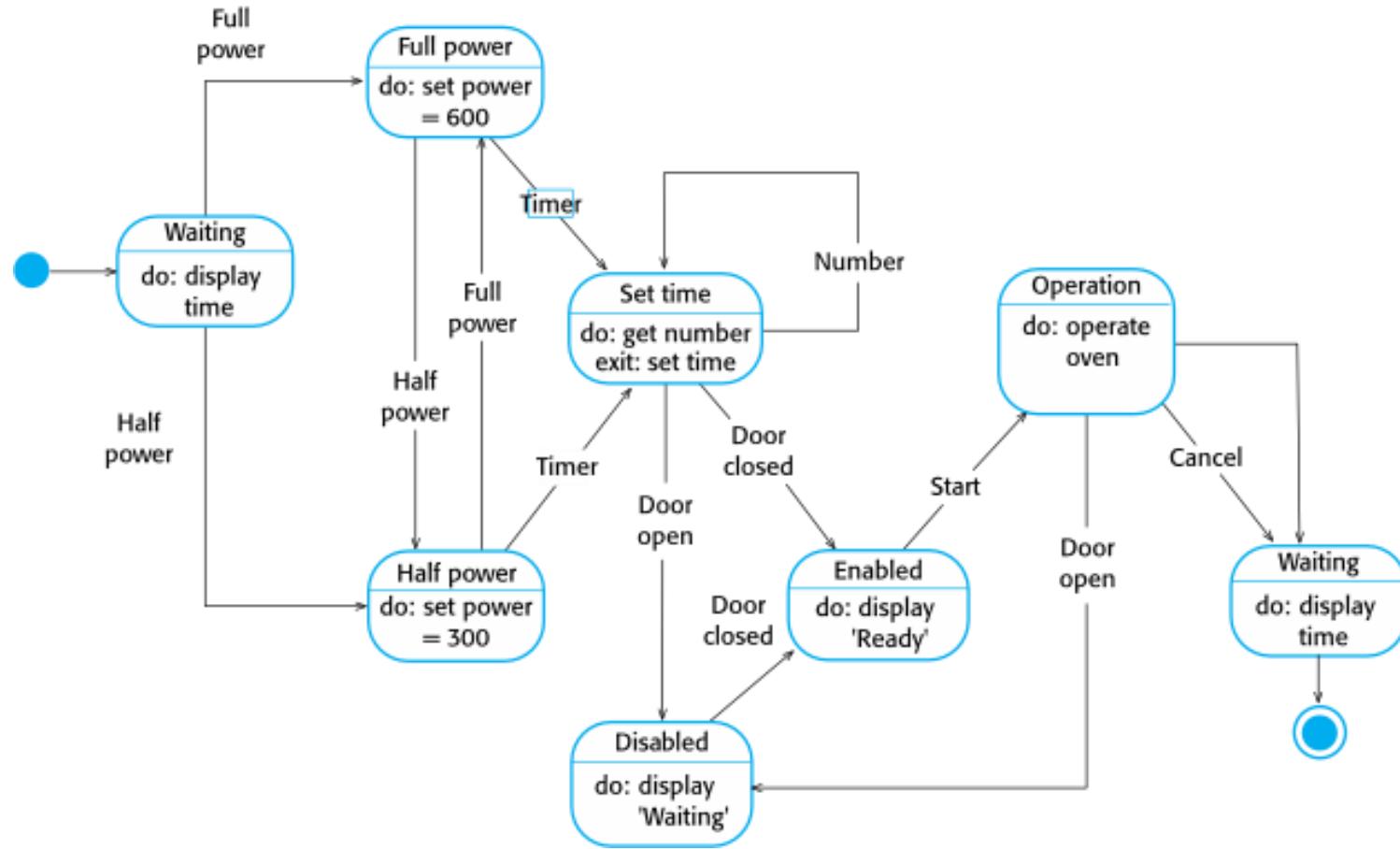


# State machine models

- ◊ These model the behavior 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 (**events, or triggers, are shown on transitions**). 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.

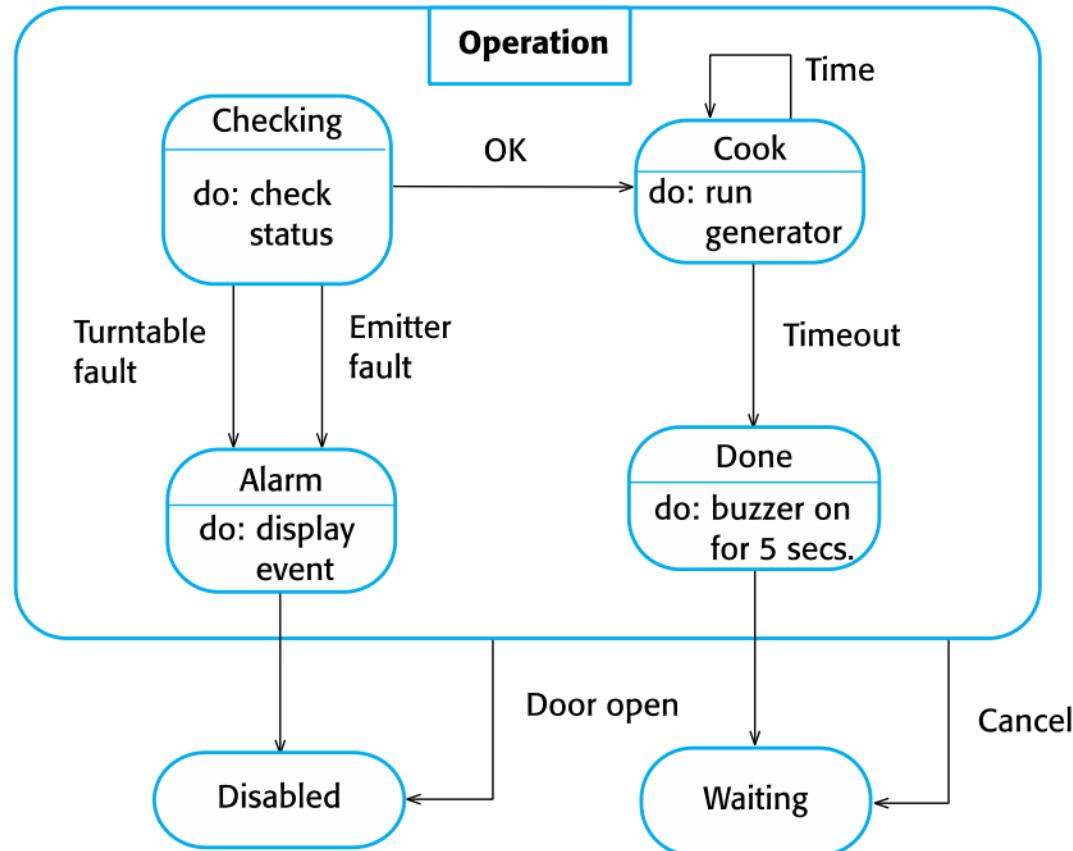


# State diagram of a microwave oven





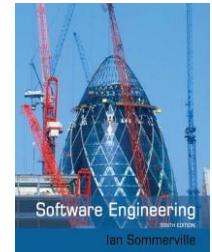
# Microwave oven operation





# 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.



# 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 a (fairly) 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.

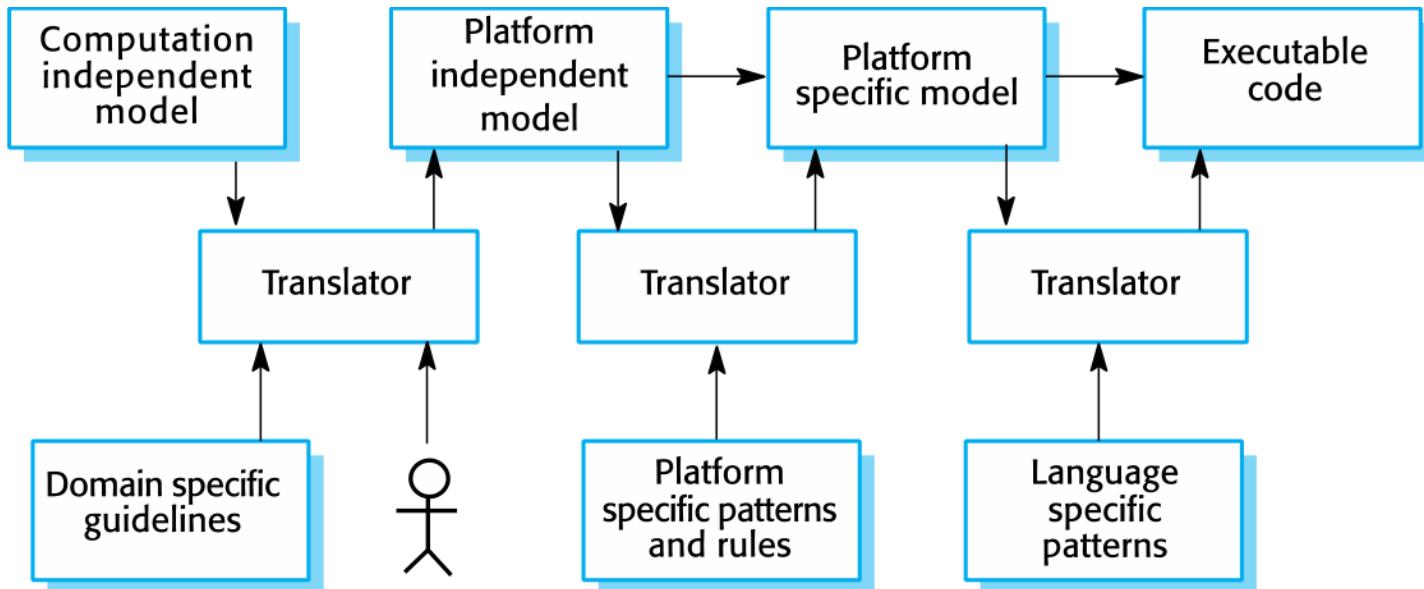


# Types of model

- ◊ 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.

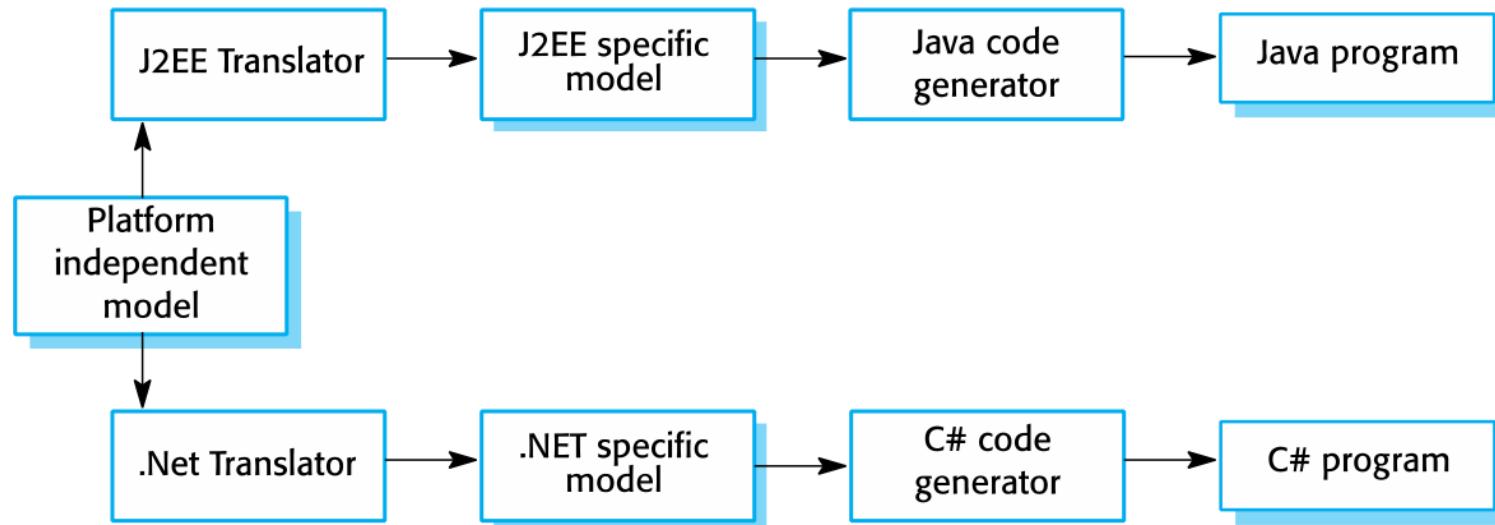


# MDA transformations





# 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



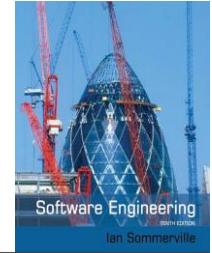
# Adoption of MDA

- ◊ 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 customization** 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**



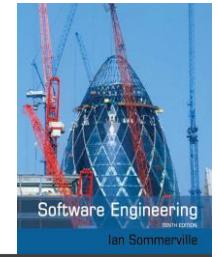
# Key points

- ◊ 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.



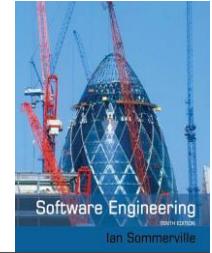
# Key points

- ◊ **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.



# Chapter 6 – Architectural Design

Note: These are a modified version of Chapter 6 slides available from the author's site <http://iansommerville.com/software-engineering-book/>



# Topics covered

---

- ◊ Architectural design decisions
- ◊ Architectural views
- ◊ Architectural patterns (styles)
- ◊ Application architectures



# Architectural design

- ◊ Architectural design is concerned with understanding how a software system should be organized and how to design 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

---

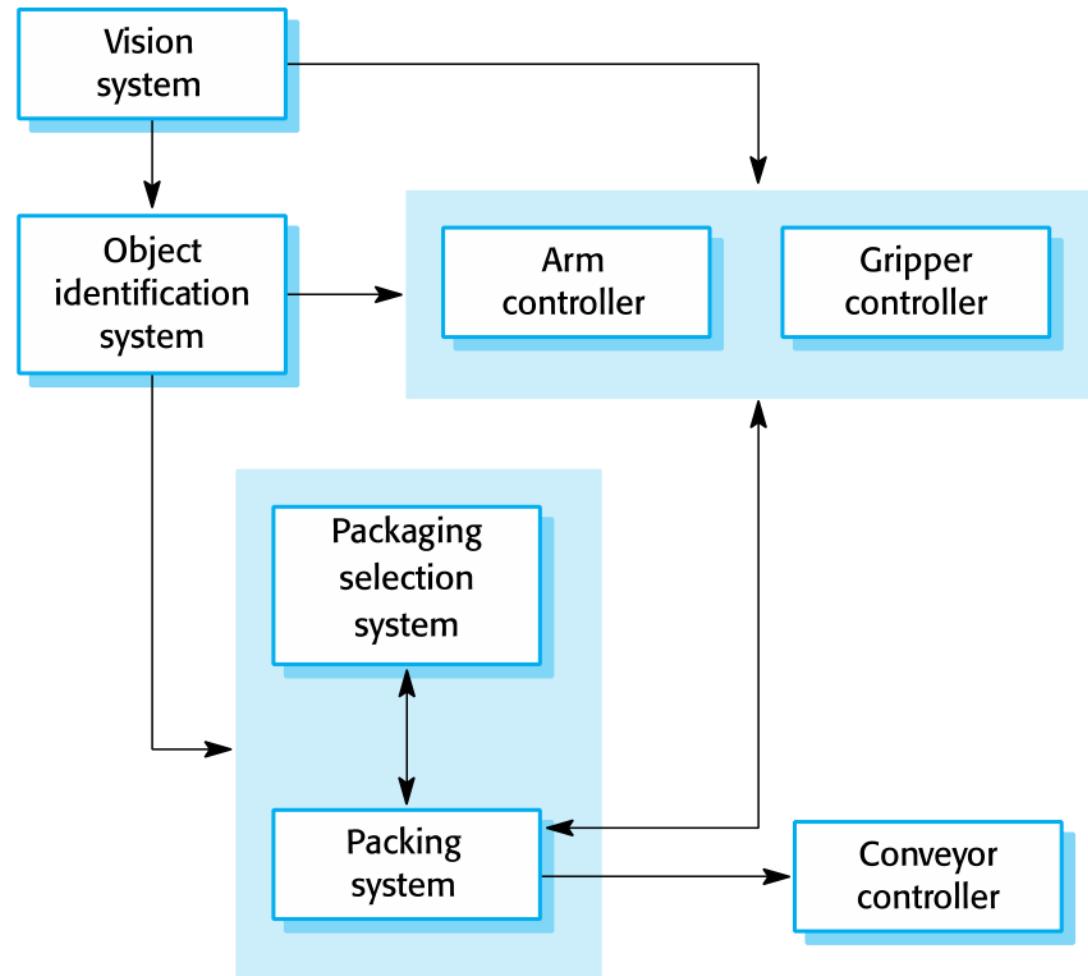


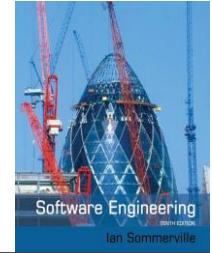
- ◊ It is generally accepted that an early stage of agile processes is to design an **overall systems architecture**
- ◊ **Refactoring** the system architecture is usually expensive because it affects so many components in the system

# The architecture of a packing robot control system



Software Engineering  
Ian Sommerville





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



# Use of architectural models

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



# Architectural design decisions

# Architectural design decisions

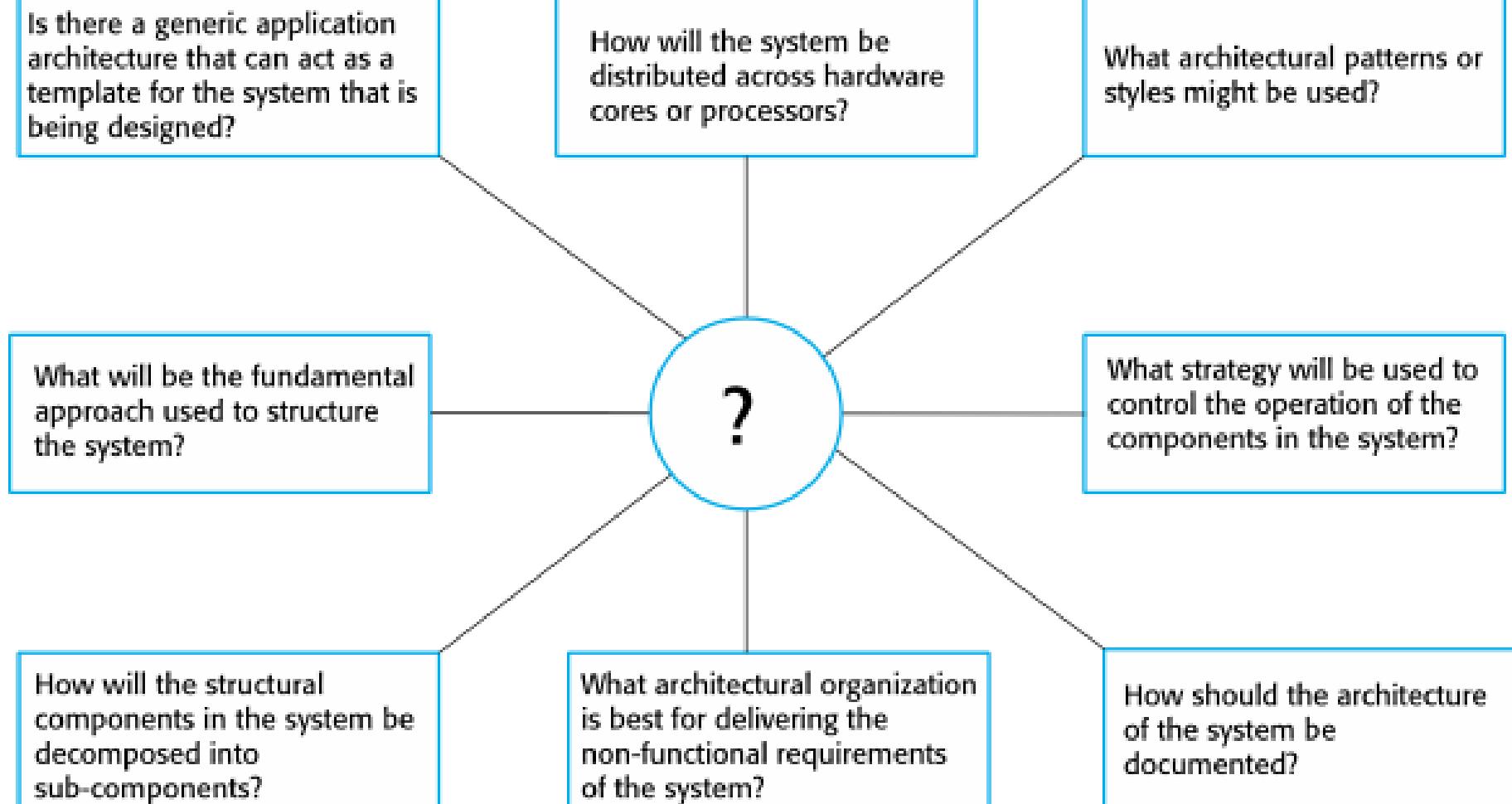


- ◊ Architectural design is a **creative process** so the process differs depending on the type of system being developed
- ◊ However, a number of **common decisions** span all design processes and these decisions affect the non-functional characteristics of the system



Software Engineering  
Ian Sommerville

# Architectural design decisions





# Architecture reuse

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

# Architecture and system characteristics



## ◊ Performance

- Localize critical operations and minimize communications. Use large rather than fine-grain components.

## ◊ Security

- Use a layered architecture with critical assets in the inner layers

## ◊ Safety

- Localize safety-critical features in a small number of sub-systems

## ◊ Availability

- Include redundant components and mechanisms for fault tolerance

## ◊ Maintainability

- Use fine-grain, replaceable components



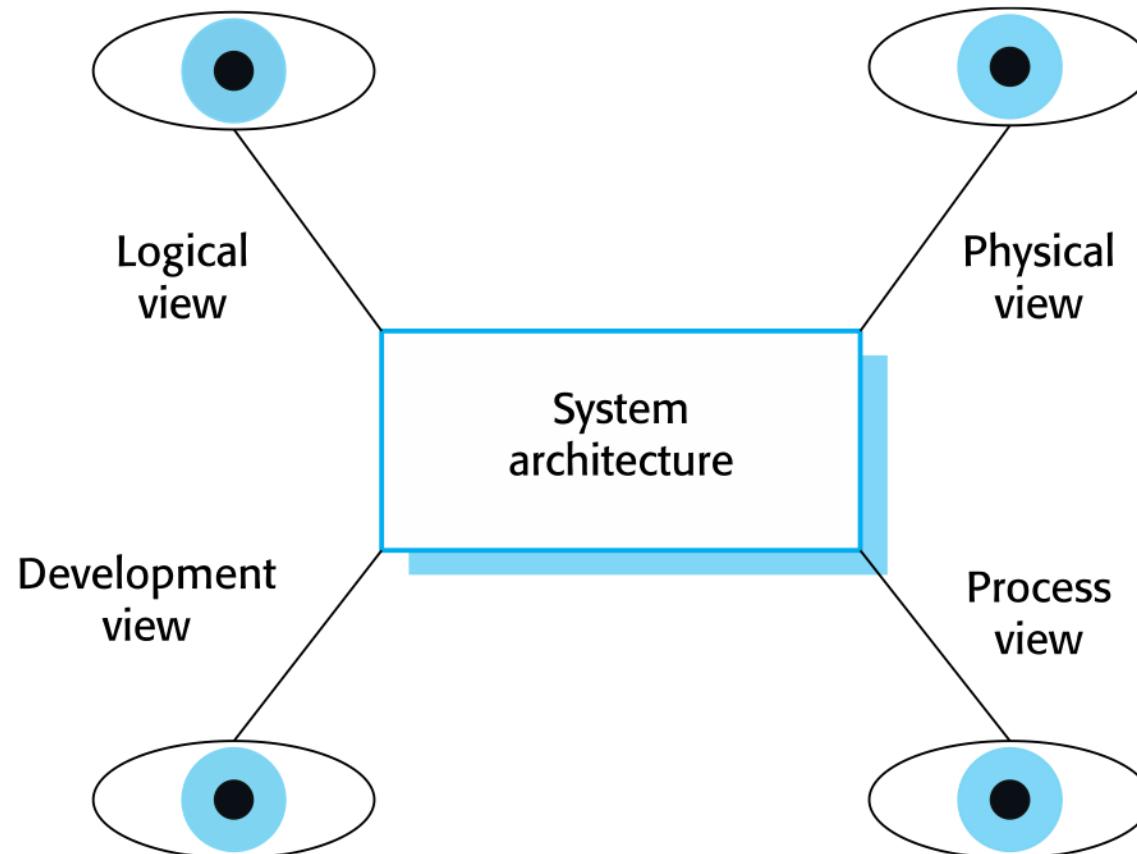
# Architectural views



# Architectural views

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

# Architectural views



## 4 + 1 view model of software architecture



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

# Architecture [Arlow and Neustadt, 2005]

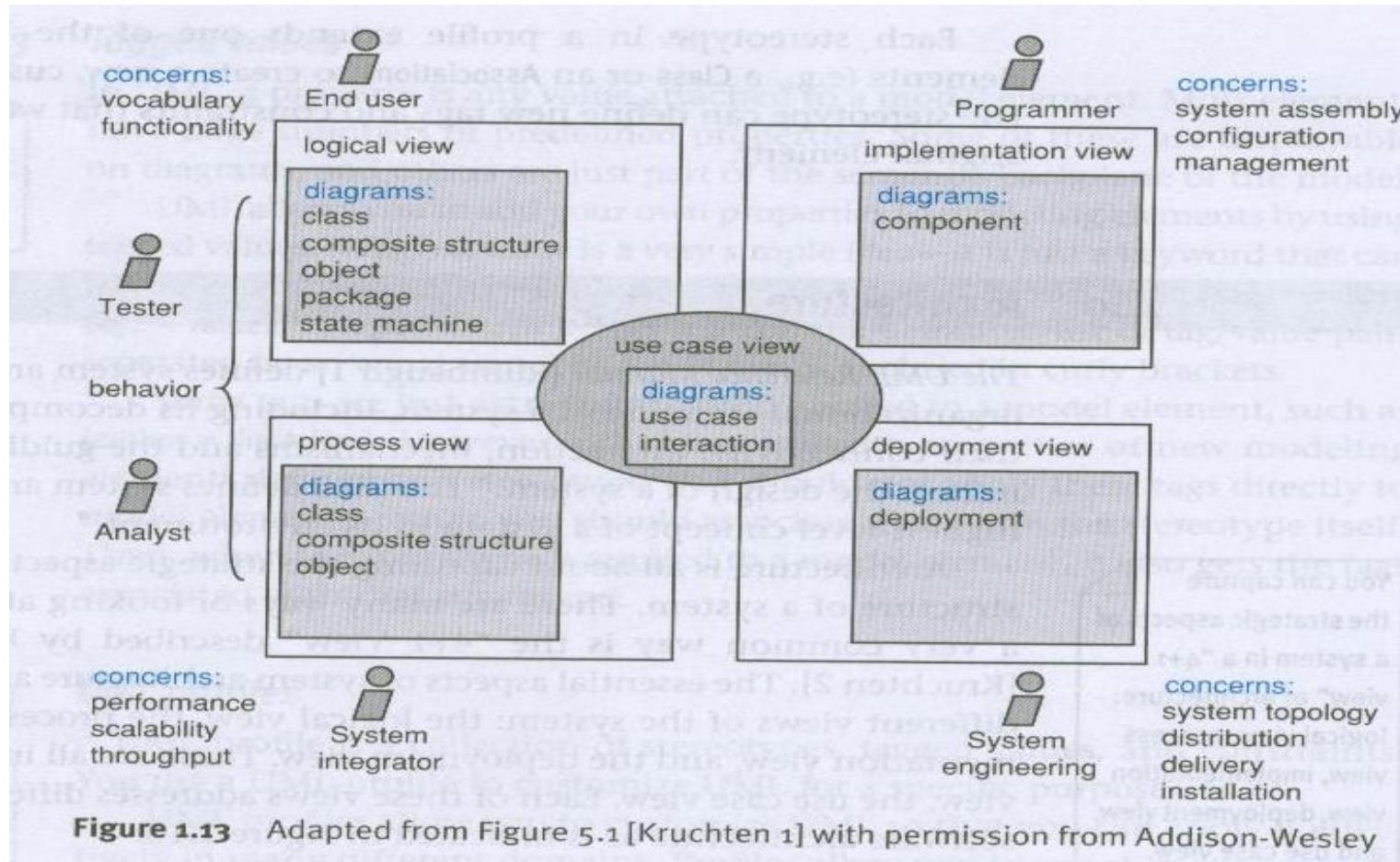


- The system architecture is “the organizational structure of the system, including its decomposition into parts, their connectivity, interaction, mechanisms and the guiding principles that inform the design of a system.” [Rumbaugh 1998]
- There is a typical “4+1 views” architecture of a system defined by UML:
  - Logical view*, captures the vocabulary of the problem domain using classes and objects
  - Process view*, depicts the threads and processes of the system as active classes
  - Implementation view*, shows the physical code base of the system in terms of components
  - Deployment view*, models the physical deployment of components onto computational nodes
- *Use case view*, captures the requirements of the system using a set of use cases. This is the view “+1” to which all other views connect.

# Architecture [Arlow and Neustadt, 2005]



The “4 + 1 views” architecture, Fig. 1.13 [Arlow & Neustadt 2005]



**Figure 1.13** Adapted from Figure 5.1 [Kruchten 1] with permission from Addison-Wesley



# Representing architectural views

- ◊ Some people argue that the **Unified Modeling Language (UML)** is an appropriate notation for describing and documenting system architectures
- ◊ Sommerville disagrees with this as he does not think that the UML includes abstractions appropriate for high-level system description
- ◊ **Architectural description languages (ADLs)** have been developed but are not widely used



# Architectural patterns



# Architectural patterns

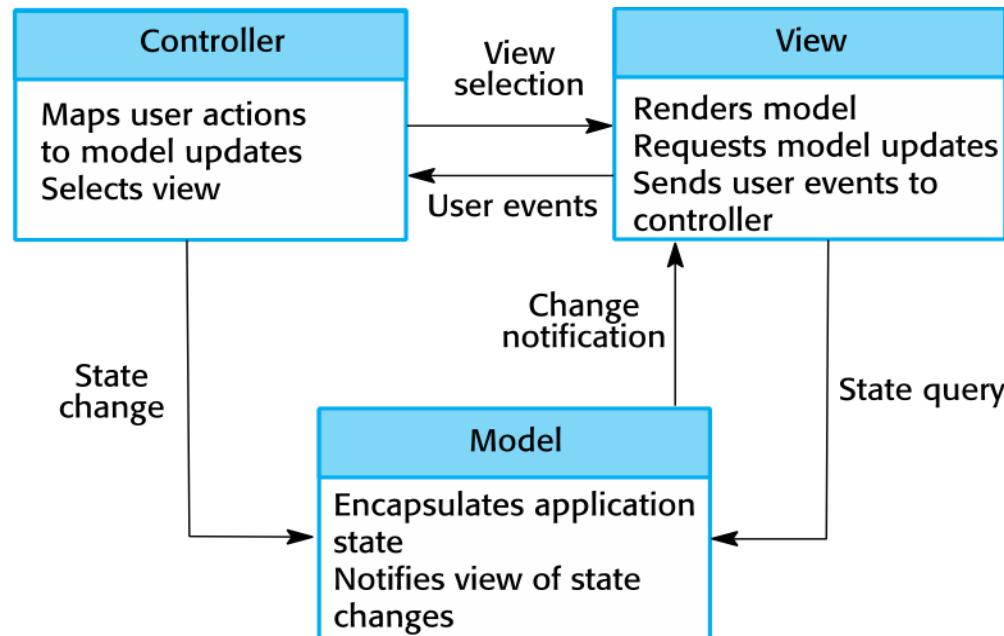
- ◊ Patterns (**styles**) are a means of representing, sharing and reusing knowledge
- ◊ An **architectural pattern** is a **stylized description of good design practice**, which has been tried and tested in different environments
- ◊ Patterns should include information about when they are and when they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

# The Model-View-Controller (MVC) pattern

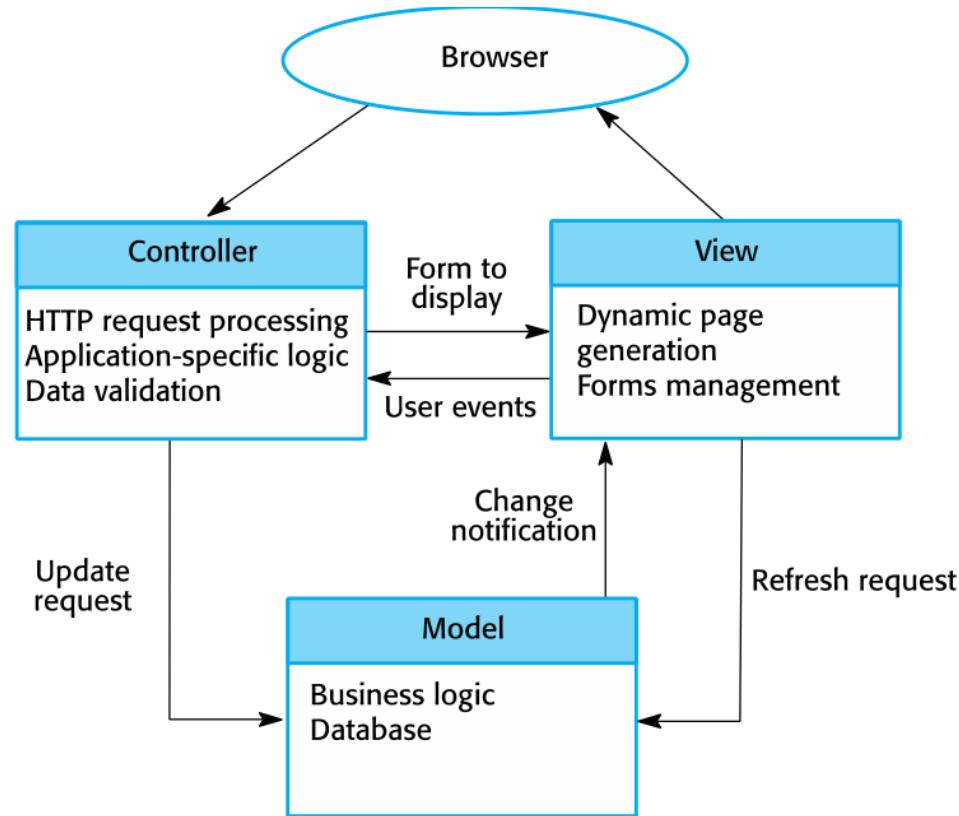


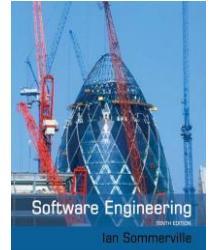
Name	MVC (Model-View-Controller)
Description	Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines 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**
- ◊ Organizes 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, it is often artificial to structure systems in this way

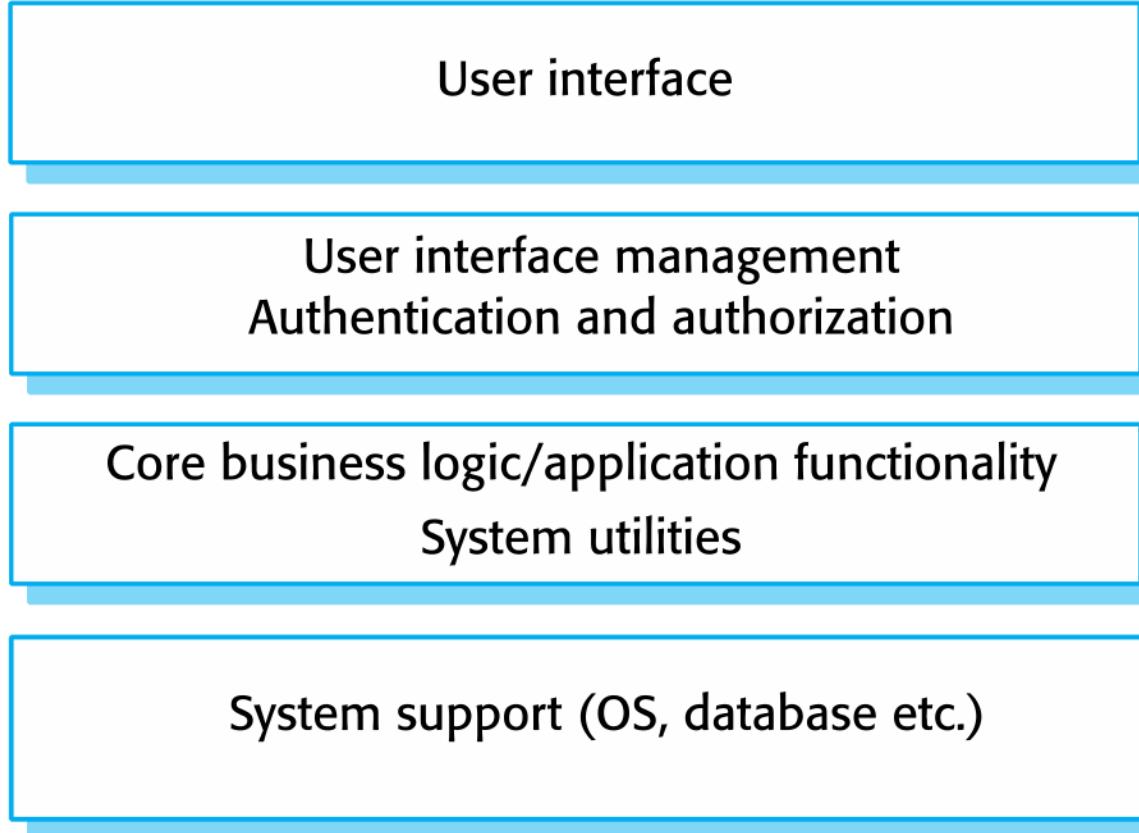


# The Layered architecture pattern

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

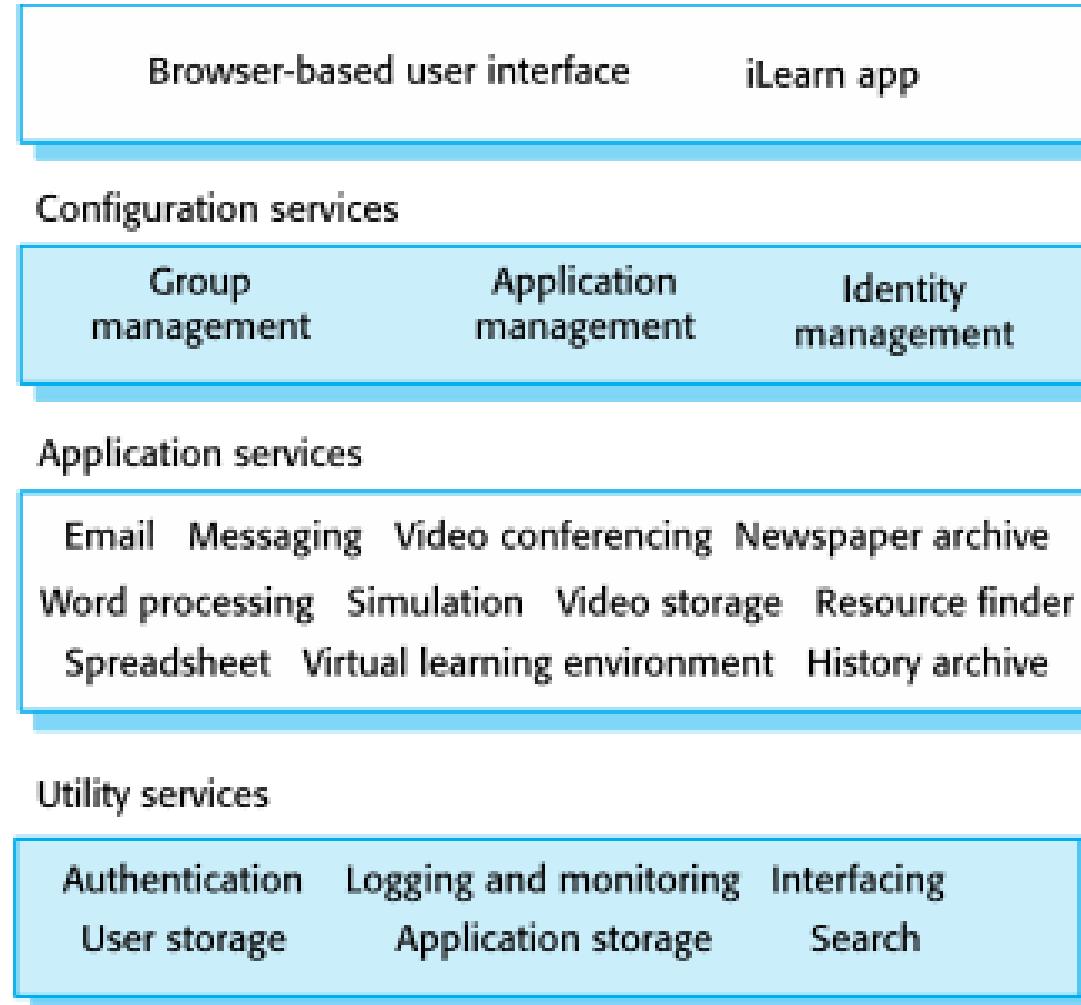


# A generic layered architecture





# The architecture of the iLearn system





# Repository architecture

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

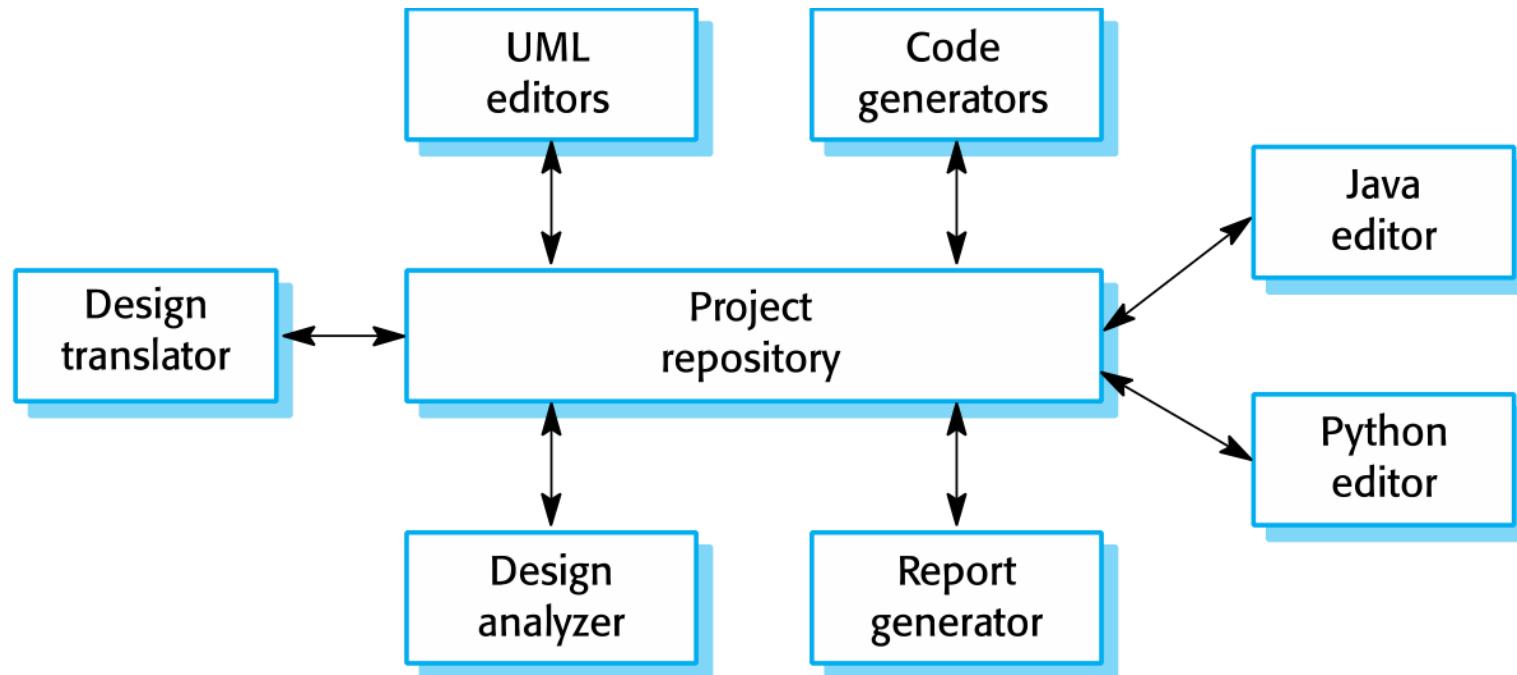


# The Repository pattern

Name	Repository
<b>Description</b>	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.
<b>Example</b>	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.
<b>When used</b>	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.
<b>Advantages</b>	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.
<b>Disadvantages</b>	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

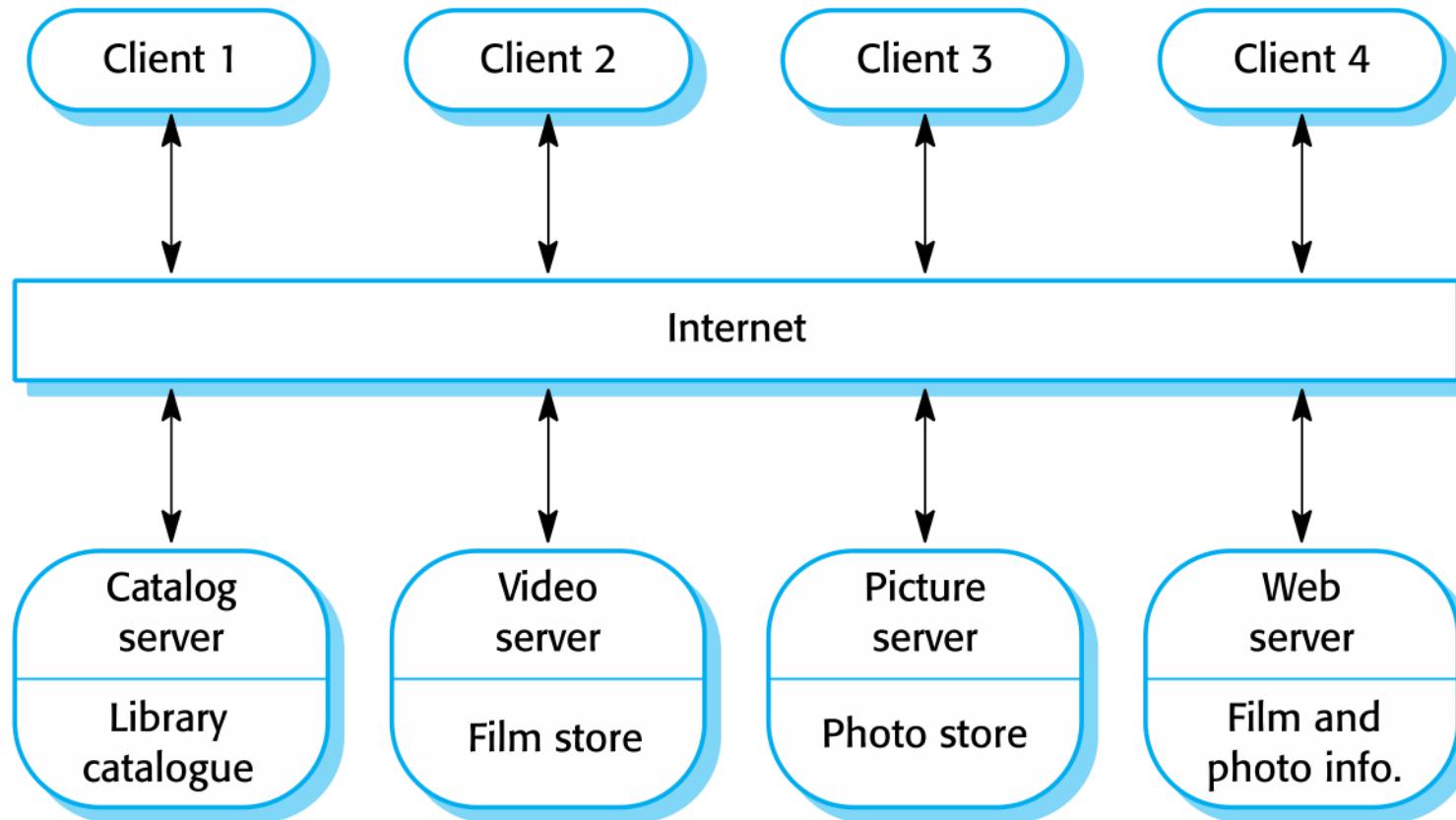


# The Client–server pattern

Name	Client-server
<b>Description</b>	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.
<b>Example</b>	Figure 6.11 is an example of a film and video/DVD library organized as a client–server system.
<b>When used</b>	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.
<b>Advantages</b>	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.
<b>Disadvantages</b>	Each service is a single point of failure so susceptible to denial of service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system. May be management problems if servers are owned by different organizations.



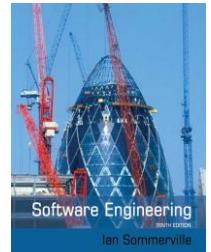
# A client–server architecture for a film library





# Pipe and filter architecture

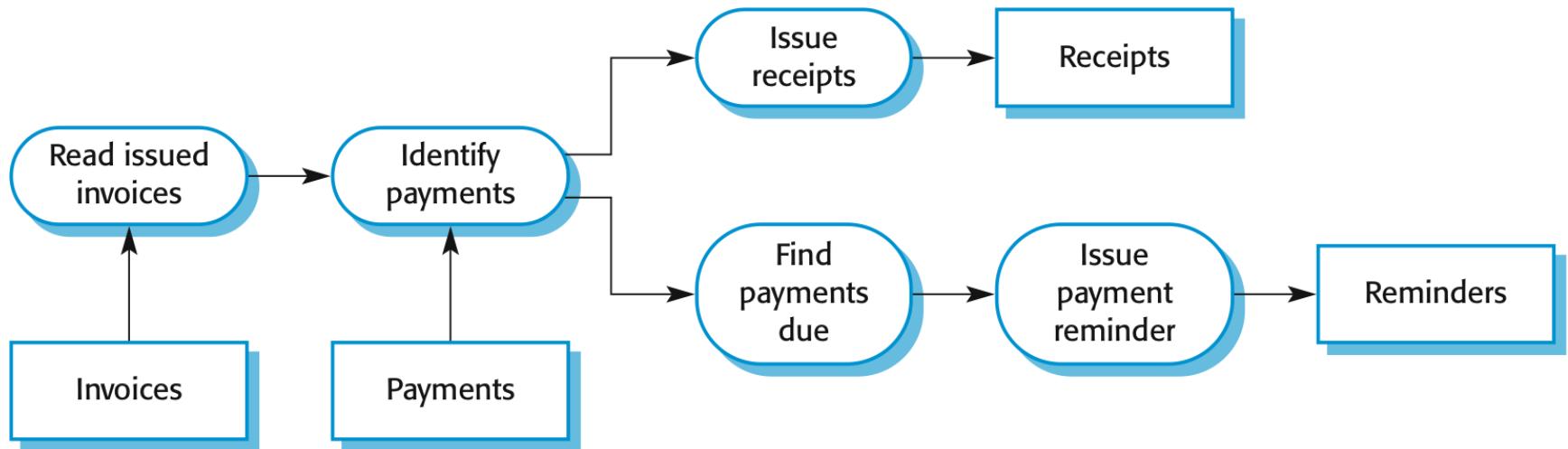
- ◊ Functional transformations process their inputs to produce outputs
- ◊ May be referred to as a pipe and filter model (as in UNIX shell)
- ◊ Variants of this approach are very common. When transformations are sequential, this is a batch sequential model which is extensively used in data processing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



# The pipe and filter pattern

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

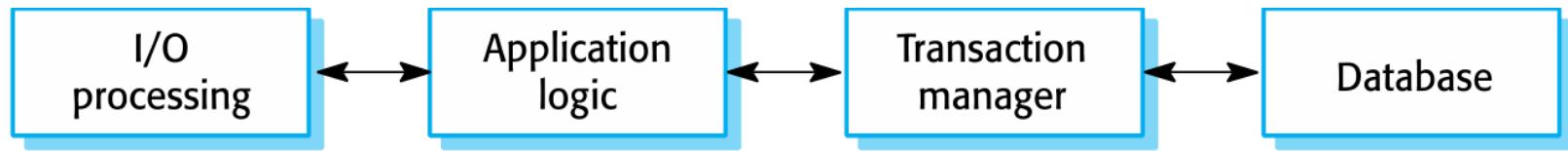
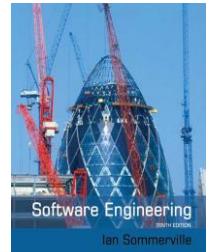
- Applications where the users' intentions are specified in a formal language that is processed and interpreted by the system



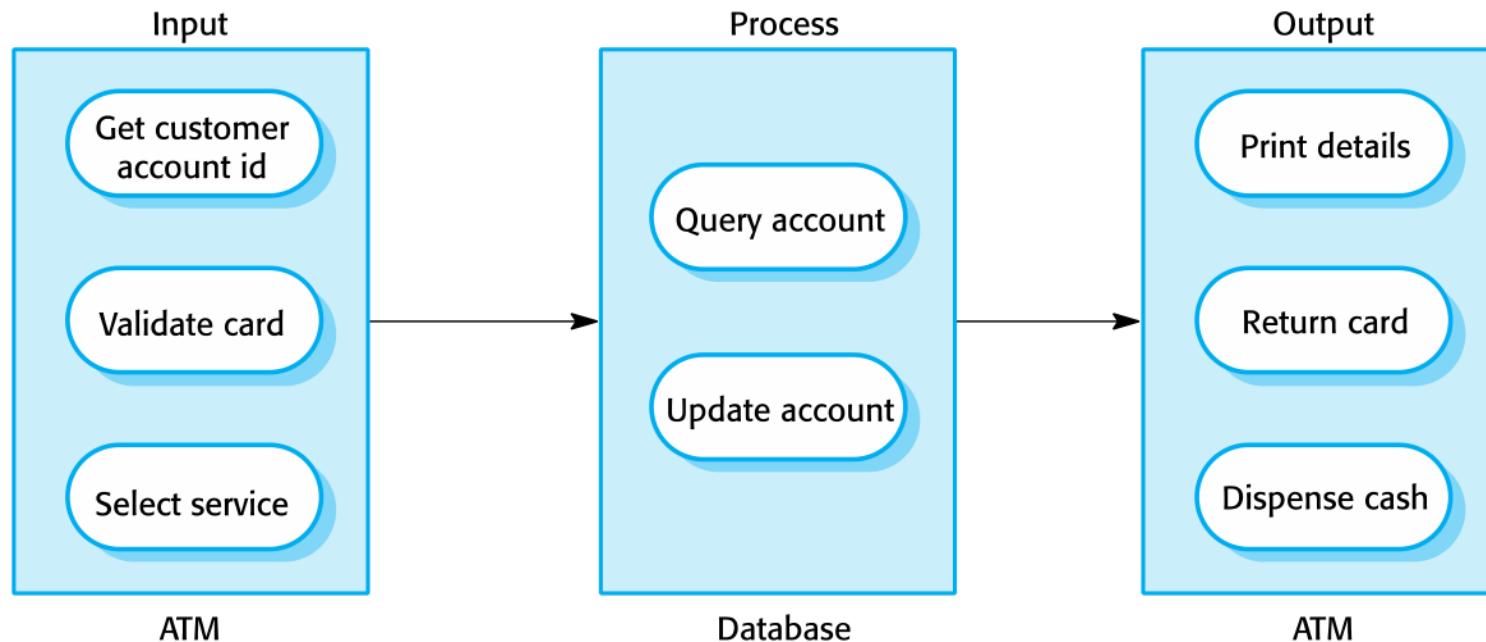
# Transaction processing systems

- ◊ 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



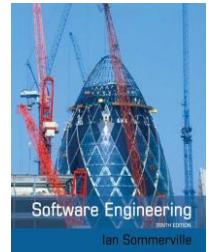
# The software architecture of an ATM system



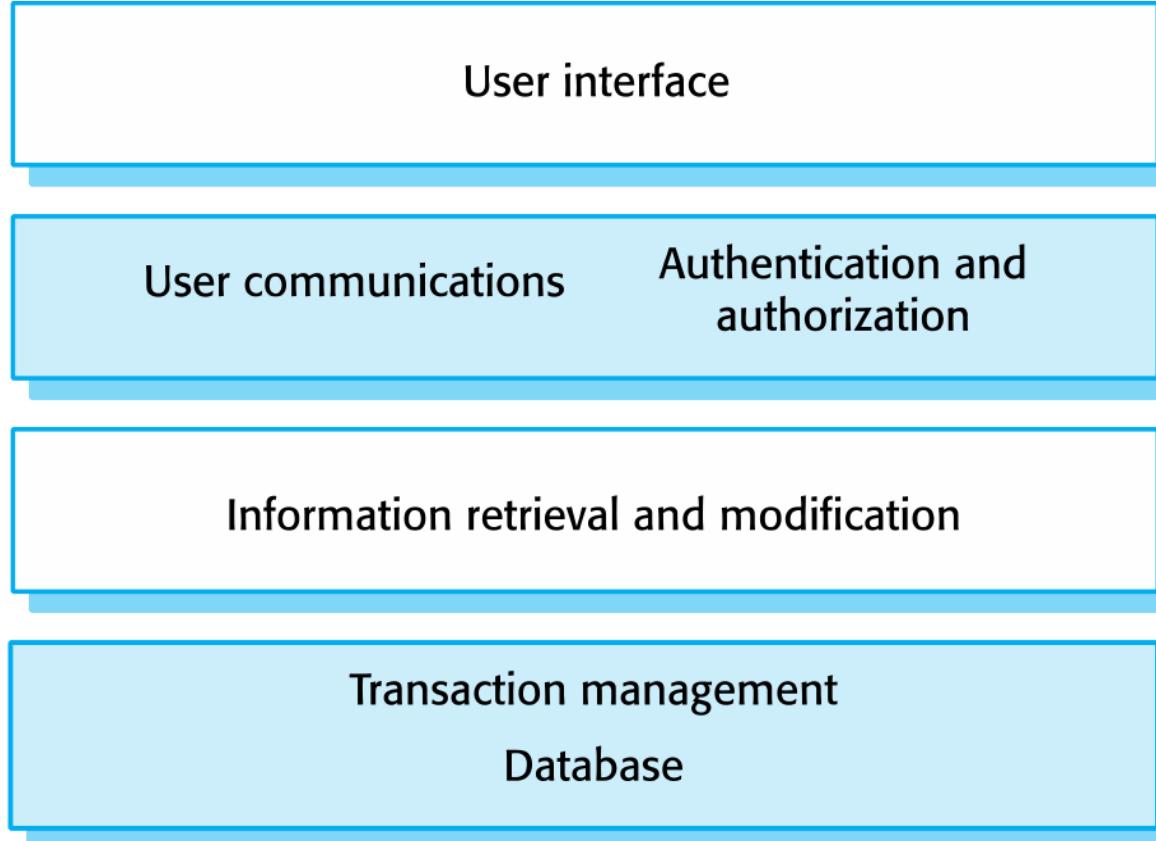
# Information systems architecture



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

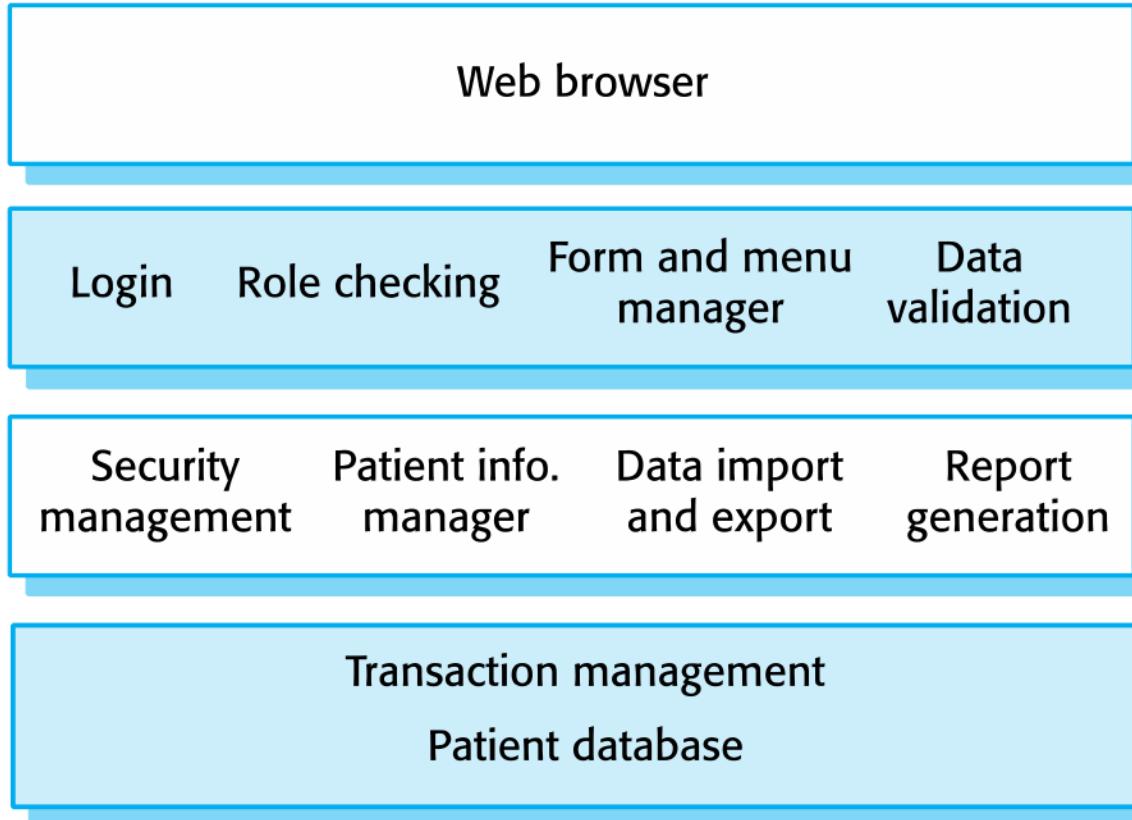


# Layered information system architecture





# The architecture of the Mentcare system





# 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 (styles)** 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



## Chapter 8 – Software Testing

Ian Sommerville,

*Software Engineering*, 10<sup>th</sup> Edition

Pearson Education, Addison-Wesley

Note: These are a slightly modified version of Chapter 8 slides available from the author's site <http://iansommerville.com/software-engineering-book/>



# 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.

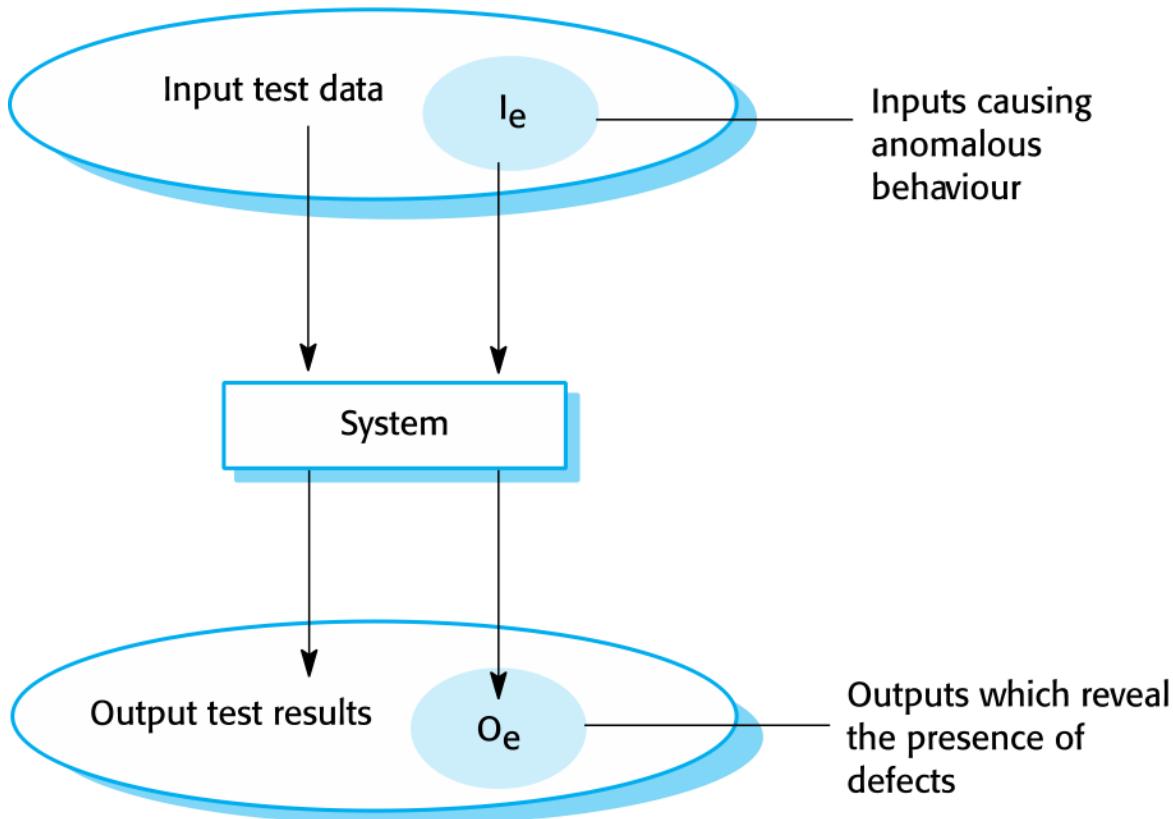


# Validation and defect testing

- ◊ The first goal leads to **validation** testing
  - To demonstrate to the developer and the customer that the software meets its requirements.
  - You expect the system to perform correctly using a given set of test cases that reflect the system's expected use.
  - A successful test shows that the system operates as intended.
- ◊ The second goal leads to **defect** testing
  - To discover situations in which the behavior of the software is incorrect, undesirable or does not conform to its specification.
  - The test cases are designed to expose defects.
  - 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

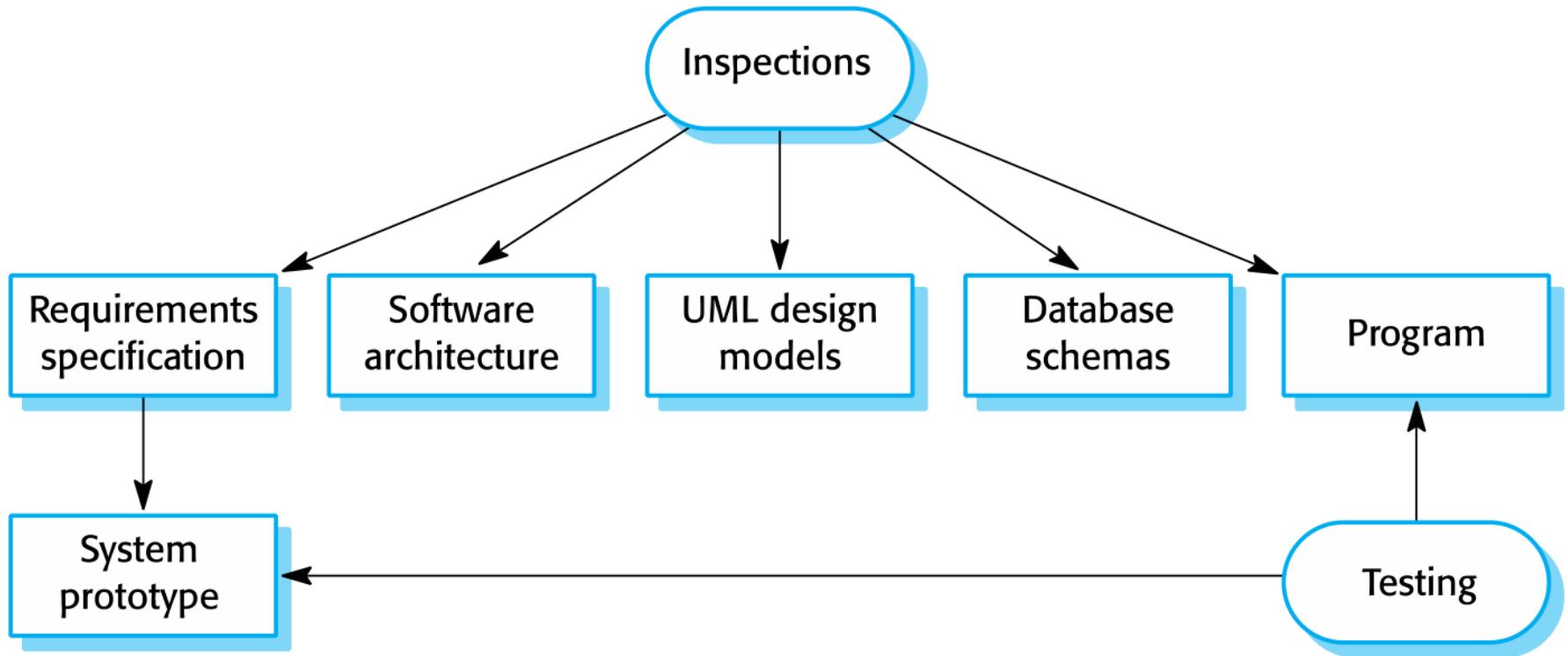
- ◊ Verification:  
"Are we building the product right".
- ◊ The software should conform to its specification.
- ◊ Validation:  
"Are we building the right product".
- ◊ The software should do what the user really requires.



# 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.

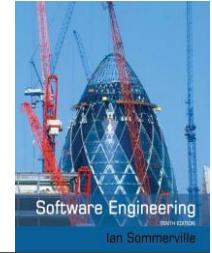
# Inspections and testing





# Advantages of inspections

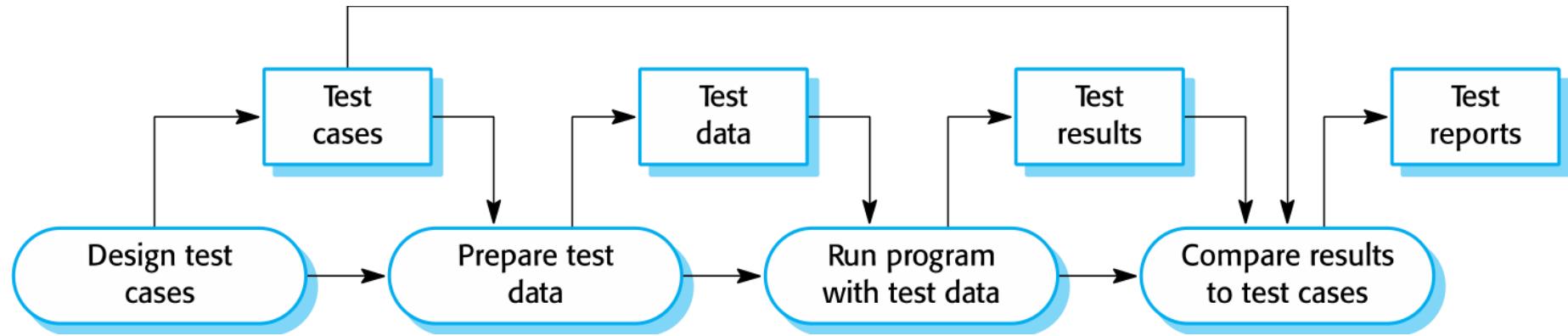
- ◊ During testing, errors can mask (hide) other errors. Because inspection is a static process, you don't have to be concerned with interactions between errors.
- ◊ Incomplete versions of a system can be inspected without additional costs. If a program is incomplete, then you need to develop specialized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

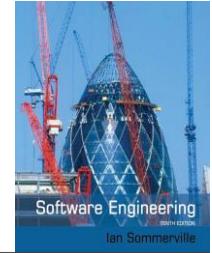


# Inspections and testing

- ◊ 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.

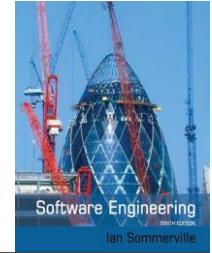
## Development testing





# 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)

◊ 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

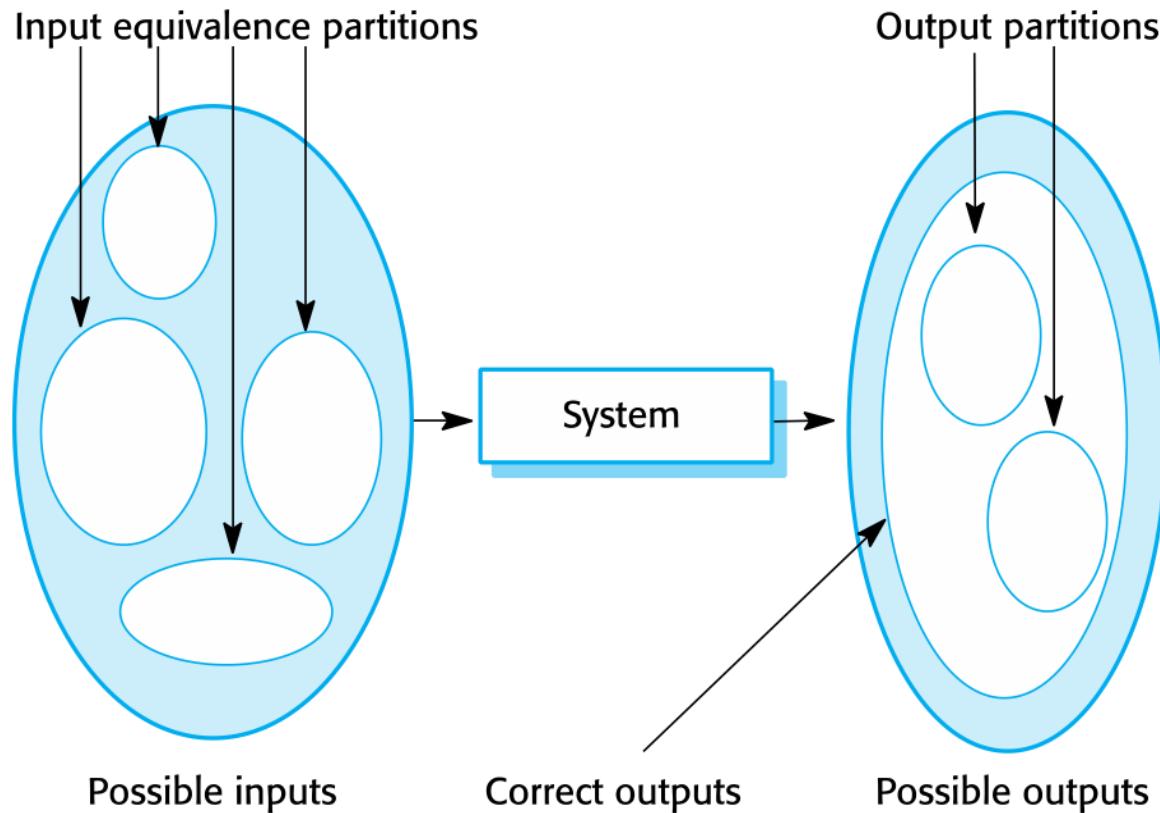
- ◊ Partition testing, where you identify groups of inputs that have common characteristics and should be processed in the same way.
  - You should choose tests from within each of these groups.
- ◊ Guideline-based testing, where you use testing guidelines to choose test cases.
  - These guidelines reflect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

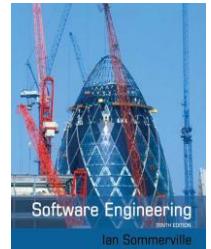


# Partition testing

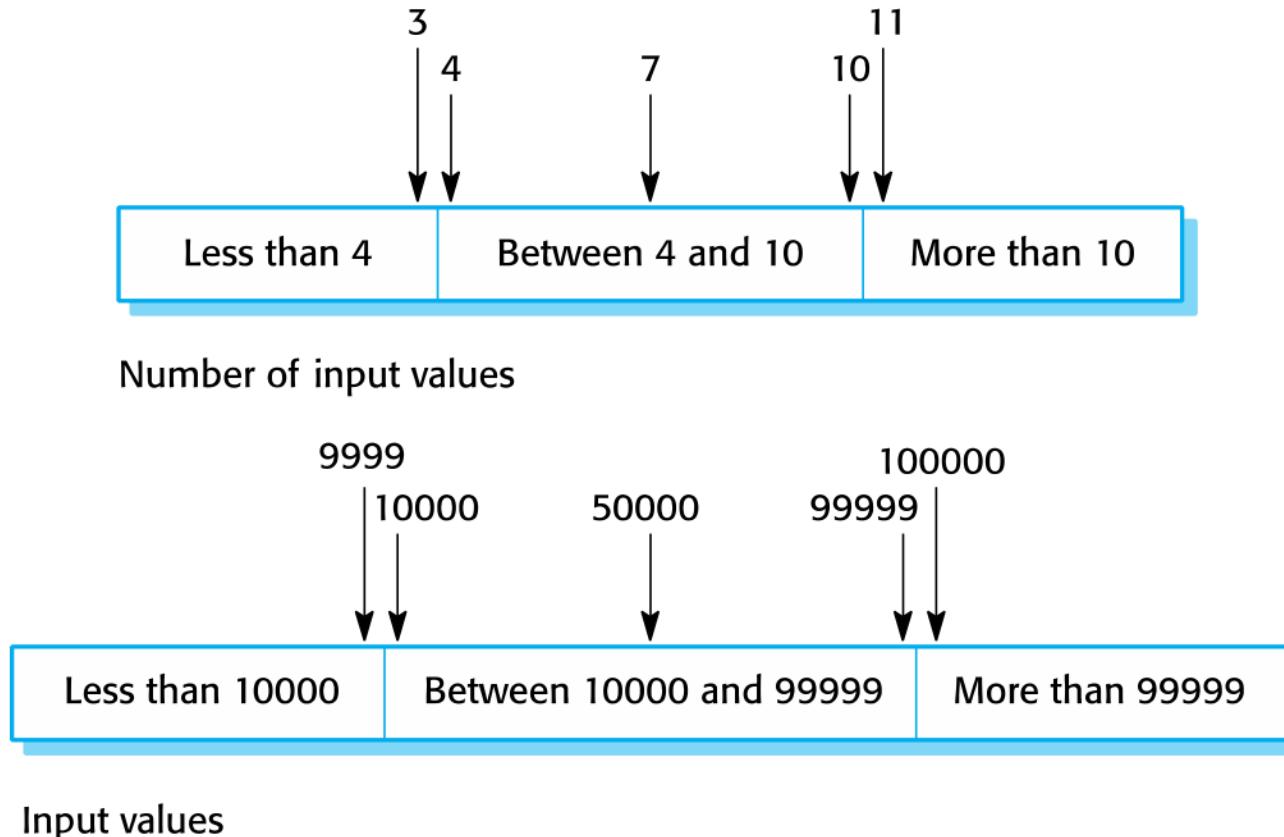
- ◊ Input data and output results often fall into different classes where all members of a class are related.
- ◊ Each of these classes is an equivalence partition or domain where the program behaves in an equivalent way for each class member.
- ◊ Test cases should be chosen from each partition.

# Equivalence partitioning





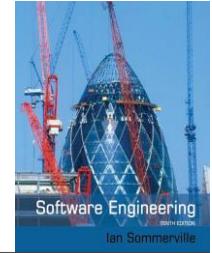
# Equivalence partitions





# 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.



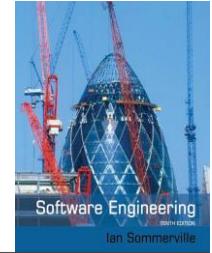
# General testing guidelines

- ◊ 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.

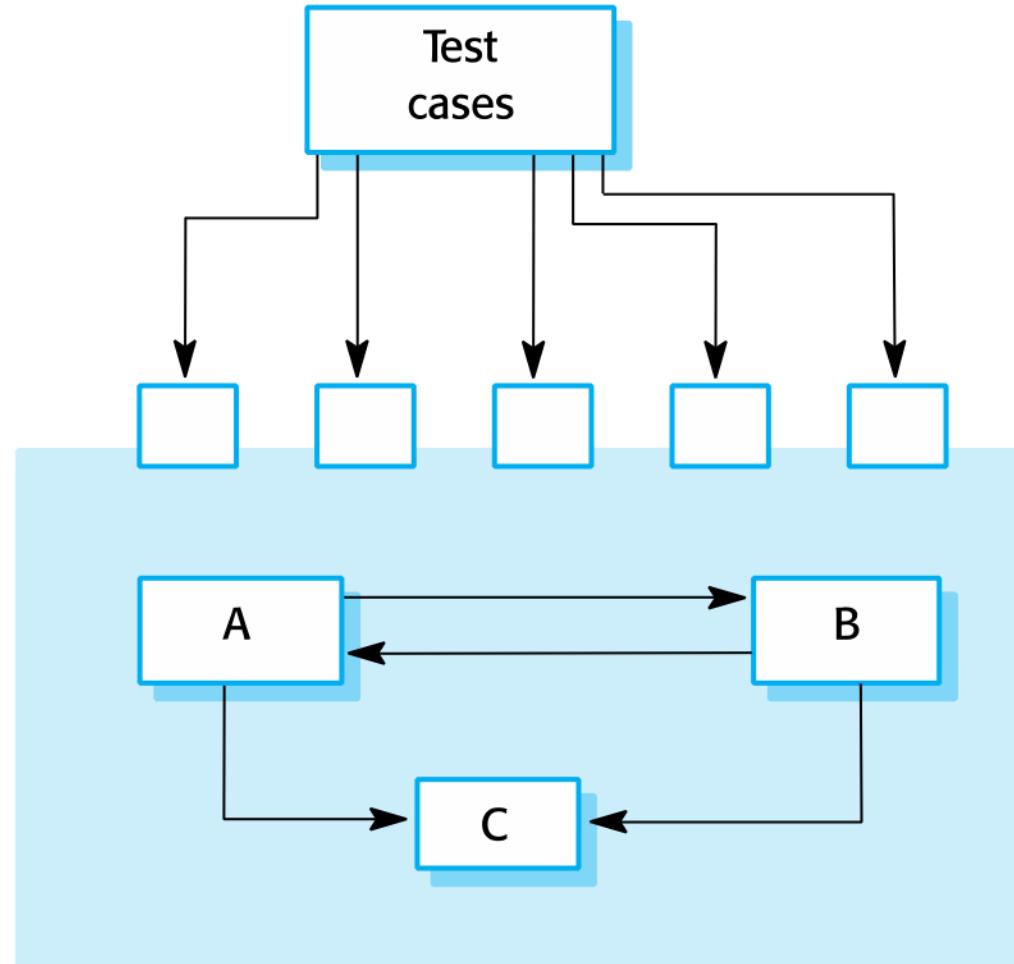


# Interface testing

- ◊ 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 behavior 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.



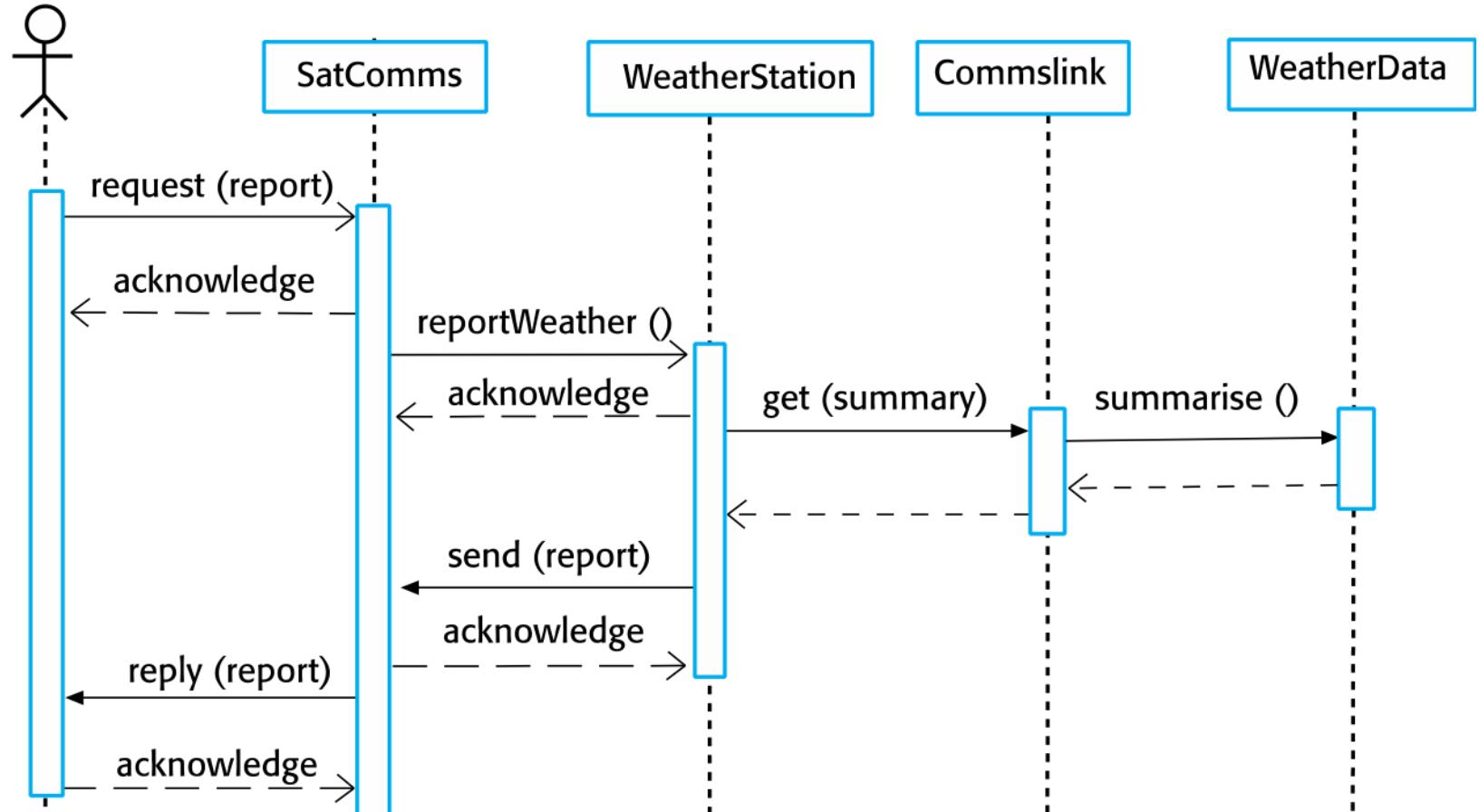
# Use-case testing

- ◊ 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





# 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.

## Component Testing

- Interface Testing

Unit Test

Unit Test

## Component Testing

- Interface Testing

Unit Test

Unit Test

## Component Testing

- Interface Testing

Unit Test

Unit Test

# System Testing

- Use Case Testing

## Unit Test

- Object class testing
- Automated testing
- Partition Testing

# Test-driven development

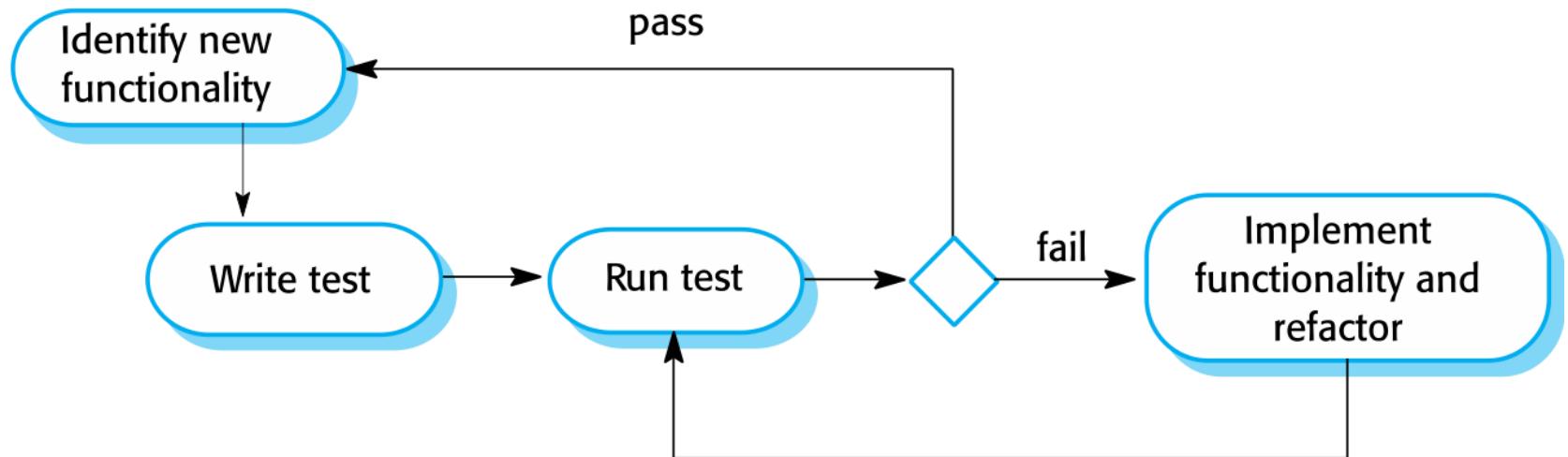




# Test-driven development

- ◊ 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





# 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.
- ◊ Mentcare 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 behavior.

## User testing



Bug Bash by Hans Bjordahl

Copyright 2005 Hans Bjordahl

<http://www.bugbash.net/>



# User testing

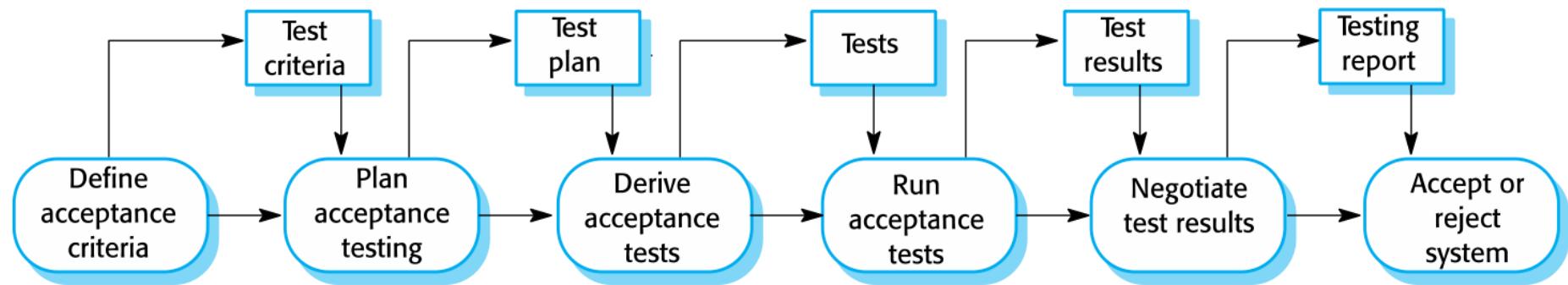
- ◊ User or customer testing is a stage in the testing process in which users or customers provide input and advice on system testing.
- ◊ User testing is essential, even when comprehensive system and release testing have been carried out.
  - The reason for this is that influences from the user's working 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.

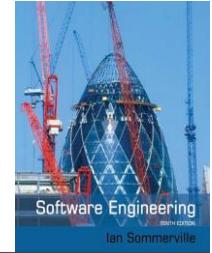
# The acceptance testing process



# 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.



# Professional Tip of the Day: Important Email

## ◇ How to send an important email:

- Send it as high priority
- If they are on vacation send it the day they get back so it is at the top of their inbox
- Put the words “IMPORTANT” or “EMERGENCY” in the email subject line
- Follow up with a phone call or Message

# CS 425

## UML Activity Diagrams & State Charts

November 2, 2023

From Chapters 14, 21, and 22 [Jim Arlow and Ila Neustadt, UML 2 and the Unified Process, Addison-Wesley 2005]



University of Nevada, Reno  
Department of Computer Science & Engineering

# Outline

## Part 1 - Activity diagrams

- Introduction
- Activities
- Nodes
  - Action nodes
  - Control nodes
  - Object nodes
- Activity parameters

# Introduction: What are activity diagrams?

## *Activity diagrams:*

- A form of “object-oriented flowcharts”
- Attached to modeling elements to describe behavior
- Typically attached to use cases, classes, operations, components, and interfaces
- Can also be used to model business processes and workflows

# Introduction: Where are activity diagrams used?

## Commonly used in:

- Analysis
  - To model the flow of a use case
  - To model the flow between use cases
- Design
  - To model details of an operation
  - To model details of an algorithm
- Business modeling
  - To model a business process
- As always in modeling, it is important to keep them **simple** and **understandable** by their intended audience

# Activities \*\*\*\*

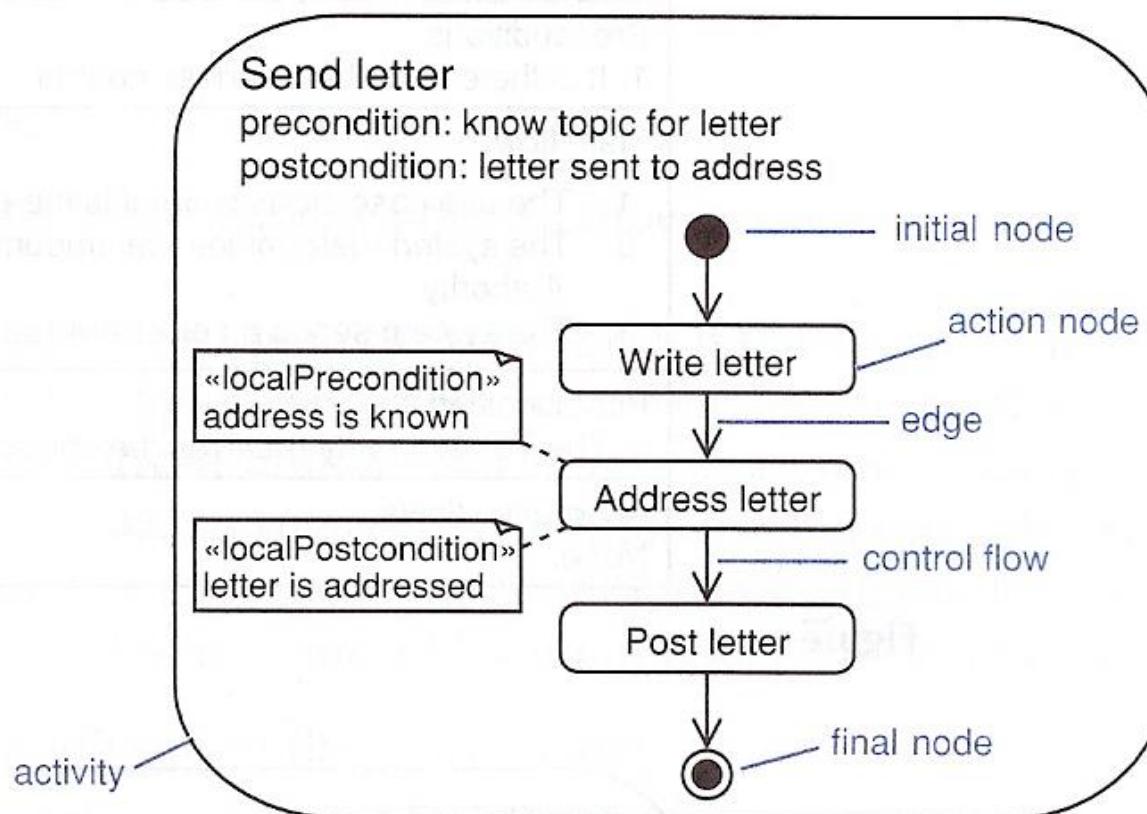
- *Activity diagrams* are networks of *nodes* connected by *edges*
- Nodes
  - Action nodes – atomic units of work within the activity
  - Control nodes – control the flow through the activity
  - Object nodes – represent objects used in the activity
- Edges
  - Control flows – depict the flow of control through activity
  - Object flows – depict the flow of objects through activity

# \* Activities \*\*\*\*

- *Activities* and *actions* can have pre- and post-conditions
- *Tokens* (part of semantics but not shown graphically) abstractly flow in the network and can represent:
  - The flow of control
  - An object
  - Some data
- A token moves from a source node to a target node across an edge depending on:
  - Source node post-conditions
  - Edge guard conditions
  - Target node preconditions

# \*\* Activities \*\*\*

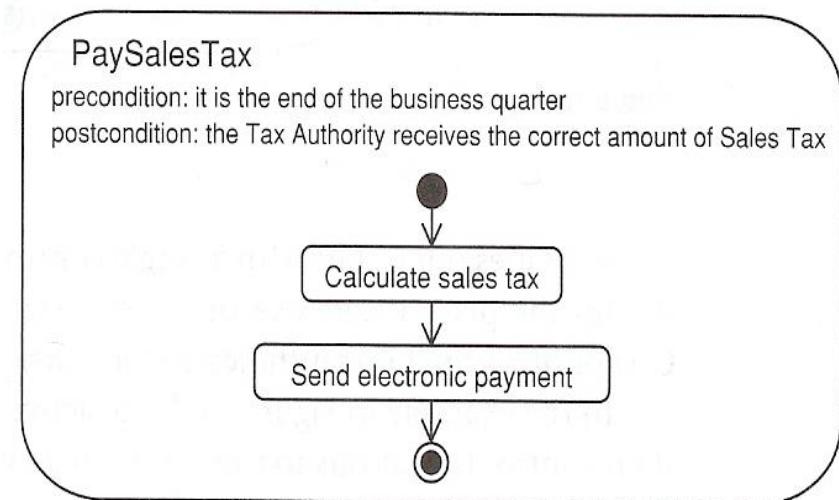
- Example of an activity (“send letter”), Fig. 14.2 [Arlow & Neustadt 2005]



# \*\*\* Activities \*\*

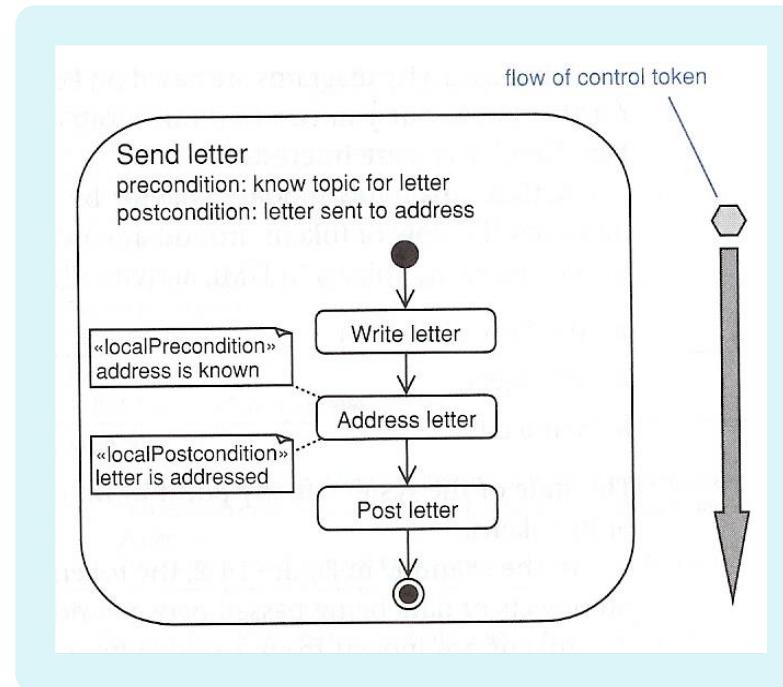
- Activity diagrams can model **use cases** as a series of actions.

Use case: PaySalesTax
ID: 1
Brief description: Pay Sales Tax to the Tax Authority at the end of the business quarter.
Primary actors: Time
Secondary actors: TaxAuthority
Preconditions: 1. It is the end of the business quarter.
Main flow: <ol style="list-style-type: none"><li>The use case starts when it is the end of the business quarter.</li><li>The system determines the amount of Sales Tax owed to the Tax Authority.</li><li>The system sends an electronic payment to the Tax Authority.</li></ol>
Postconditions: 1. The Tax Authority receives the correct amount of Sales Tax.
Alternative flows: None.



# \*\*\*\* Activities \*

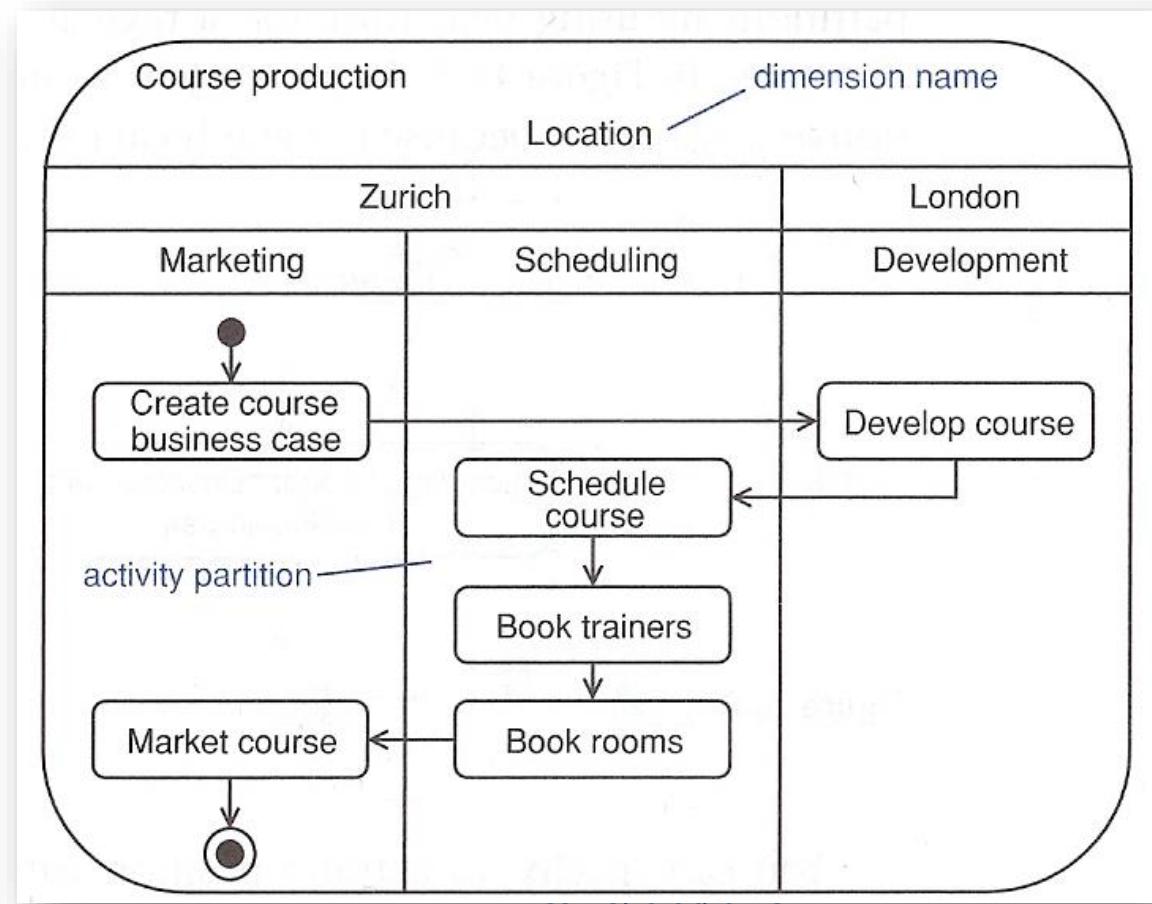
- Activity diagrams have semantics based on Petri Nets
- They model behavior using the token game
- Tokens move through the network subject to conditions
- Object nodes represent objects flowing around the system
- Example of flow of control token



Activity diagrams

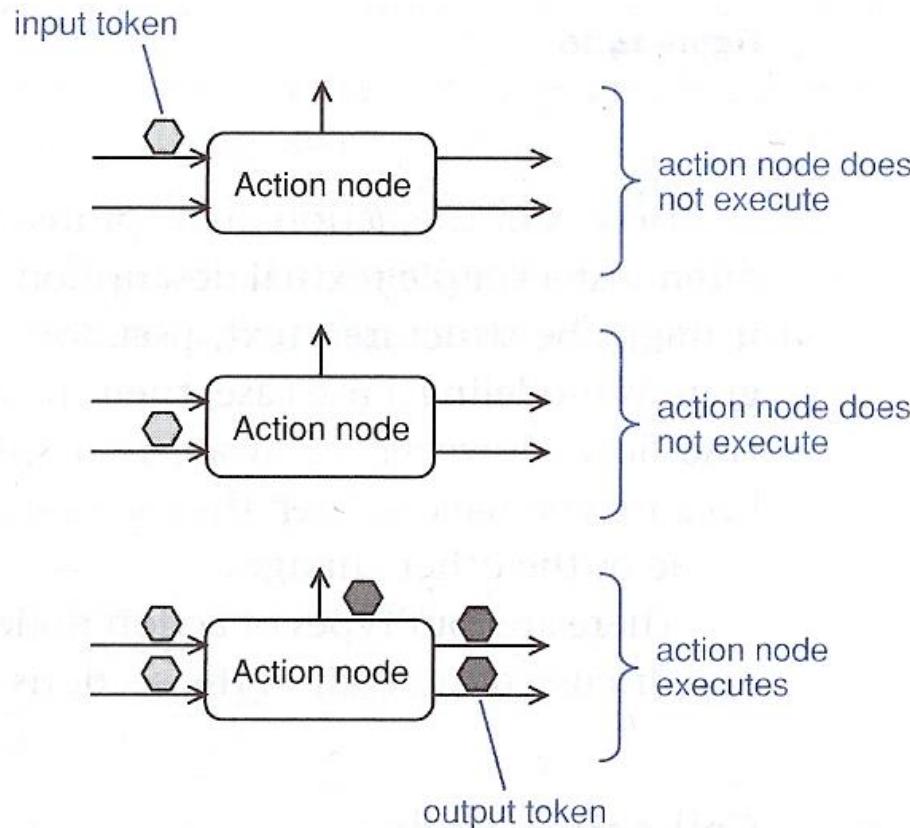
# \*\*\*\*\* Activities

- Activity diagrams can be divided in *partitions (swimlanes)* using vertical, horizontal, or curved lines.



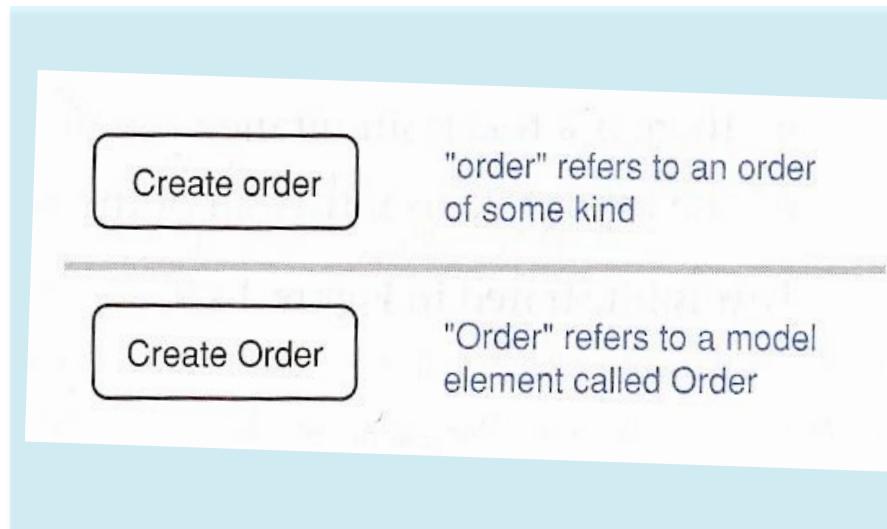
# Action nodes \*\*\*

- Action nodes execute when:
  - There are tokens present at all their input nodes AND
  - The input tokens satisfy all action node's local preconditions



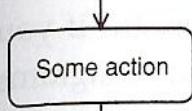
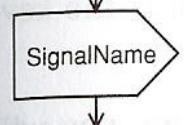
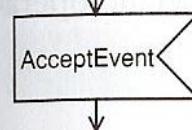
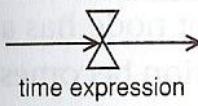
# \* Action nodes \*\*\*

- After execution, the local post-conditions are checked; if all are satisfied, the node simultaneously offers tokens to all its output edges (this is an **implicit fork** that may give rise to many flows)



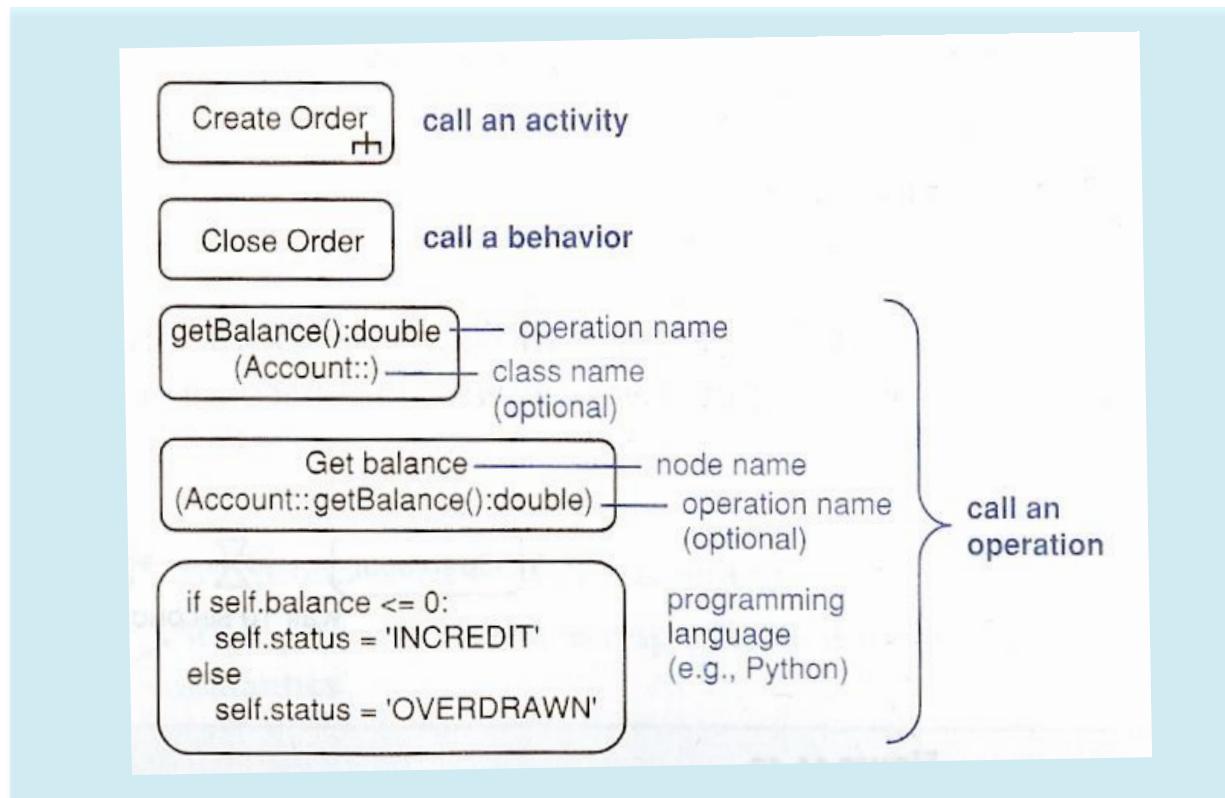
# \*\* Action nodes \*\*

- Types of *action nodes*, Table. 14.1 [Arlow & Neustadt 2005]

Syntax	Name	Semantics	Section
	Call action node	Invokes an activity, behavior, or operation	14.7.1
	Send signal	Send signal action – sends a signal asynchronously (the sender <i>does not</i> wait for confirmation of signal receipt) It may accept input parameters to create the signal	15.6
	Accept event action node	Accepts an event – waits for events detected by its owning object and offers the event on its output edge Is enabled when it gets a token on its input edge If there is <i>no</i> input edge, it starts when its containing activity starts and is always enabled	15.6
	Accept time event action node	Accepts a time event – responds to time Generates time events according to its time expression	14.7.2

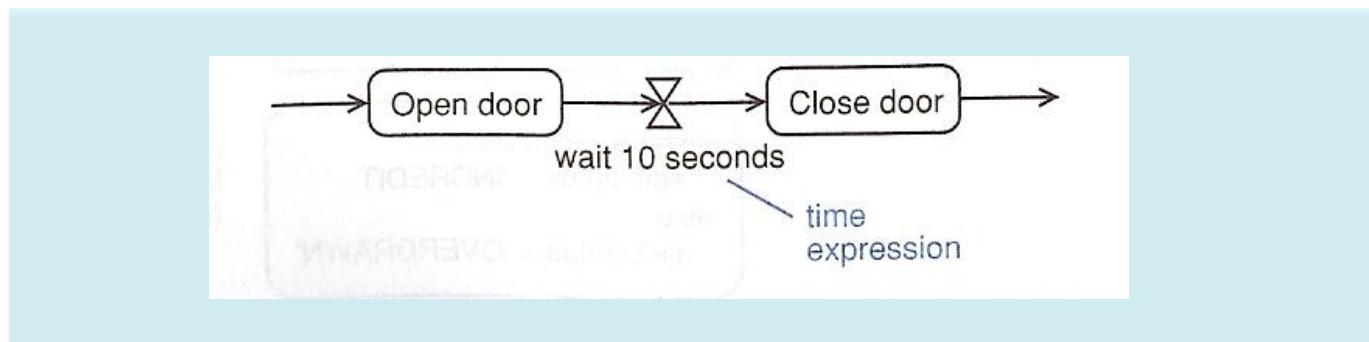
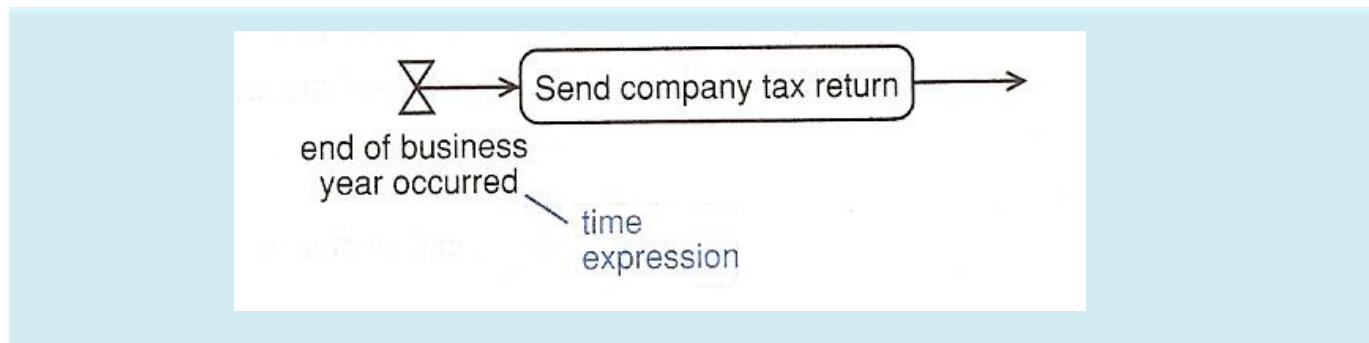
# \*\*\* Action nodes \*

- A *call action node* invokes an activity, behavior, or operation



# \*\*\*\* Action nodes

- An *accept time event* action node responds to time



# Control nodes \*\*

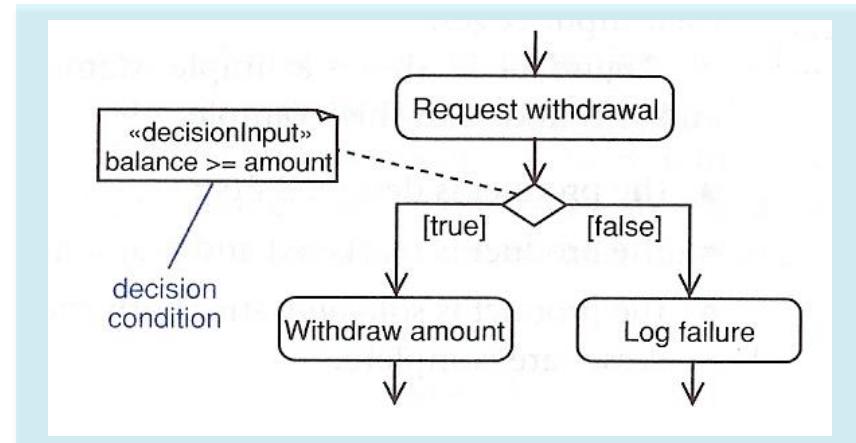
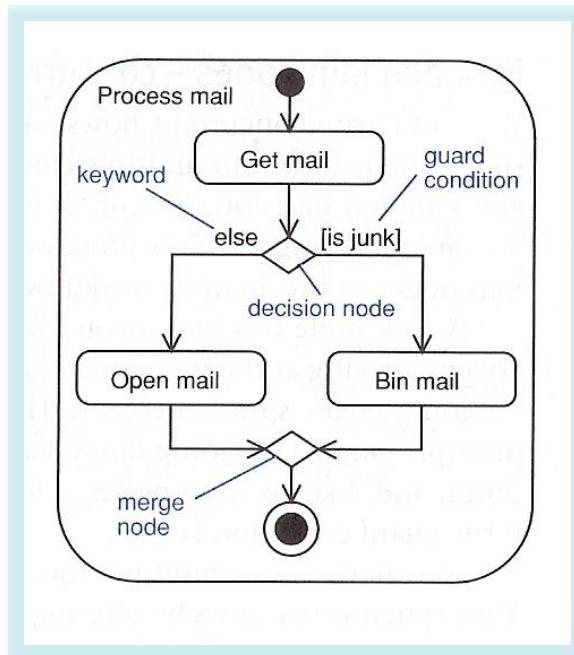
*Control nodes* manage the flow of control within an activity

Table 14.2 [Arlow & Neustadt 2005] shows the types of control nodes

Syntax	Name	Semantics	Section
	Initial node	Indicates where the flow starts when an activity is invoked	14.8.1
	Activity final node	Terminates an activity	14.8.1
	Flow final node	Terminates a specific flow within an activity – the other flows are unaffected	14.8.1
	Decision node	The output edge whose guard condition is true is traversed May optionally have a «decisionInput»	14.8.2
	Merge node	Copies input tokens to its single output edge	14.8.2
	Fork node	Splits the flow into multiple concurrent flows	14.8.3
	Join node	Synchronizes multiple concurrent flows May optionally have a join specification to modify its semantics	14.8.3

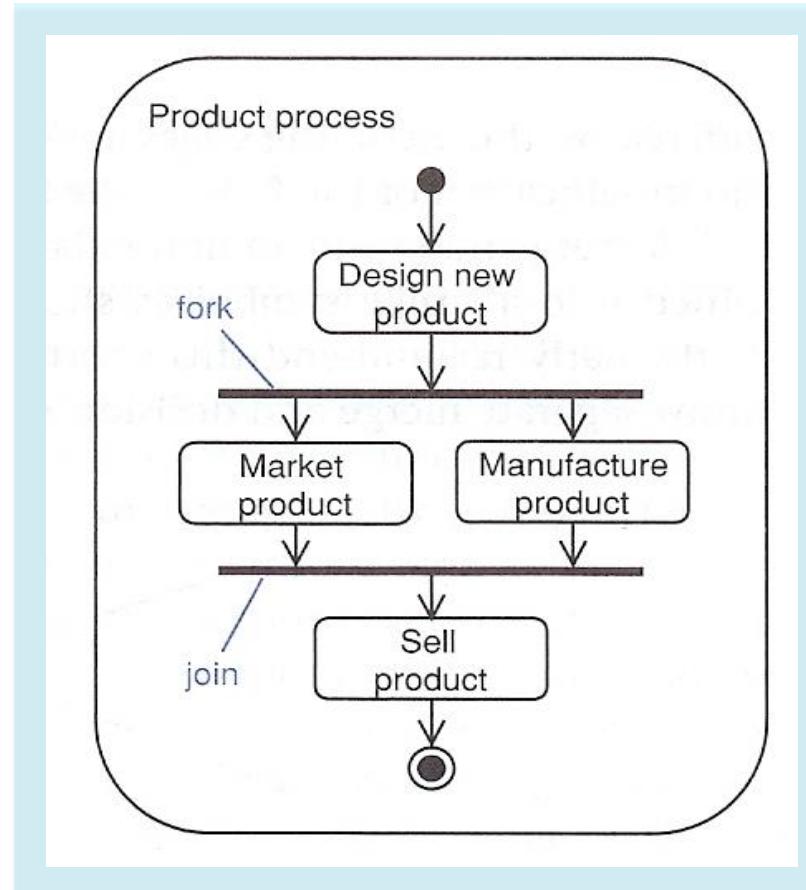
# \* Control nodes \*

- Examples of *decision* and *merge* nodes



# \*\* Control nodes

- Examples of *join* and *fork* nodes

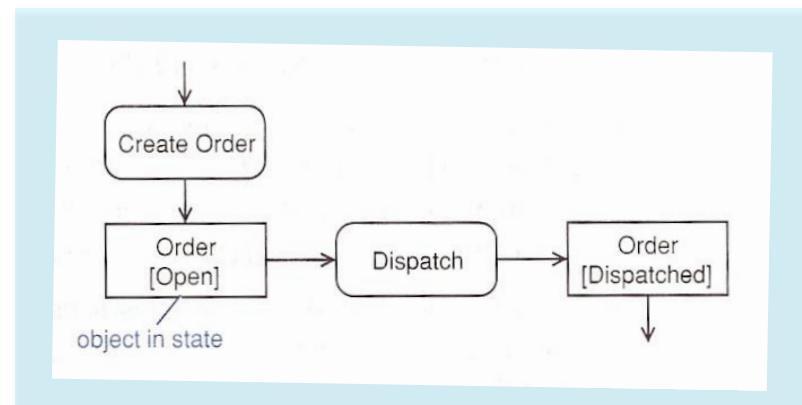
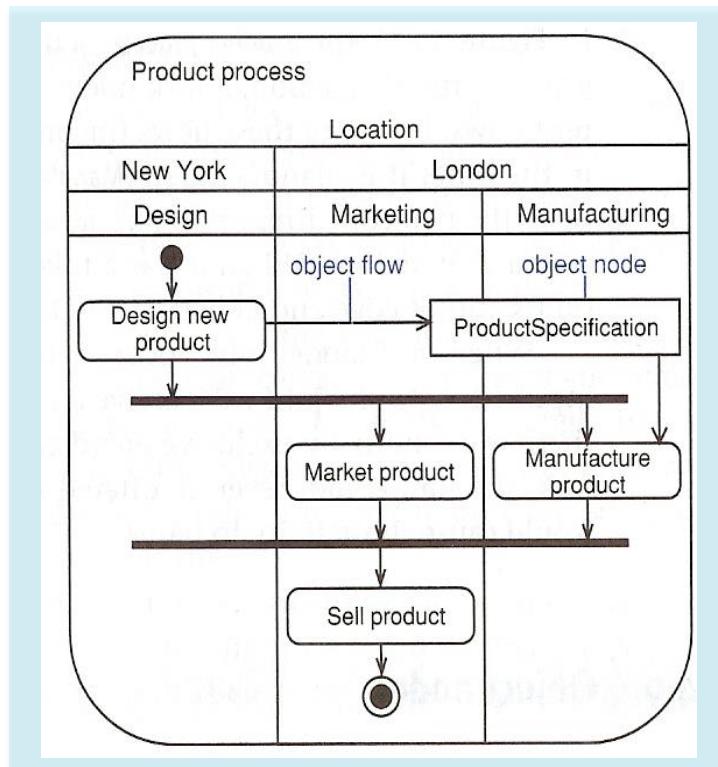


# Object nodes \*

- *Object nodes* indicate that instances of a particular classifier are available at a specific point in the activity
- They are labeled with the name of the classifier and represent instances of that classifier or its subclasses
- The input and output edges are *object flows*
- The objects are created and consumed by action nodes
- When an object node receives an object token on one of its input edges, it offers this token to all its output edges, which compete for the token.

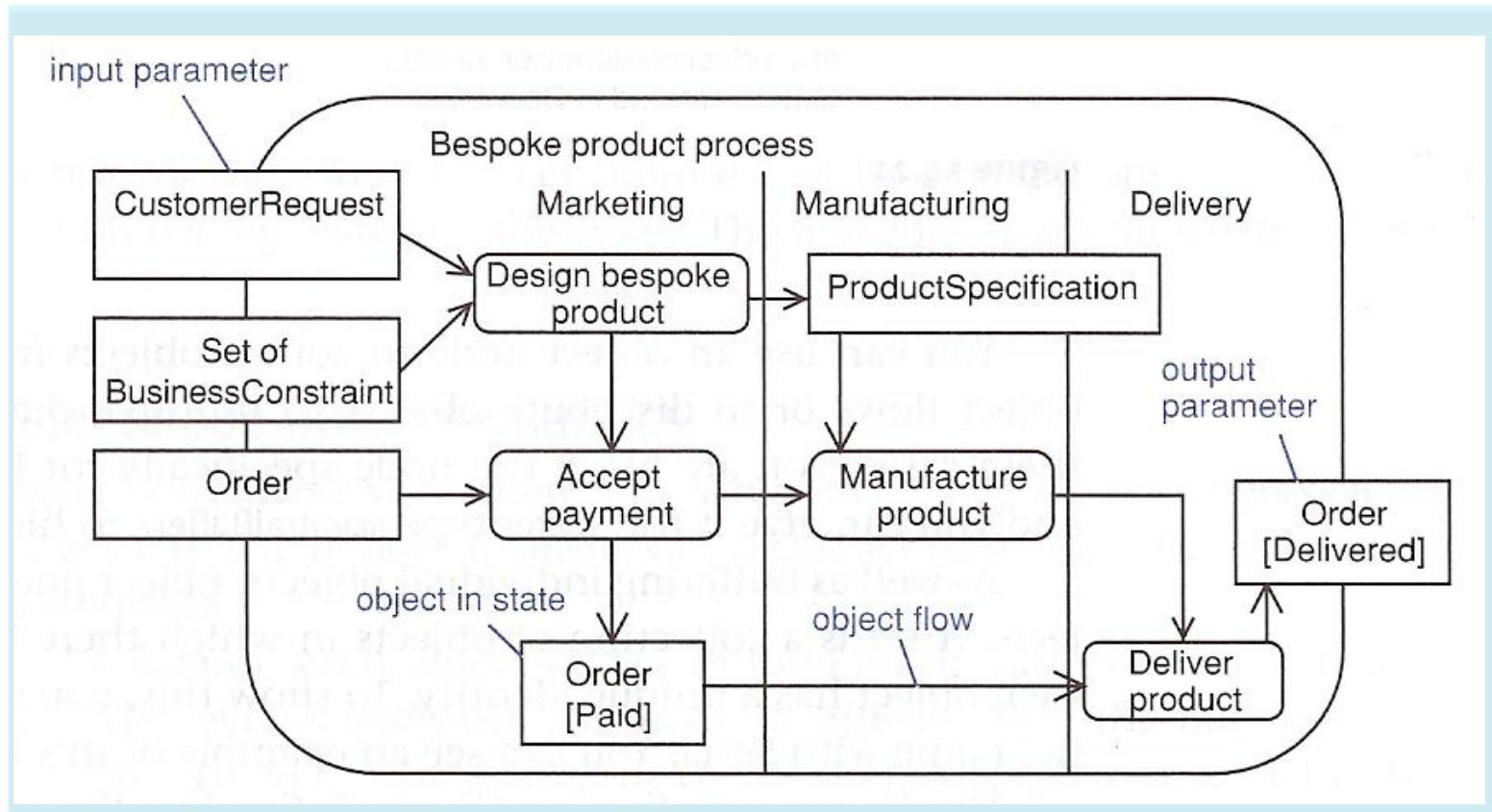
# \* Object nodes

- Examples of using object nodes. Note that object nodes can represent objects in **particular states**.



# Activity parameters

- Activities can have object nodes to provide **inputs** and **outputs**



## Part 2 – State Charts

From Chapter 21: State Machines &  
Chapter 22: Advanced State Machines (partial)

[Arlow and Neustadt 2005]

# Outline

## ■ State machines

- Introduction
- State machine diagrams
- States
- Transitions
- Events

## ■ Advanced state machines

- Composite states
  - Simple
  - Orthogonal
- History

# Introduction

- Both activity diagrams and state machine diagrams model system behavior
- However, they have different semantics:
  - **Activity diagrams** are based on Petri Nets and usually model processes when several objects participate
  - **State machines** are based on Harel's statecharts and typically used to model single reactive objects

# Introduction

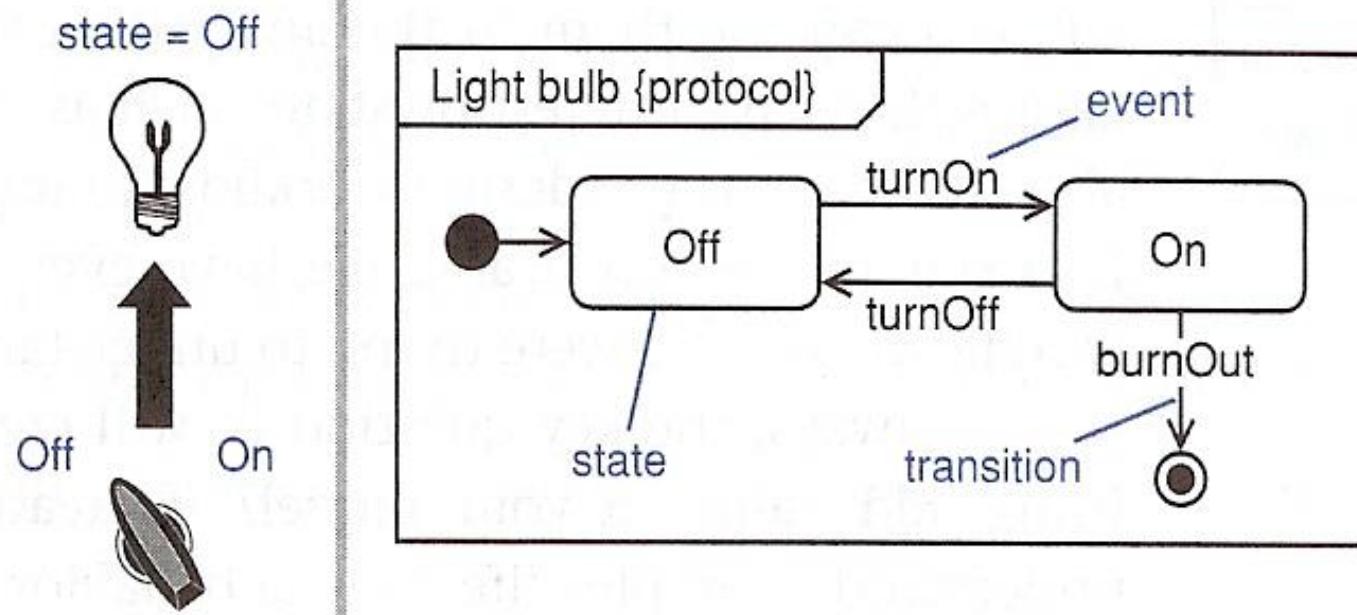
## ■ Reactive objects:

- Respond to external events
- May generate and respond to internal events
- Have a lifecycle modeled as a progression of states, transitions and events
- May have current behavior that depends on past behavior

## ■ State machines are used to model behavior of items such as classes, use cases, subsystems, systems

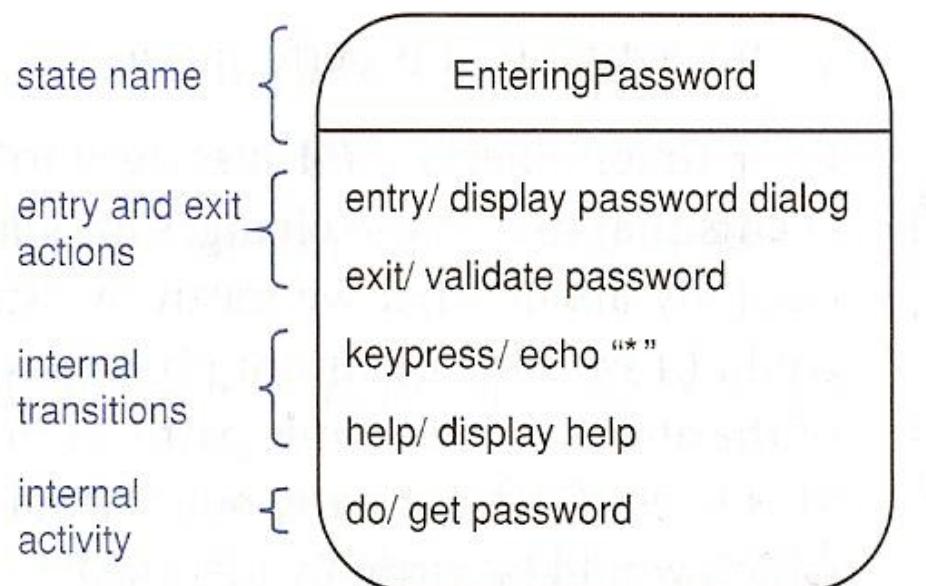
# State machine diagrams

- There are three main modeling elements in state diagrams: *states*, *transitions*, and *events*.
- Example of a simple state machine, Fig. 21.2 [Arlow & Neustadt]



# States

## Summary of UML state syntax

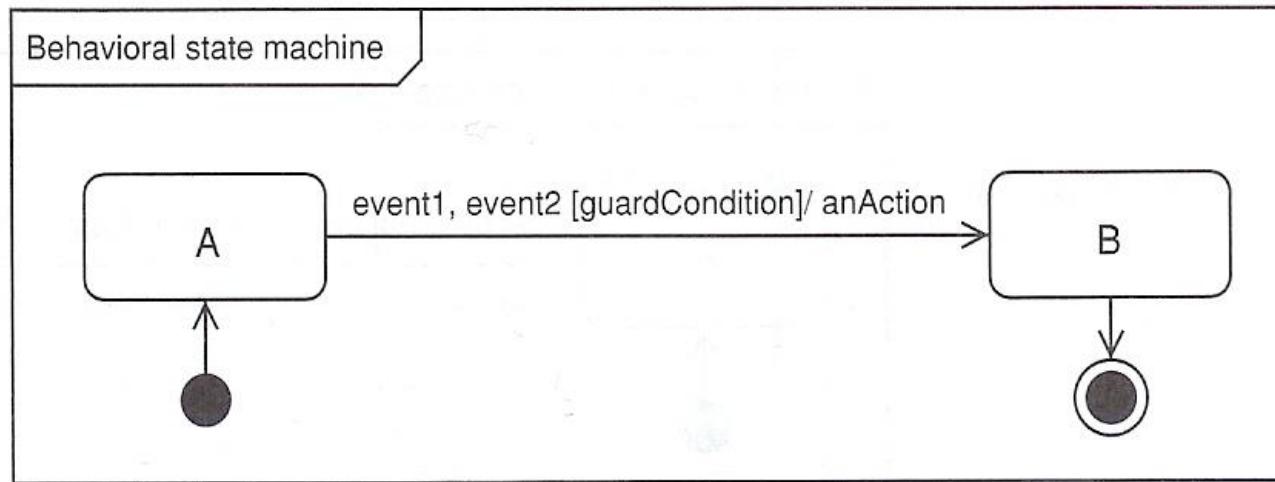


action syntax: eventName/ someAction

activity syntax: do/ someActivity

# Transitions

Summary of UML syntax for transitions in *behavioral state diagrams*, Fig.21.5

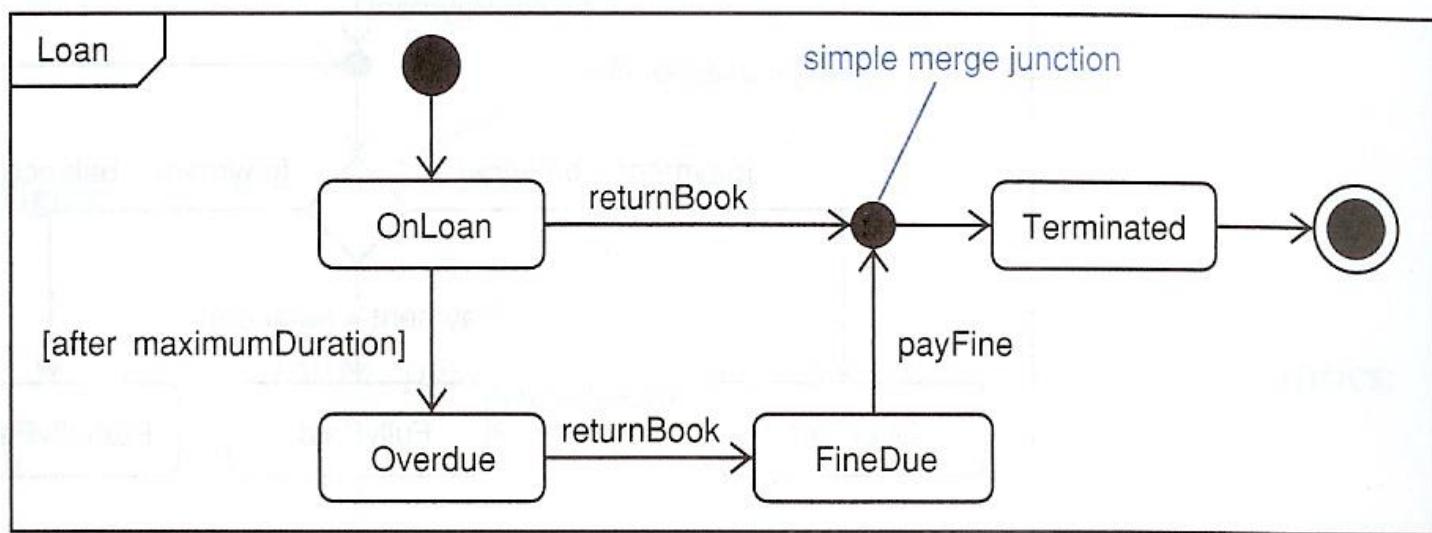


Where:

- **event(s)**= internal or external occurrence(s) that trigger the transition
- **guardCondition** = boolean expression, when *true* the transition is allowed
- **anAction** = some operation that takes place when the transition fires

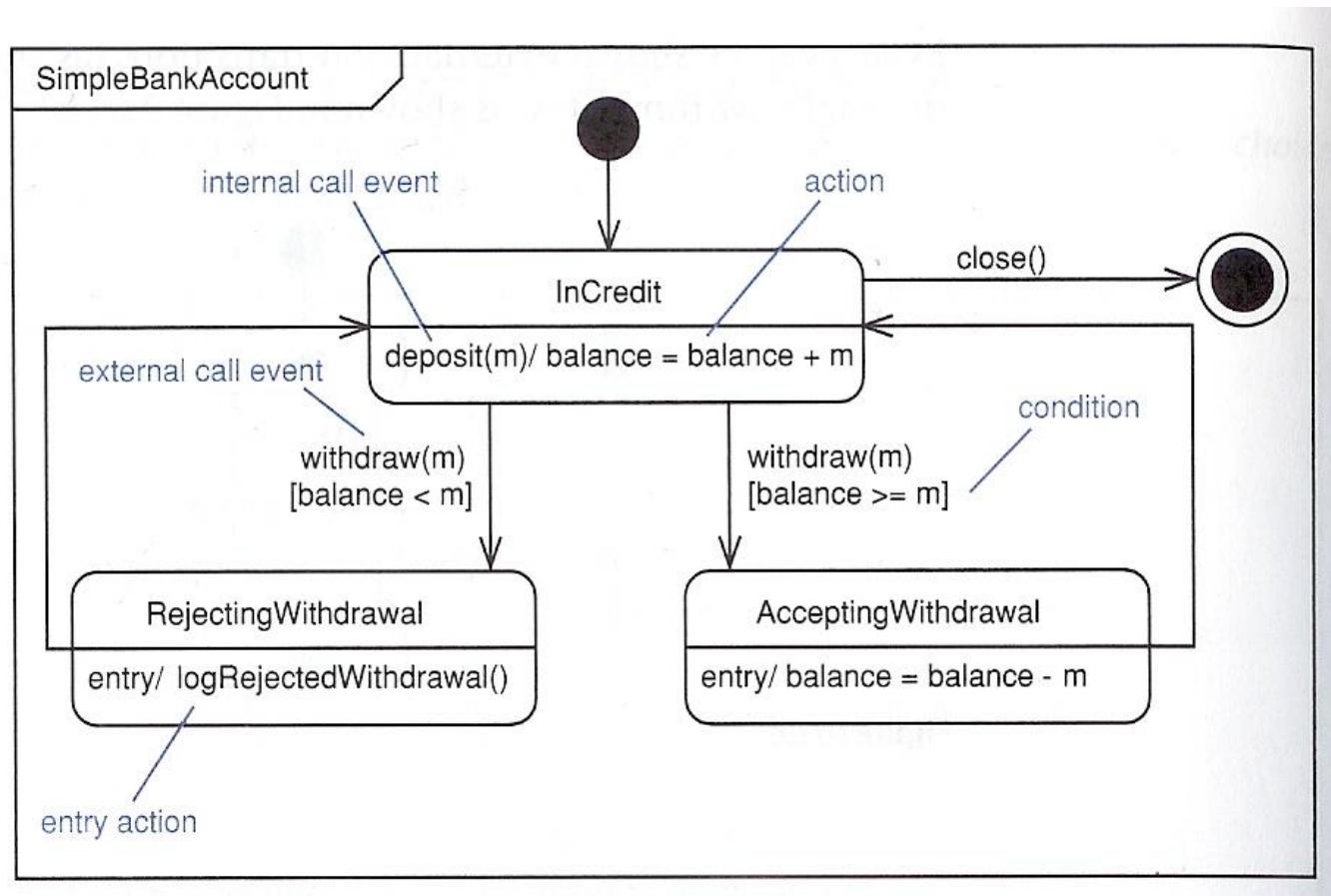
# Transitions

A *junction pseudo-state* represents a point where transitions merge or branch



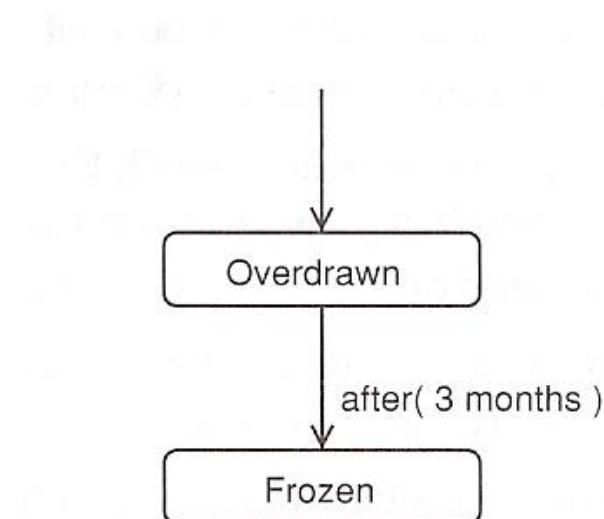
# Events

Example of a call event, Fig.21.11 [Arlow & Neustadt 2005]



# Events

Time events are indicated by the keywords *when* and *after*.  
Example of a time event:



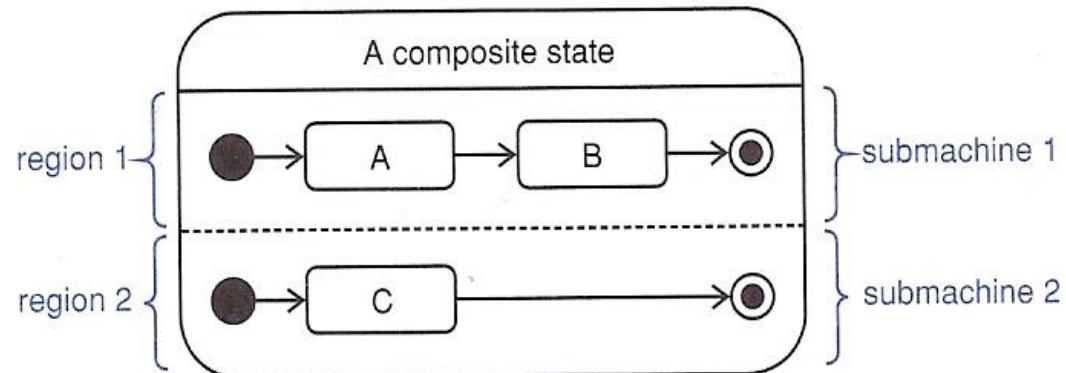
context: CreditAccount class

Example of a  
state machine  
[Dascalu 2001]

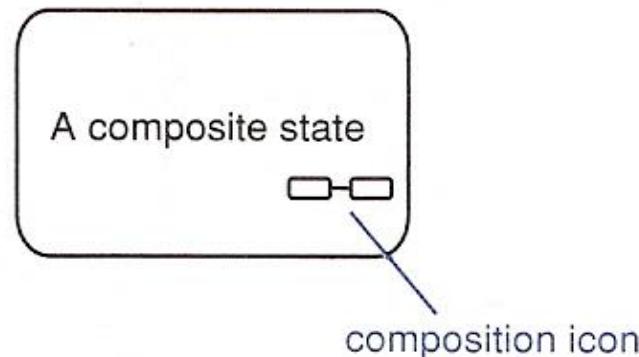
# Composite states

- A *composite state* contains one or more nested state machines (*submachines*), each existing in its own *region*.

Fig 22.2

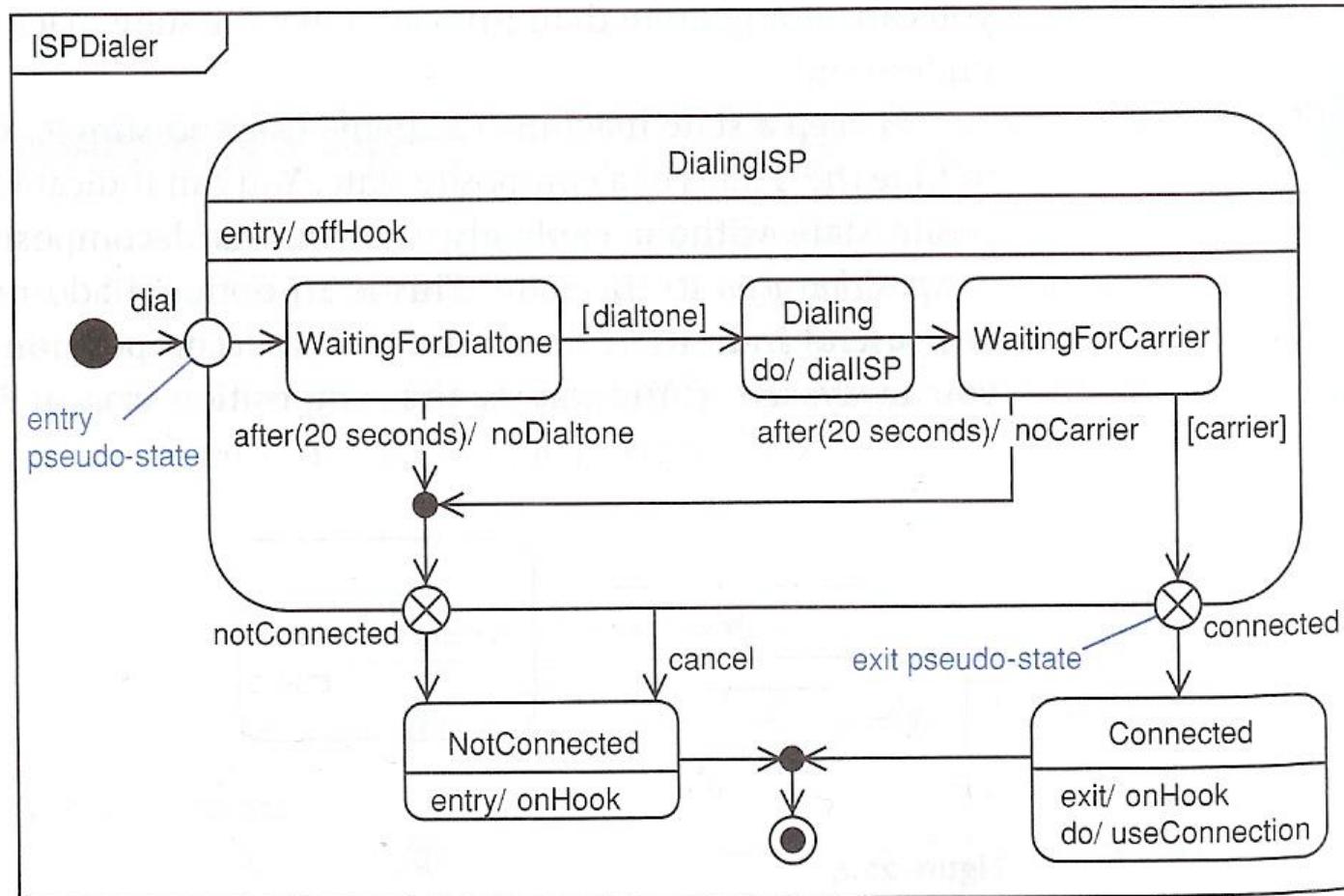


- The composition icon is shown in Fig. 22.4



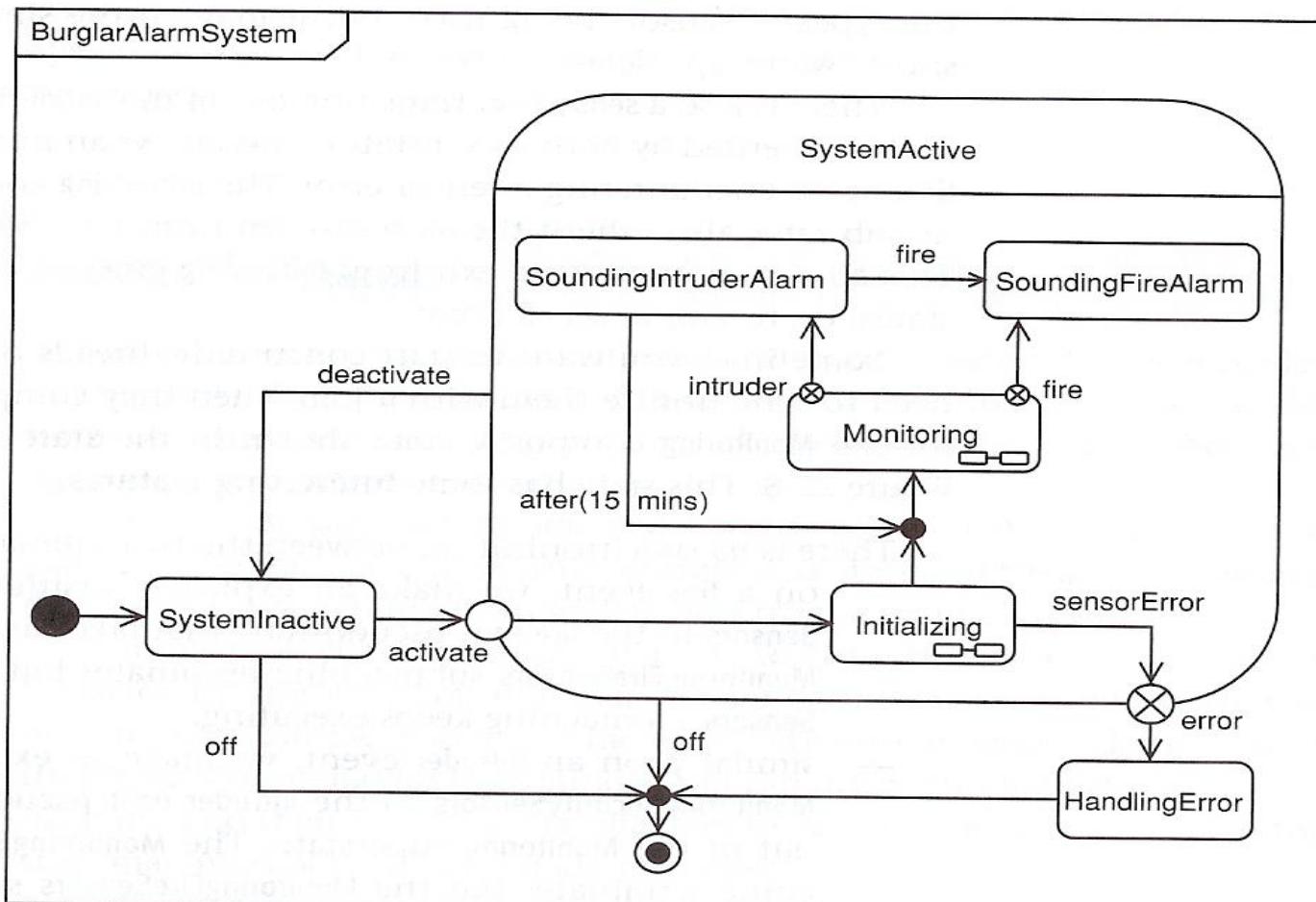
# Simple composite states

- A superstate that contains a single region is called a *simple composite state*,



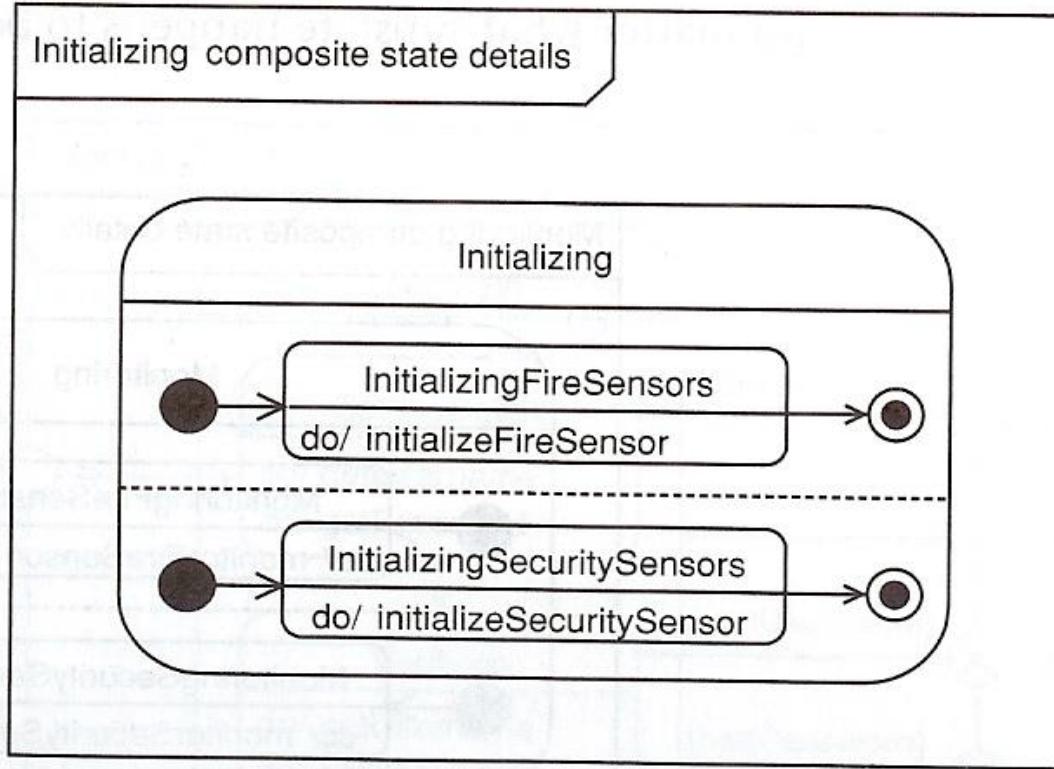
# Orthogonal composite states

- Orthogonal composite states consist of two or more sub-machines that execute in parallel. Composite states below are *Initializing* and *Monitoring*



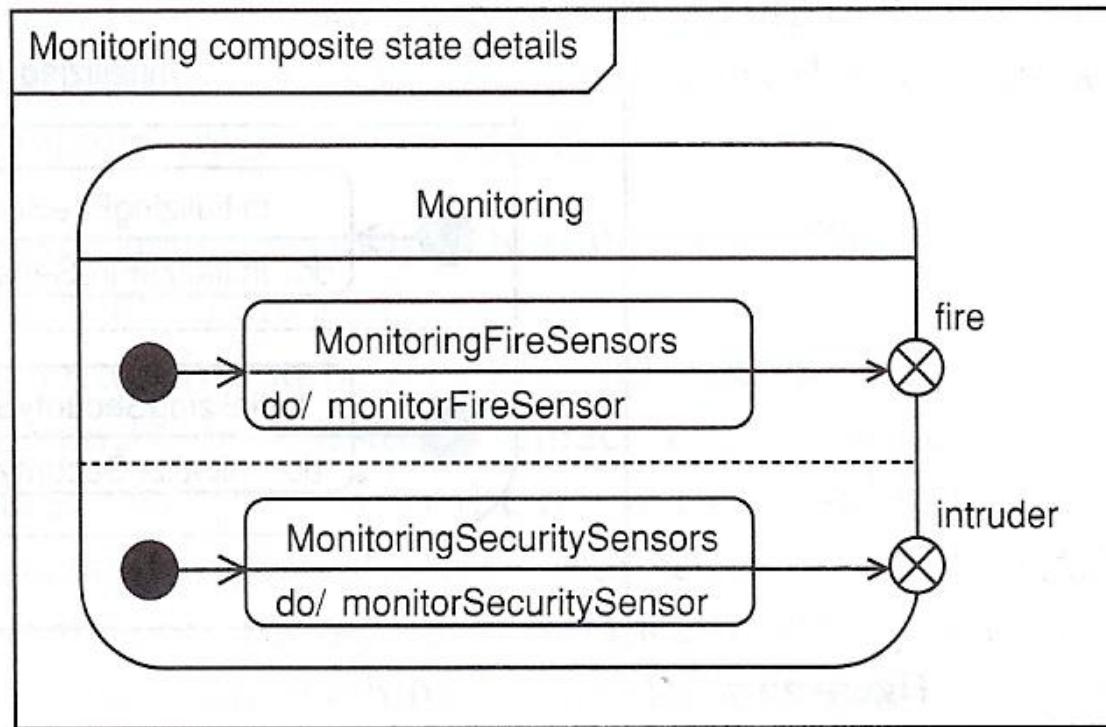
# Orthogonal composite states

The composite state **Initializing**:



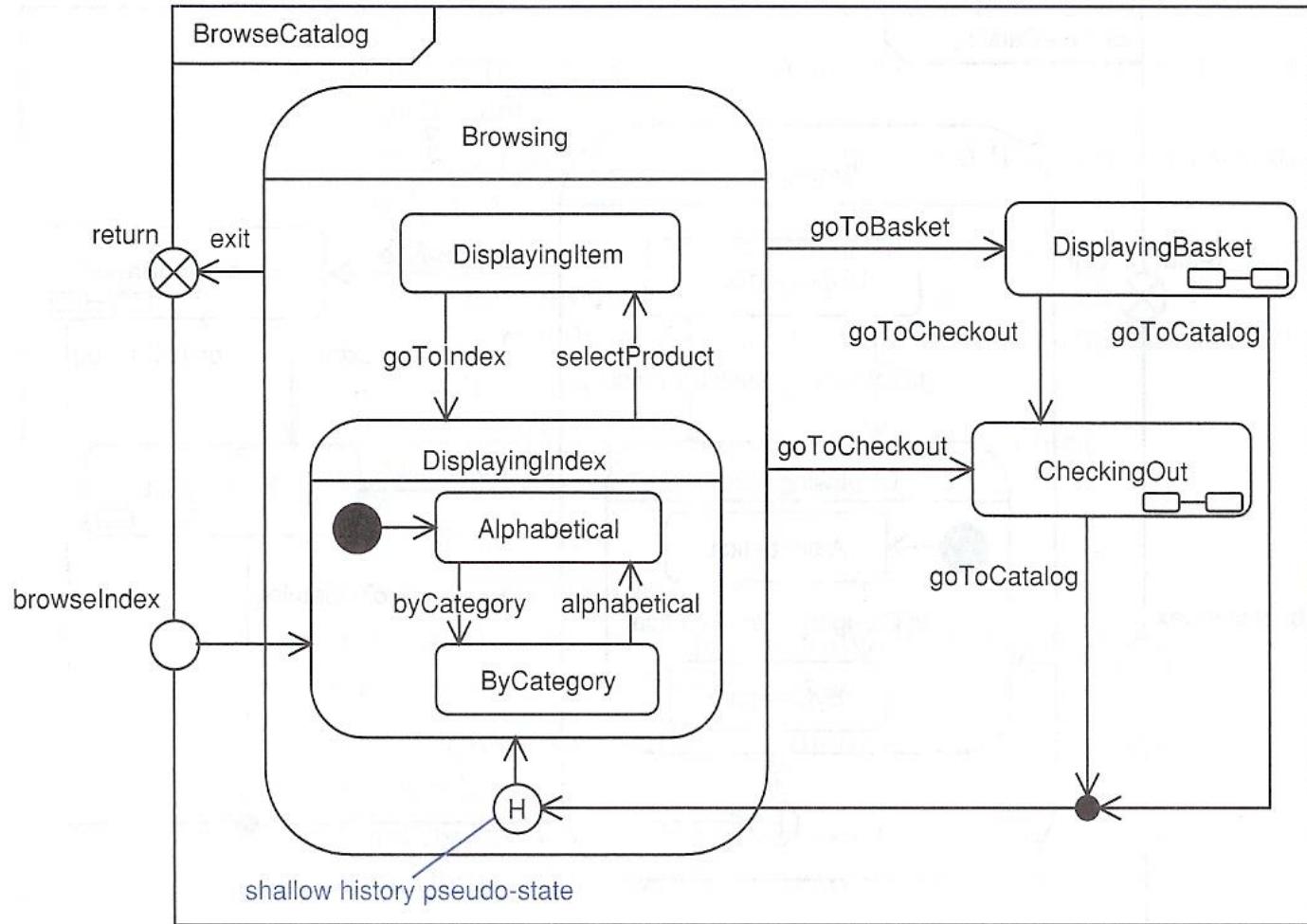
# Orthogonal composite states

## The composite state Monitoring



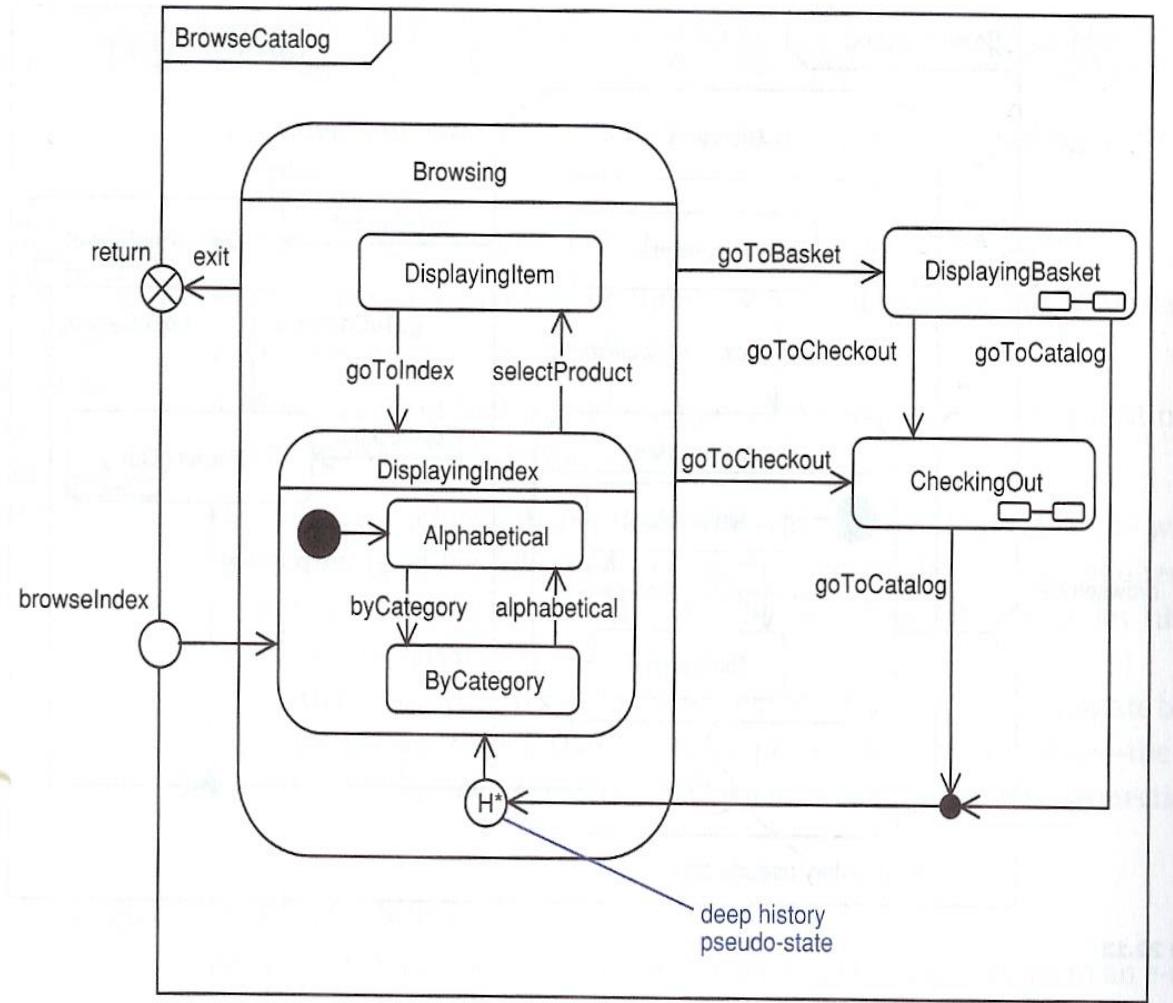
# History

Example of using the *shallow history* indicator



# History

Example of using  
the *deep history*  
indicator





# Chapter 9 – Software Evolution

[Ian Sommerville, Software Engineering, 2015]



# Topics covered

---

- ✧ Evolution processes
- ✧ Legacy systems
- ✧ Software maintenance
  - Re-engineering
  - Refactoring

# Software change

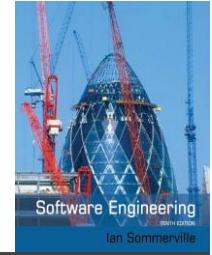


## ❖ Software change is inevitable

- New requirements emerge when the software is used
- The business environment changes
- Errors must be repaired
- New computers and equipment are added to the system
- The performance or reliability of the system may have to be improved

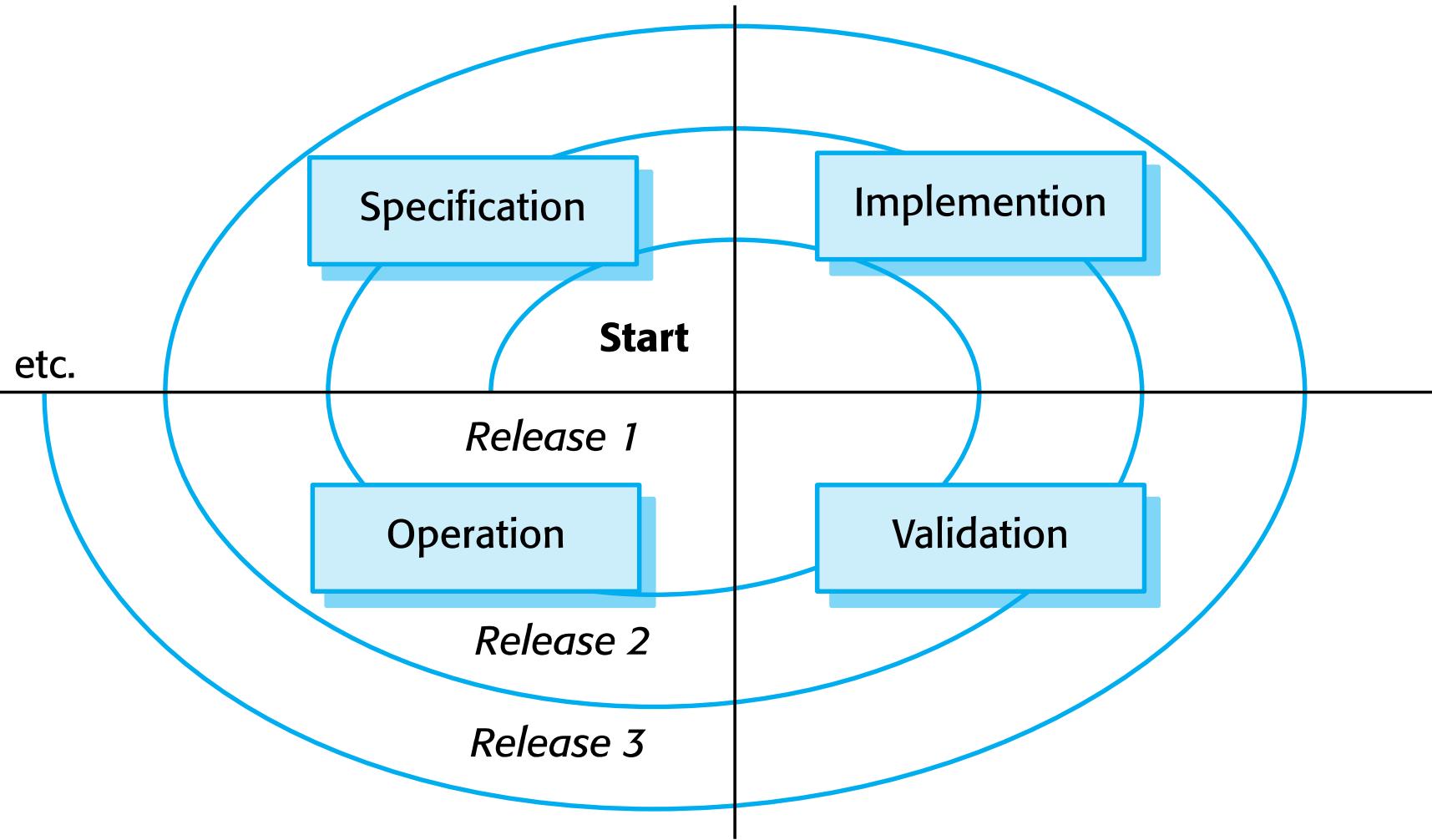
## ❖ A key problem for all organizations is **implementing and managing change** to their existing software systems

# Importance of evolution

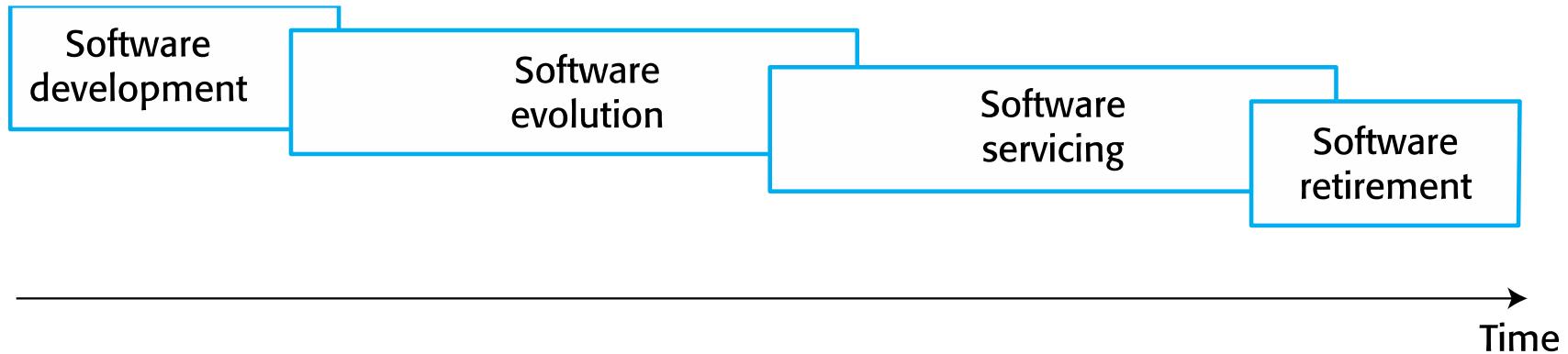
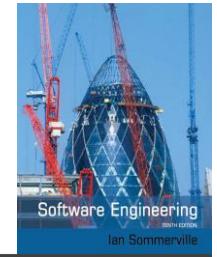


- ✧ Organizations have huge investments in their software systems - they are **critical business assets**
- ✧ To maintain **the value** of these assets to the business, they must be changed and updated
- ✧ The majority of the software budget in large companies is devoted to **changing and evolving** 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 modifications 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



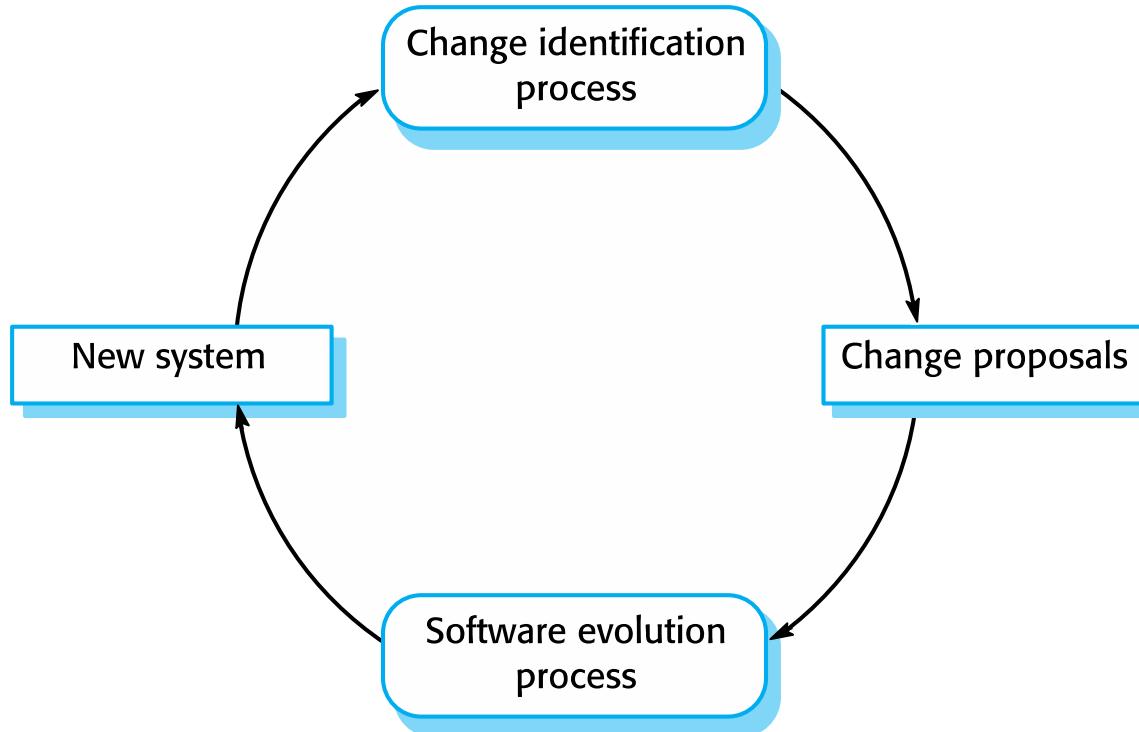
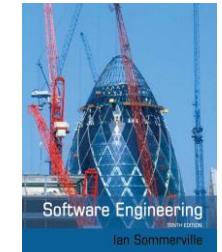
# Evolution processes



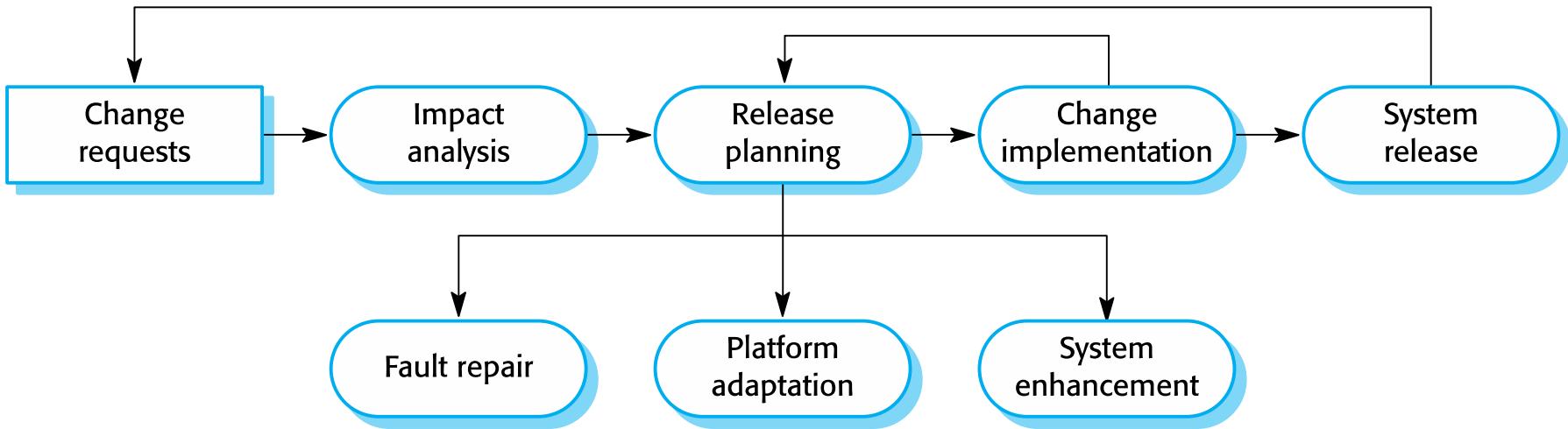
# Evolution processes

- ✧ 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

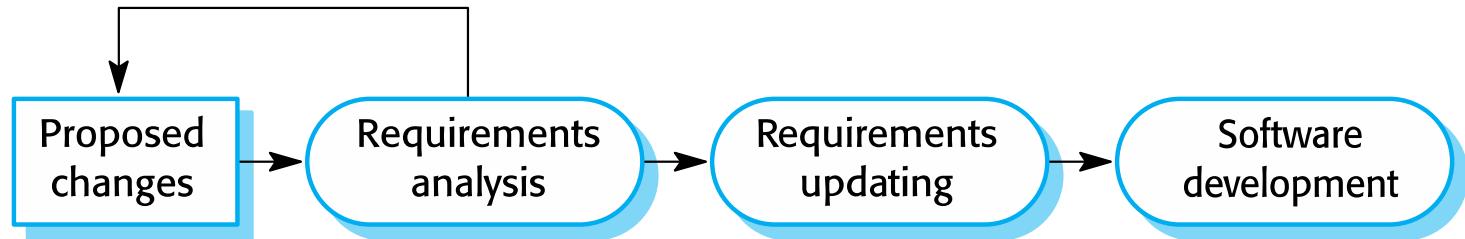
# Change identification and evolution processes



# The software evolution process



# Change implementation



# Change implementation



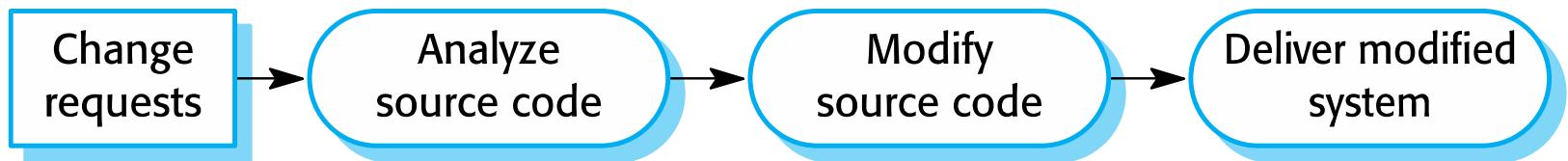
- ✧ Iteration of the development process where the revisions to the system **are designed, implemented and tested**
- ✧ A critical difference is that the first stage of change implementation may involve **program understanding**, especially if the original system developers are not responsible for the 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



# 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



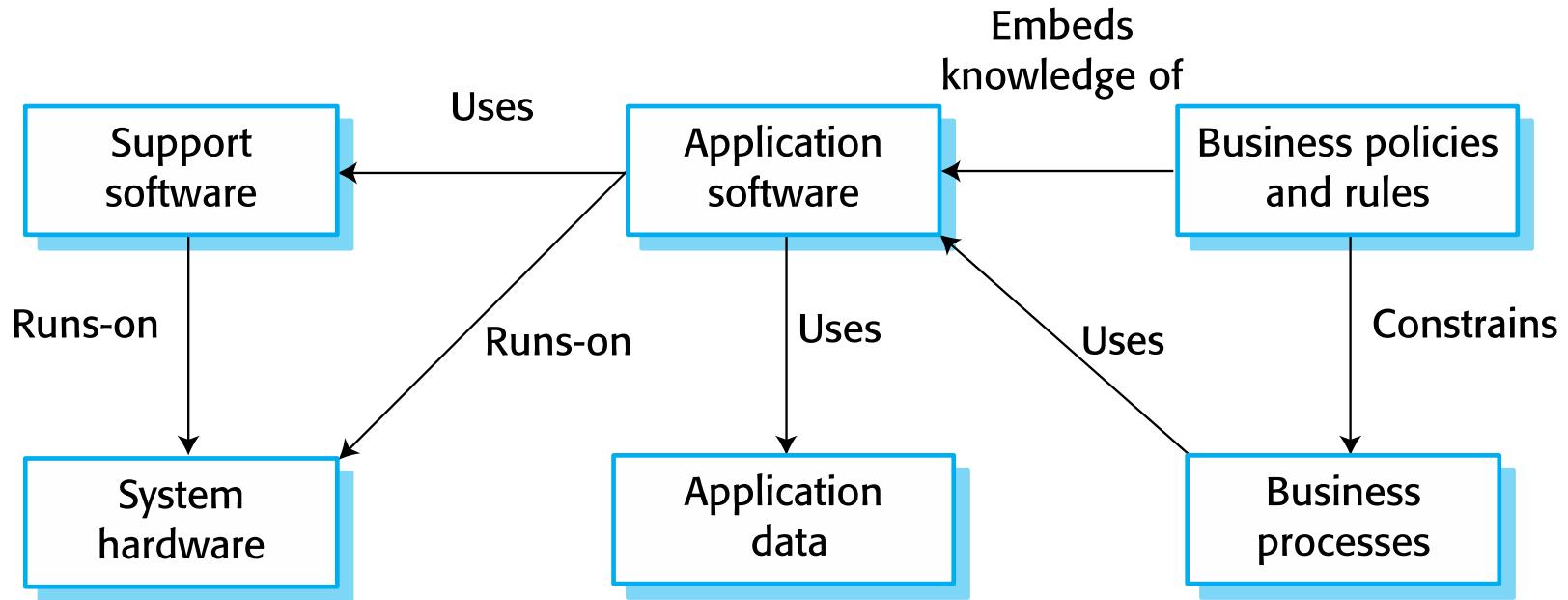
# Legacy systems

# 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

Business processes

Application software

Platform and infrastructure software

Hardware

# 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



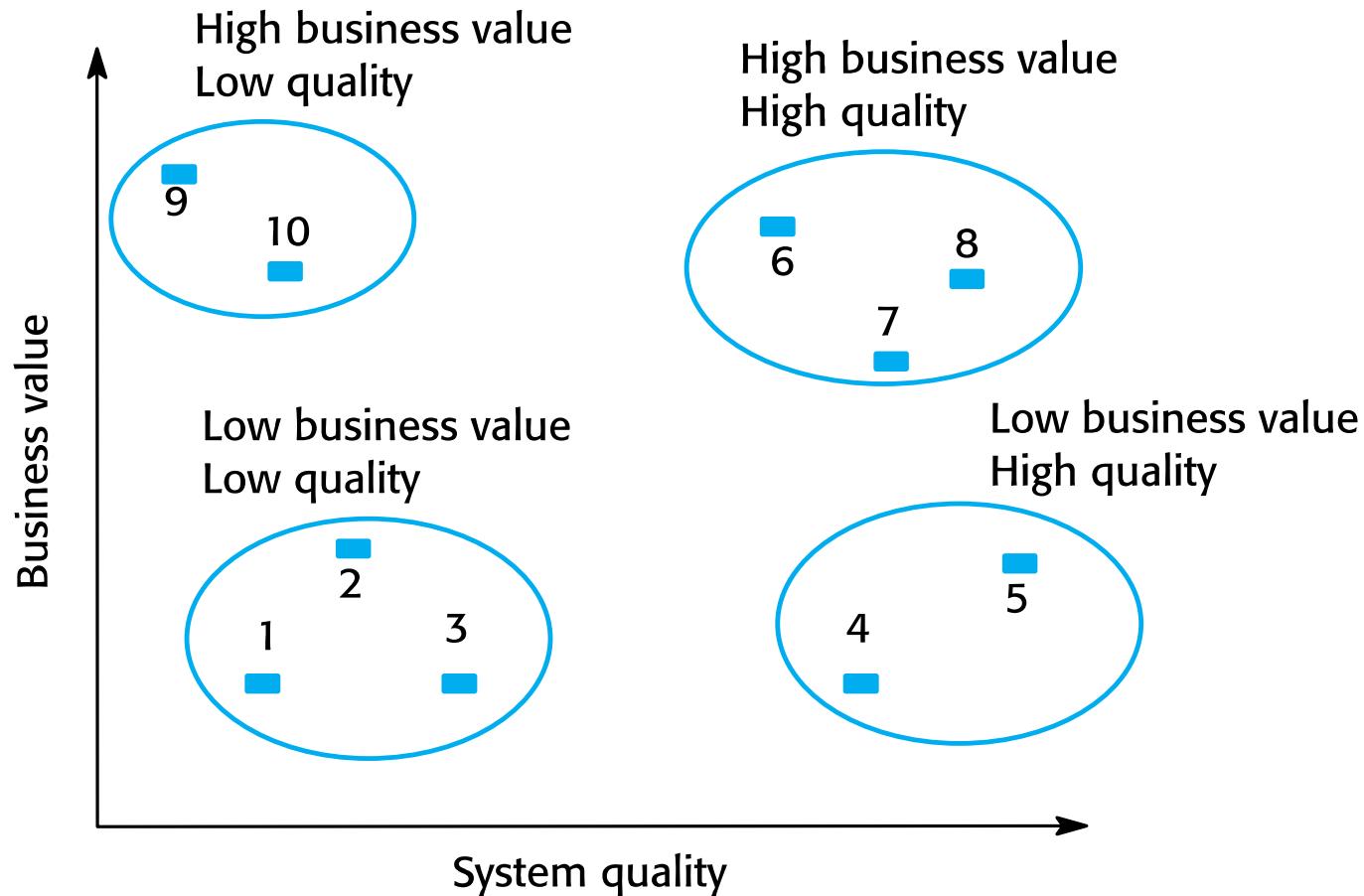
- ✧ Legacy systems are **expensive** to change for a number of reasons:
  - No consistent programming style
  - Use of obsolete programming languages with few people available with these language skills
  - Inadequate system documentation
  - System structure degradation
  - Program optimizations may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

# Legacy system management



- ✧ Organizations 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

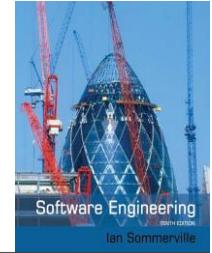
- These make an important business contribution but are expensive to maintain. Should be re-engineered or replaced if a suitable system is available.

## ✧ 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

- ✧ Business value assessment should take different viewpoints into account
  - System end-users
  - Business customers
  - 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 organization 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
<b>Support requirements</b>	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.
<b>Maintenance costs</b>	What are the costs of hardware maintenance and support software licenses? Older hardware may have higher maintenance costs than modern systems. Support software may have high annual licensing costs.
<b>Interoperability</b>	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



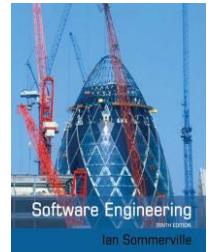
Factor	Questions
Understandability	How difficult is it to understand the source code of the current system? How complex are the control structures that are used? Do variables have meaningful names that reflect their function?
Documentation	What system documentation is available? Is the documentation 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
<b>Programming language</b>	Are modern compilers available for the programming language used to develop the system? Is the programming language still used for new system development?
<b>Configuration management</b>	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?
<b>Test data</b>	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?
<b>Personnel skills</b>	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



- ✧ 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

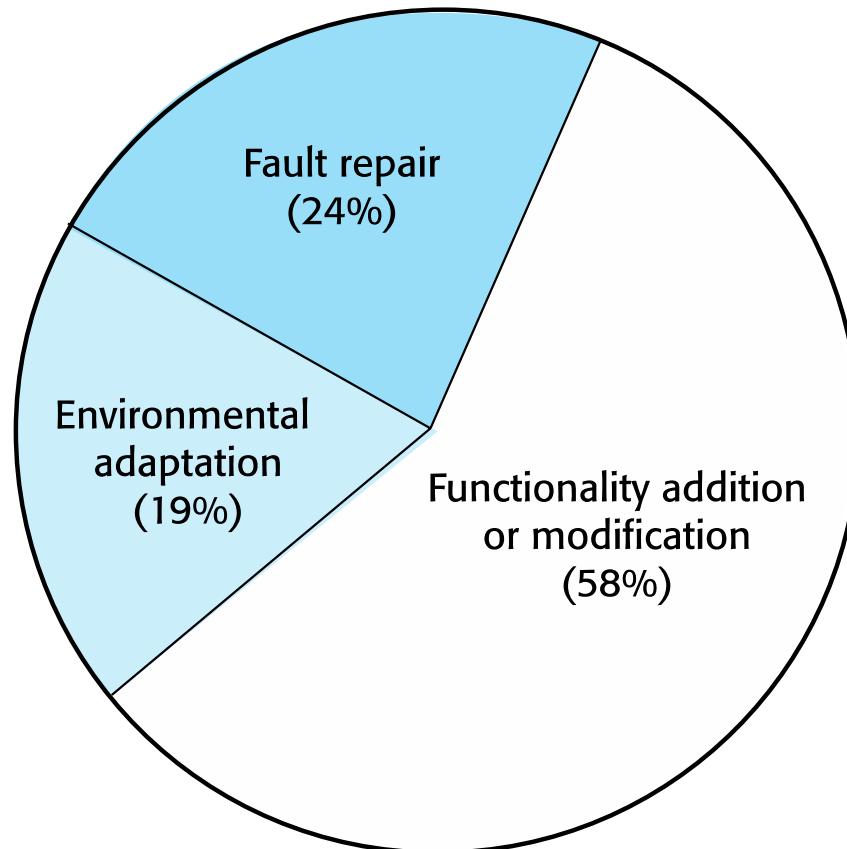
- Changing a system so that it operates in a different environment (computer, OS, etc.) from its initial implementation

## ✧ Functionality addition and modification

- Modifying the system to satisfy new requirements



# Maintenance effort distribution



# Maintenance costs



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

# Maintenance costs



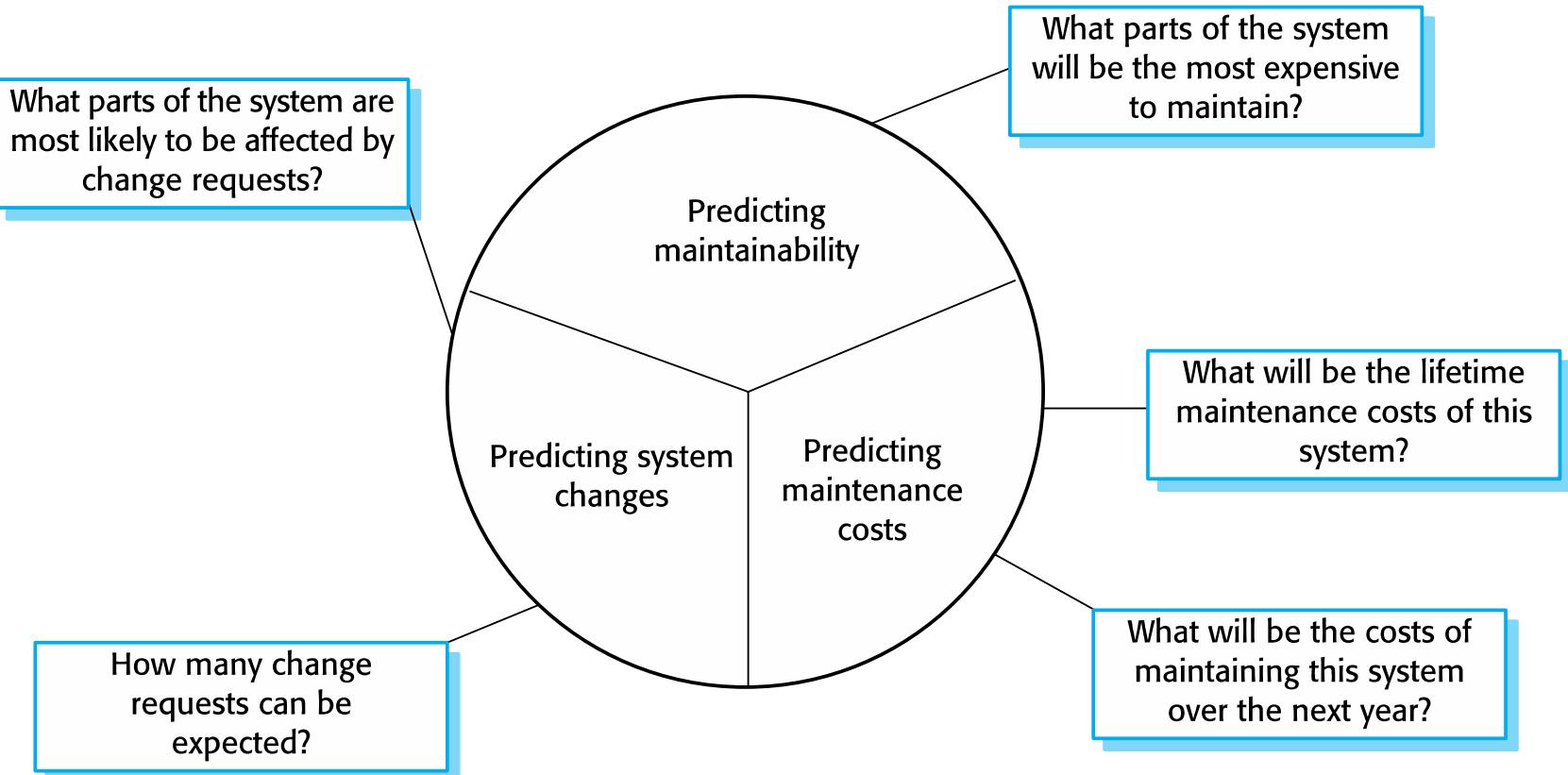
- ✧ It is usually **more expensive** to add new features to a system during maintenance than it is to add the same features during development
  - A new team has to understand the programs being maintained
  - Separating maintenance and development means there is no incentive for the development team 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





# 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



- ✧ **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 redocumented.



# Advantages of reengineering

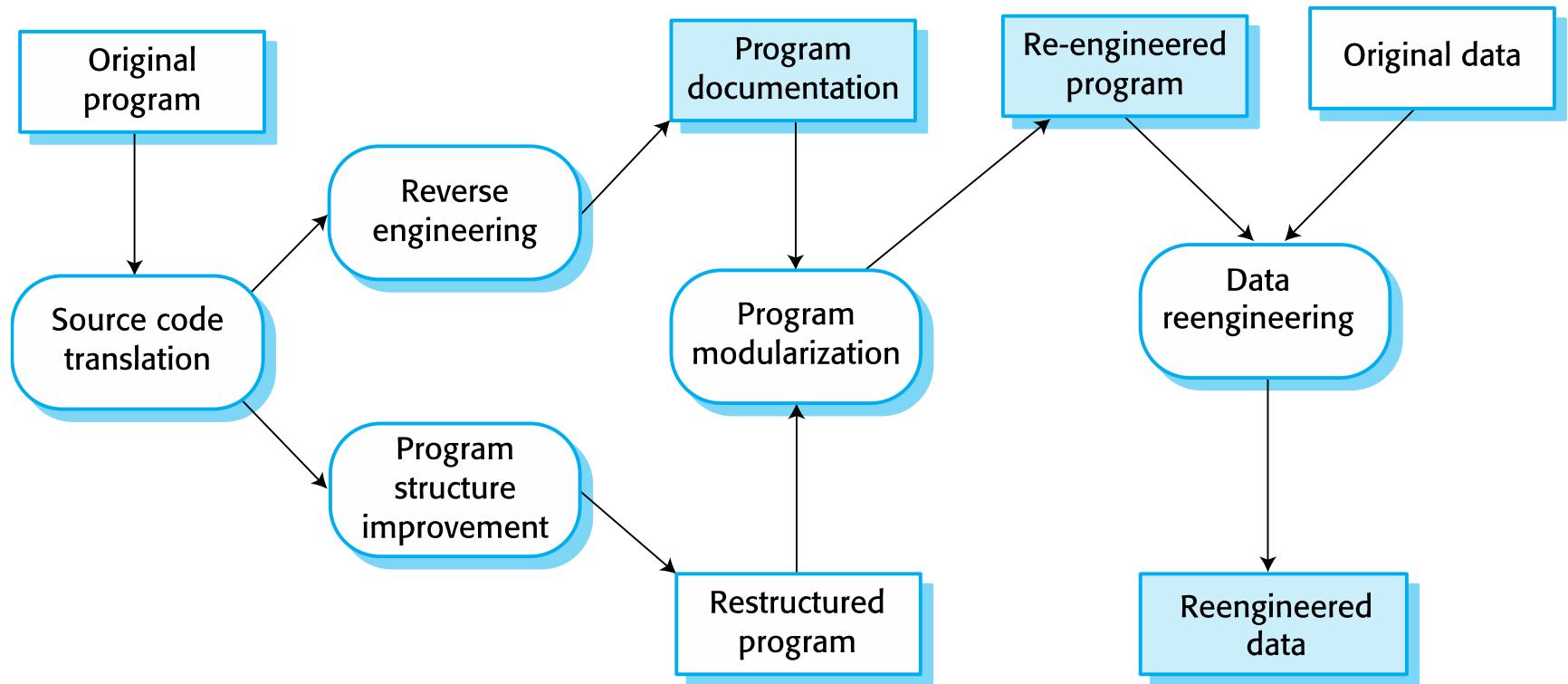
## ✧ Reduced risk

- 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

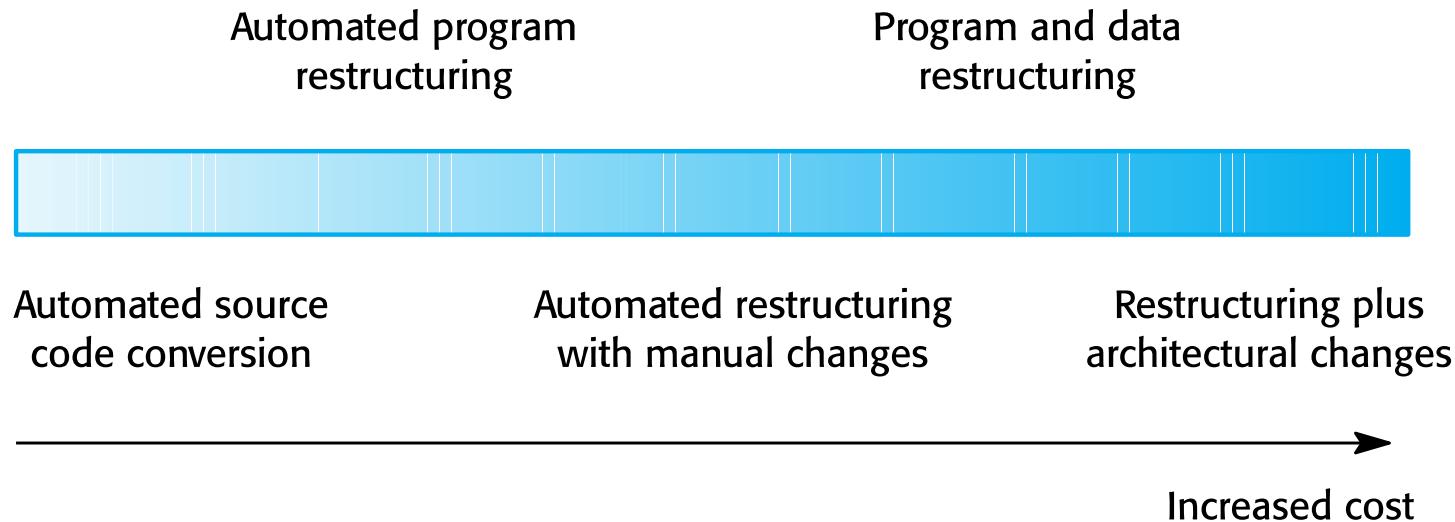


# Reengineering process activities



- ✧ Source code translation
  - Convert code to a new language
- ✧ Reverse engineering
  - Analyze the program to understand it
- ✧ Program structure improvement
  - Restructure automatically for understandability
- ✧ Program modularization
  - Reorganize 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 one refactors a program, one 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

Ian Sommerville,

*Software Engineering*, 10<sup>th</sup> Edition

Pearson Education, Addison-Wesley

Note: These are a slightly modified version of Chapter 10 slides available from the author's site <http://iansommerville.com/software-engineering-book/>



# Topics covered

- Dependability properties
- Sociotechnical systems

The following (in purple) are not required for the final exam:

- An example of STS: the cyber infrastructure of the NSF-funded Nevada Nexus project (past project with Dascalu co-PI)
- 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 the 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 interdependent.



# Importance of dependability

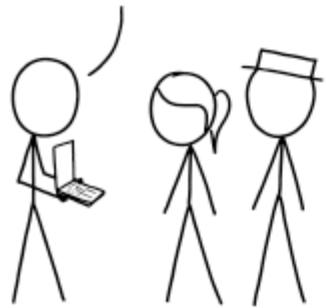
- System failures may have widespread effects with large numbers of people affected by the failure
- Systems that are not dependable and are unreliable, unsafe or insecure may be rejected by their users
- The costs of system failure may be very high if the failure leads to economic losses or physical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



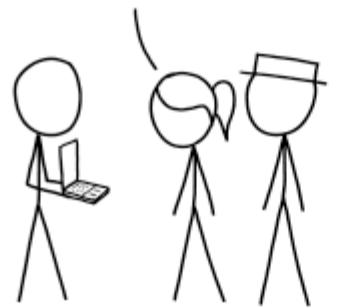
# Causes of failure

- **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 (human) failure**
  - Human operators make mistakes. This is perhaps the largest single cause of system failures in socio-technical systems.

CHECK IT OUT—I MADE A FULLY AUTOMATED DATA PIPELINE THAT COLLECTS AND PROCESSES ALL THE INFORMATION WE NEED.



IS IT A GIANT HOUSE OF CARDS BUILT FROM RANDOM SCRIPTS THAT WILL ALL COMPLETELY COLLAPSE THE MOMENT ANY INPUT DOES ANYTHING WEIRD?



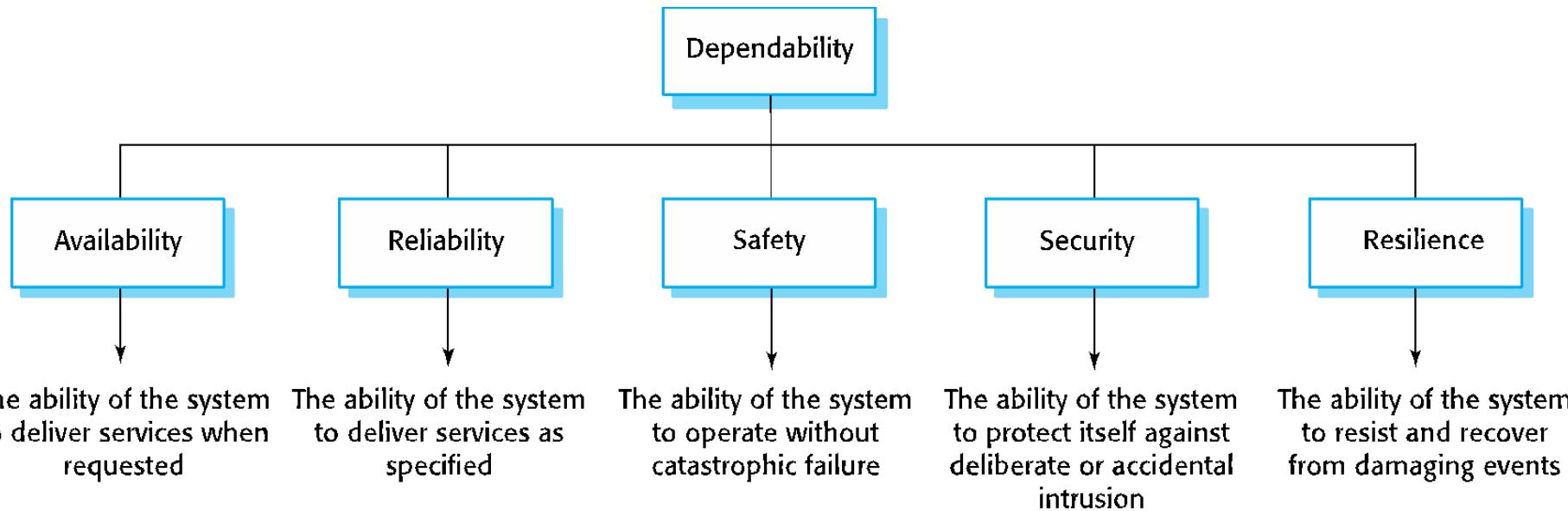
IT... MIGHT NOT BE.  
I GUESS THAT'S SOMETHING.  
WHOOPS, JUST COLLAPSED. HANG ON, I CAN PATCH IT.





# Dependability properties

# The principal dependability properties





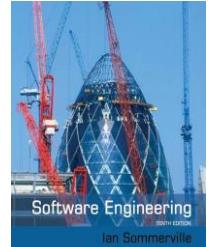
# Principal properties

- **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 recognize and resist cyberattacks
- Include recovery mechanisms to help restore normal system service after a failure



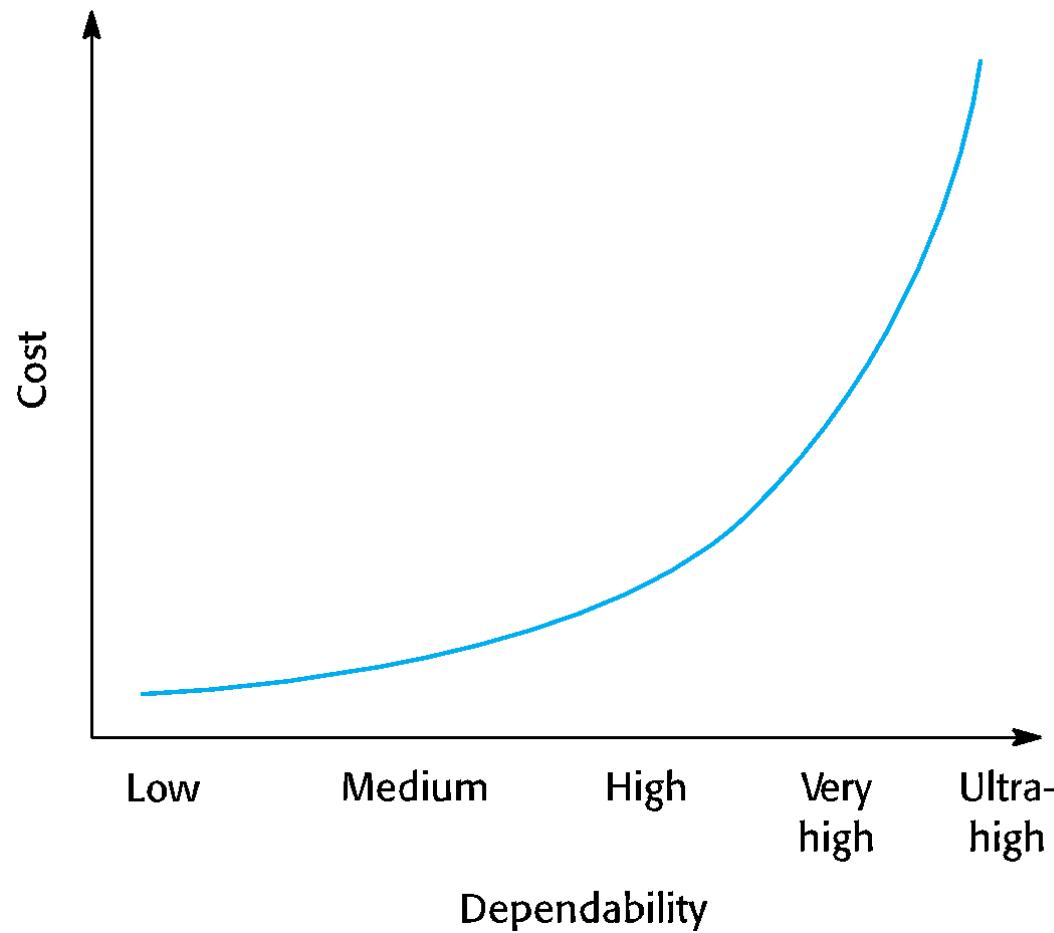
# Dependability costs

- 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



Software Engineering  
Ian Sommerville

# Cost/dependability curve





# 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 cannot be trusted may lose future business
- **Depends on system type** - for business systems in particular, modest levels of dependability may be adequate



# Sociotechnical systems (STS)

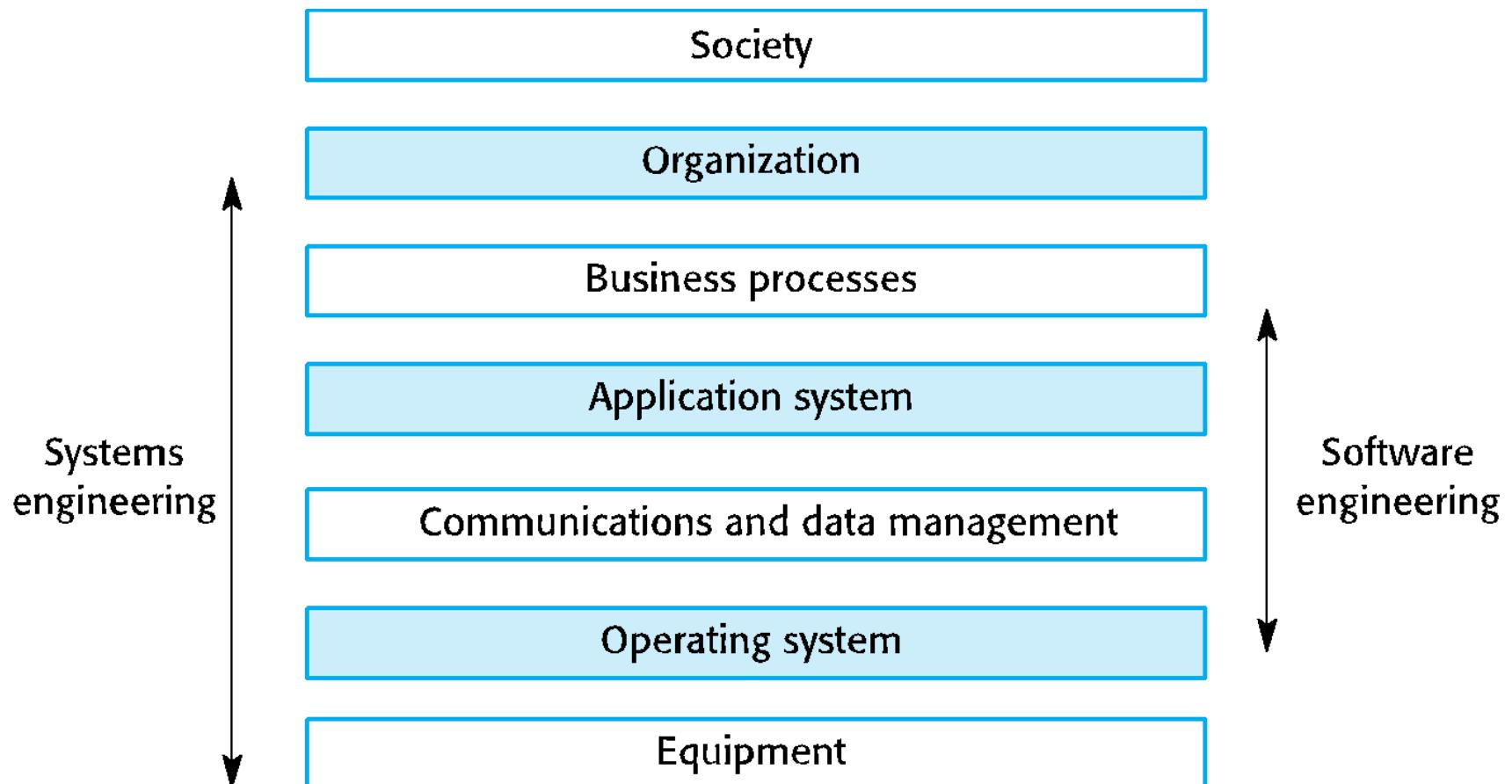
# Systems and software



- 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.

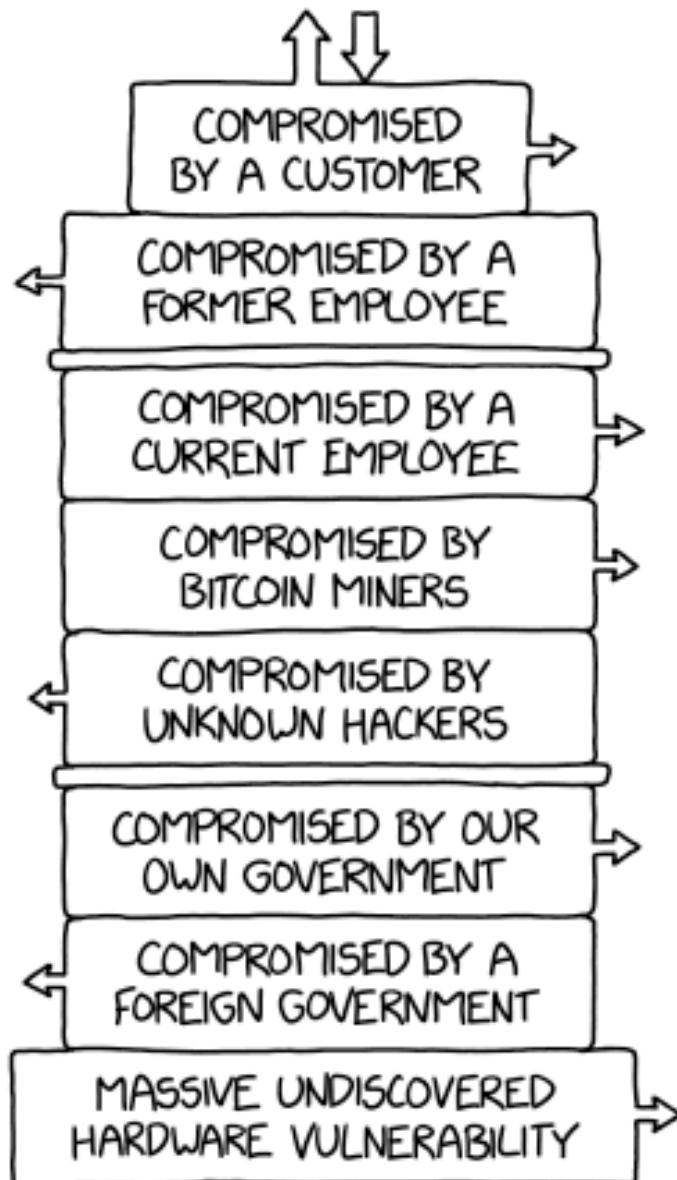


# The sociotechnical systems stack





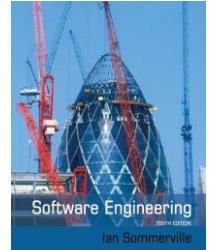
## THE MODERN TECH STACK





# 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

- 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



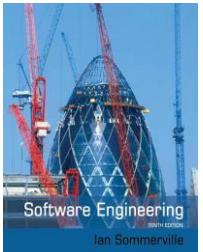
# Regulation and compliance

- 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 privately owned companies so that they **must follow certain standards** to ensure that their products are safe and secure

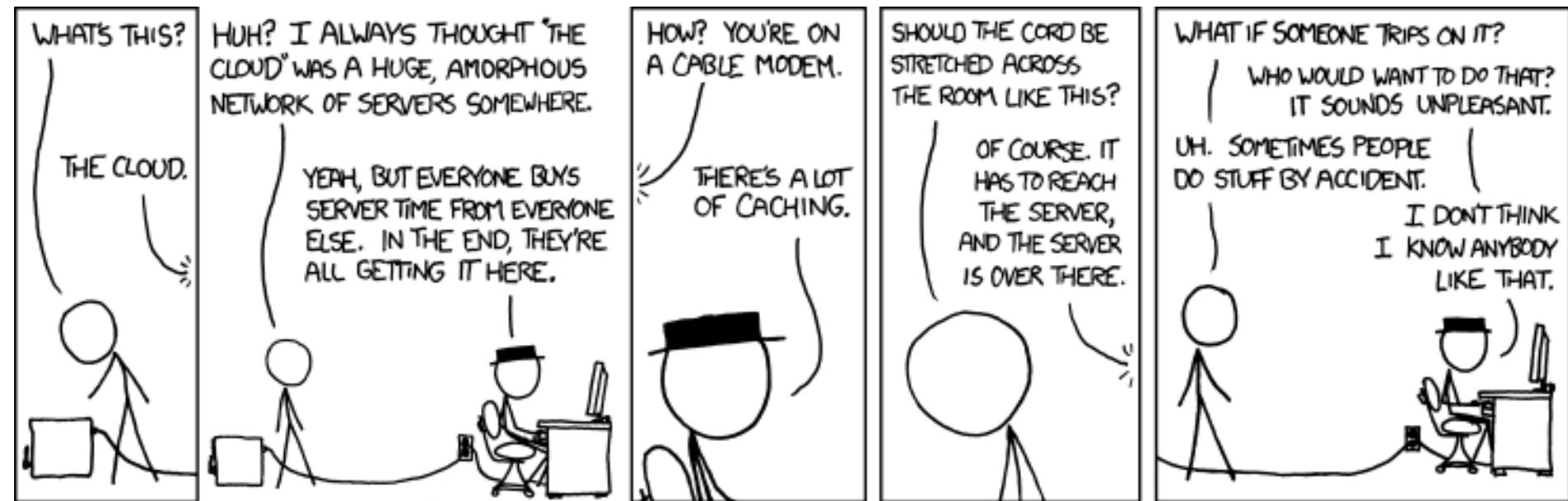
# Regulated systems



- 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.



Software Engineering  
Ian Sommerville





# 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 to develop the documentation for certification as it is to develop the system itself



An example of STS: the cyber infrastructure (CI) of the NSF-funded Nevada Nexus project

# The Nexus project: *overview*



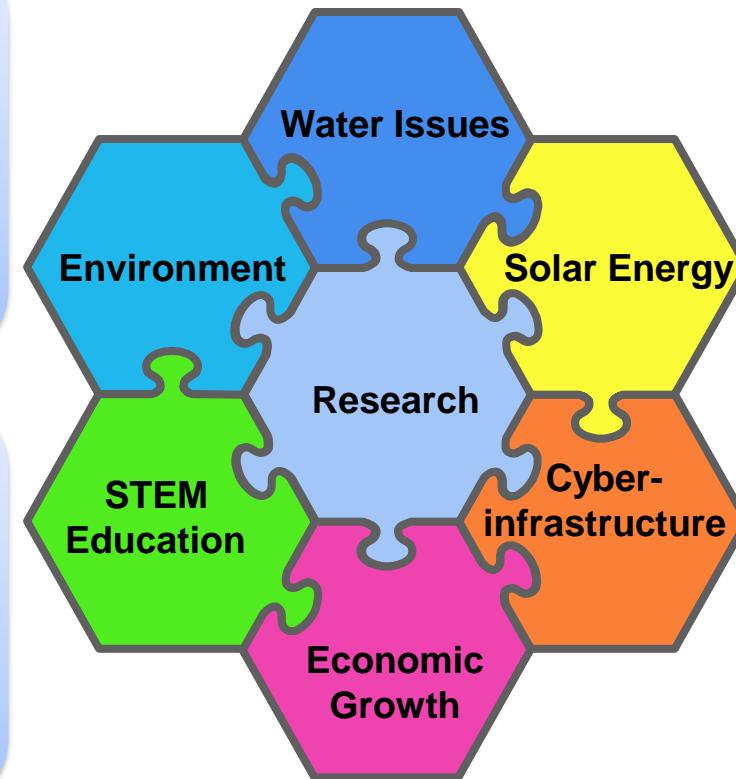
- Project funded by the National Science Foundation (NSF) as an EPSCoR RII Track 1 project:
  - 2013-2023
  - About \$28.0 million in funding
- Project Director: Dr. Gayle Dana (DRI), until 2019, then Dr. Fred Harris (UNR)
- Participants: over 40 faculty members (of which 5 co-Principal Investigators – PIs), 30 students, and 20 technicians from several main research institutions in the state, including UNR, UNLV, and DRI

# The Nexus project: *mission*

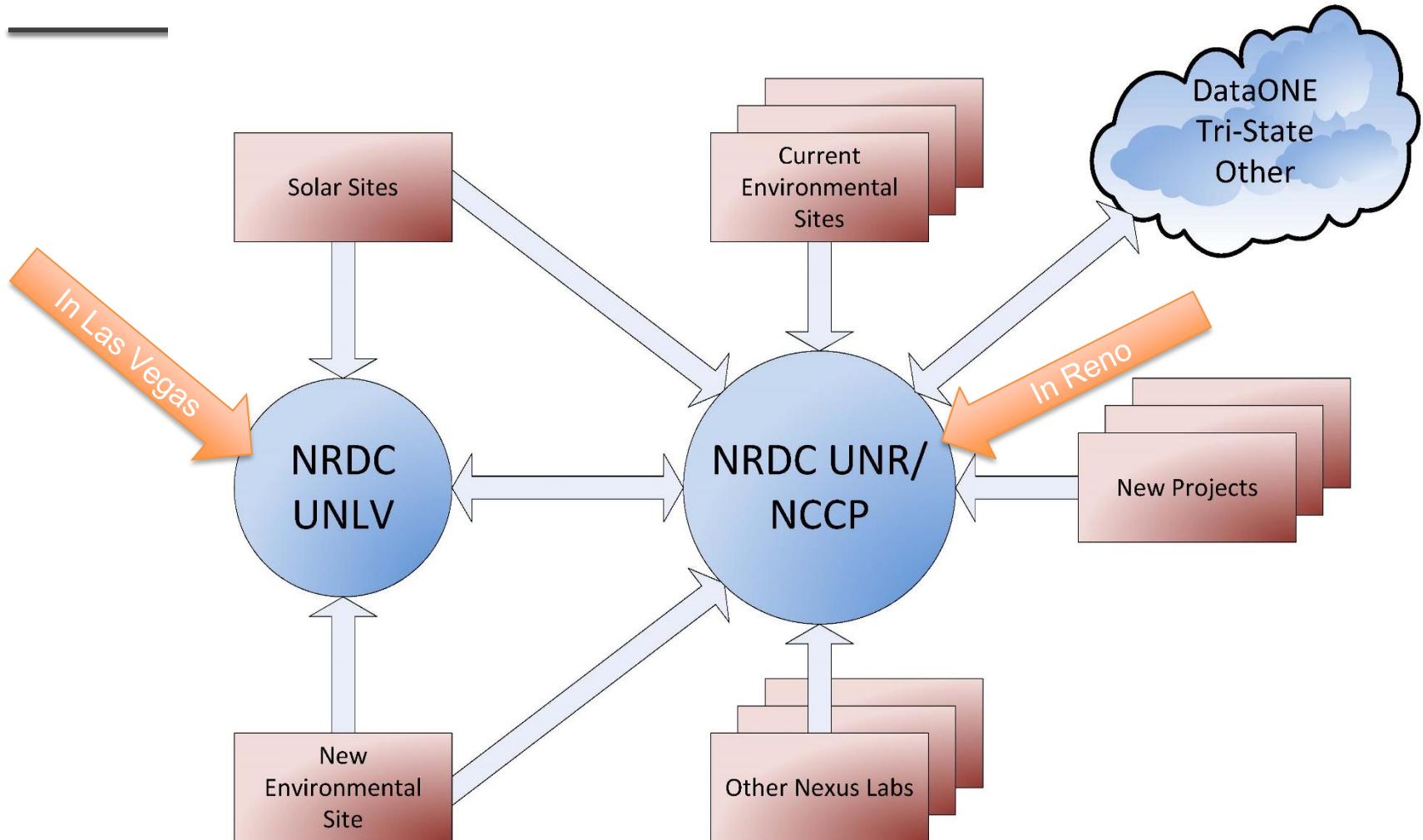
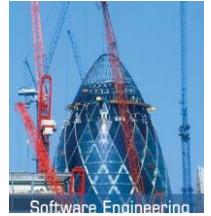


Our mission is to **advance knowledge and discovery** through research on solar energy generation technology, its environmental impacts and associated water issues, and accelerate this research by **developing new capabilities in cyberinfrastructure**.

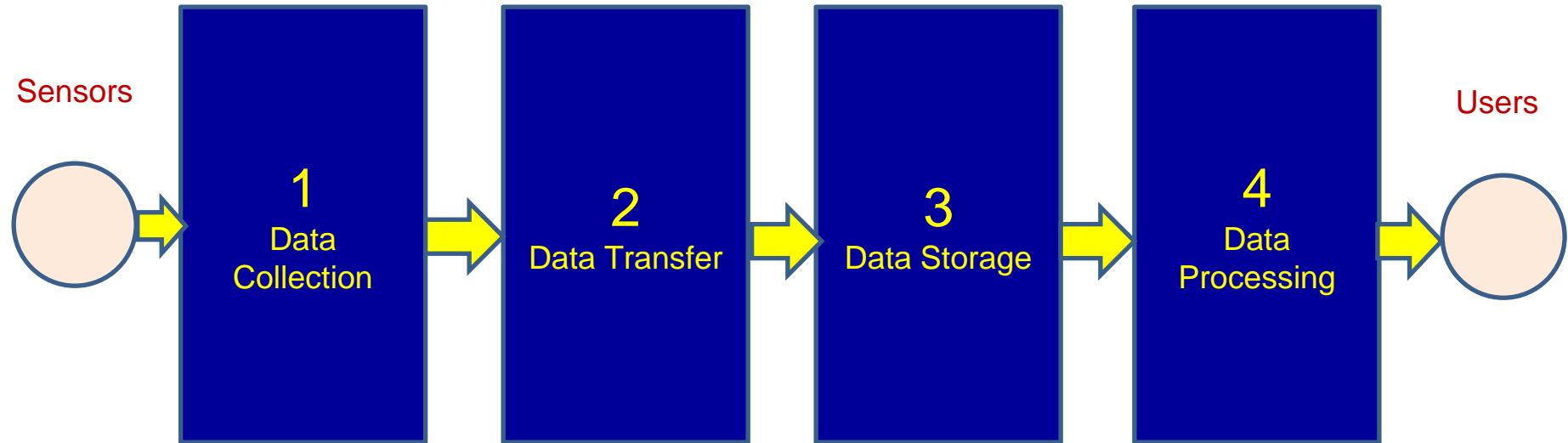
Benefits include diversifying Nevada's economy, building its workforce, and developing innovative approaches to STEM education.



# The Nexus project: *Cl synopsis*



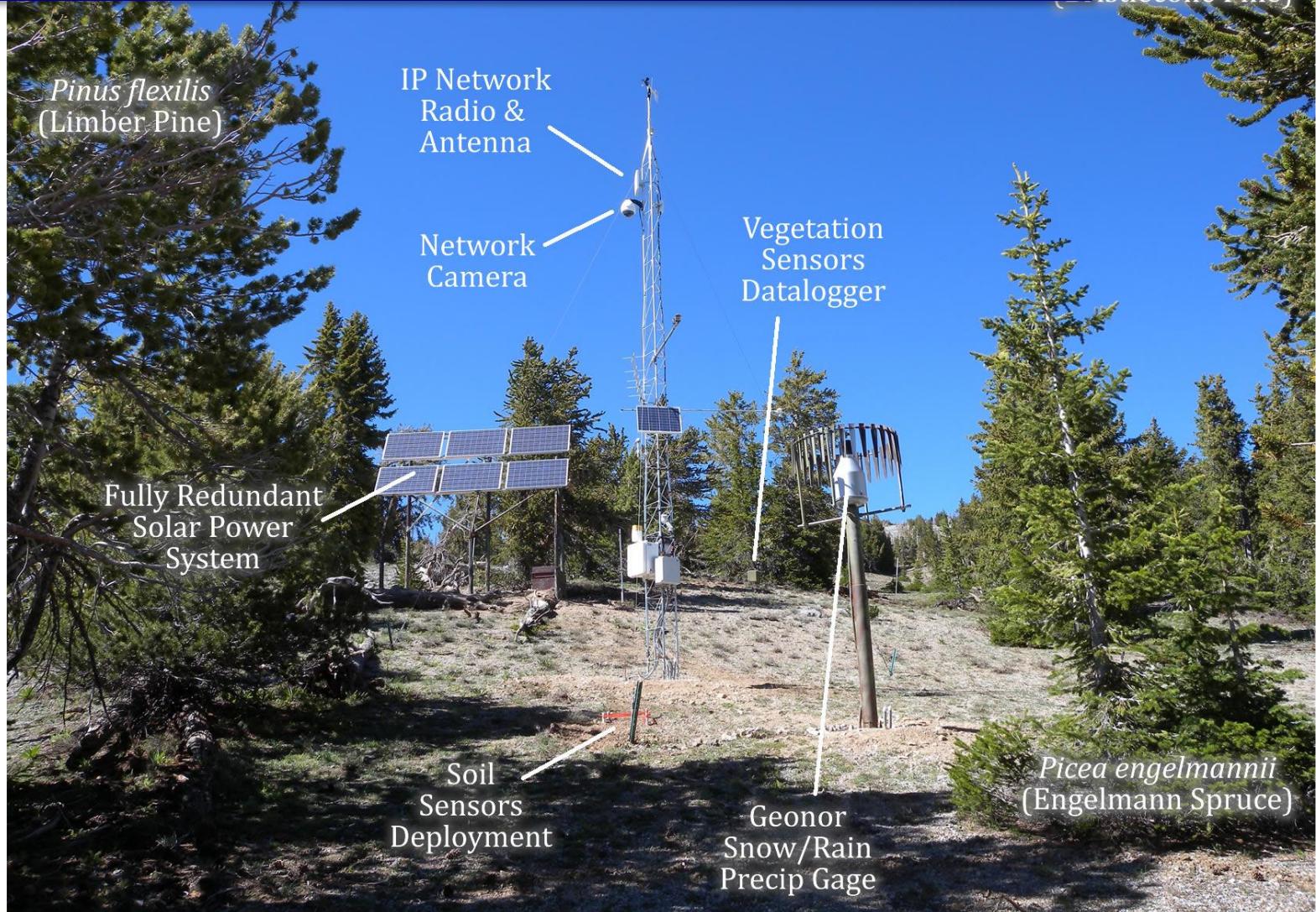
# The end-to-end CI system



# The CI system: 1 - data collection



*Pinus longaeva*  
(Bristlecone Pine)



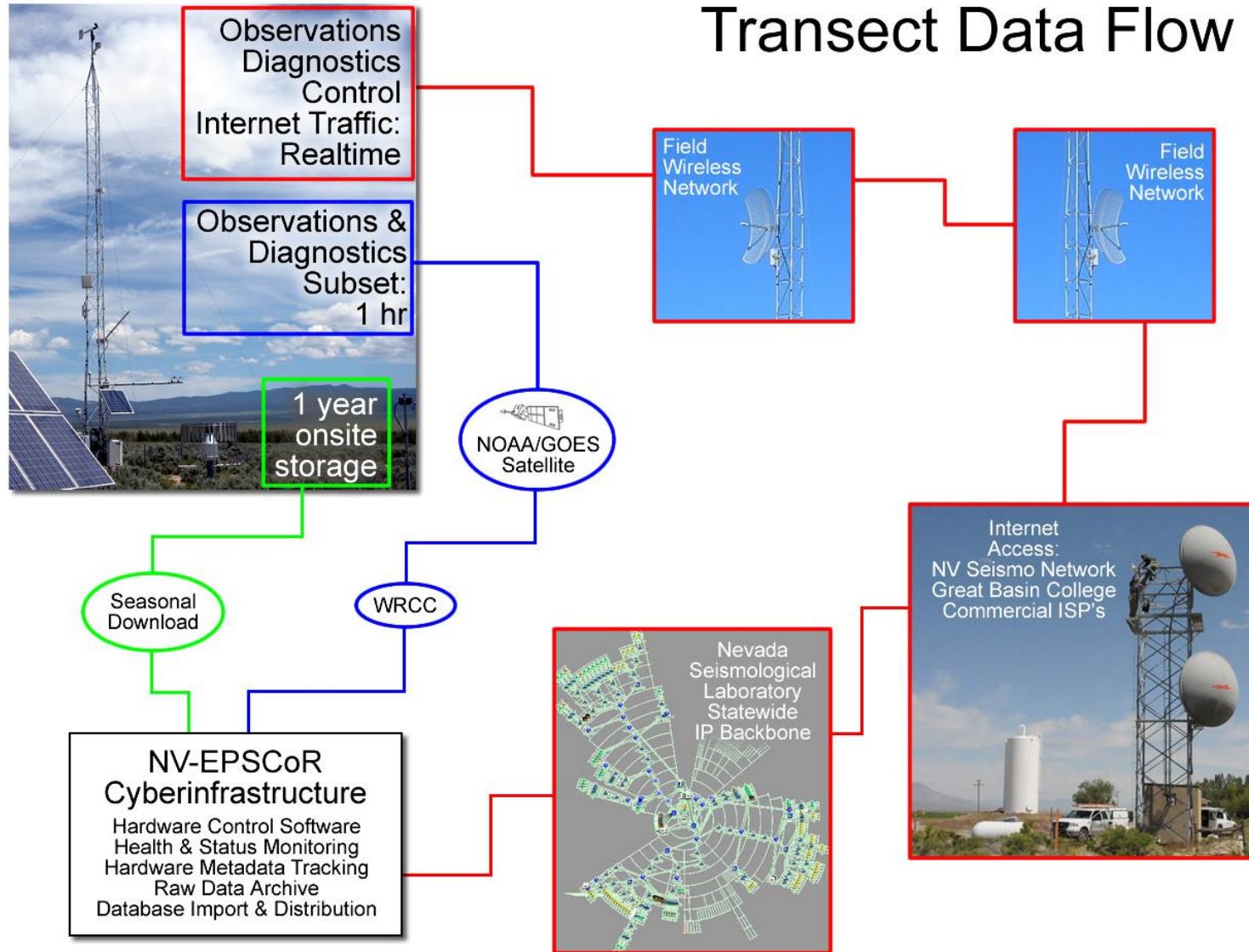
# The CI system: 1- data collection



# The CI system: 1- data collection



# The CI system: 2 - data transfer



# The CI system: 2 - data transfer

## Environmental connectivity



# The CI system: 3 - data management



## Data Management Cycle

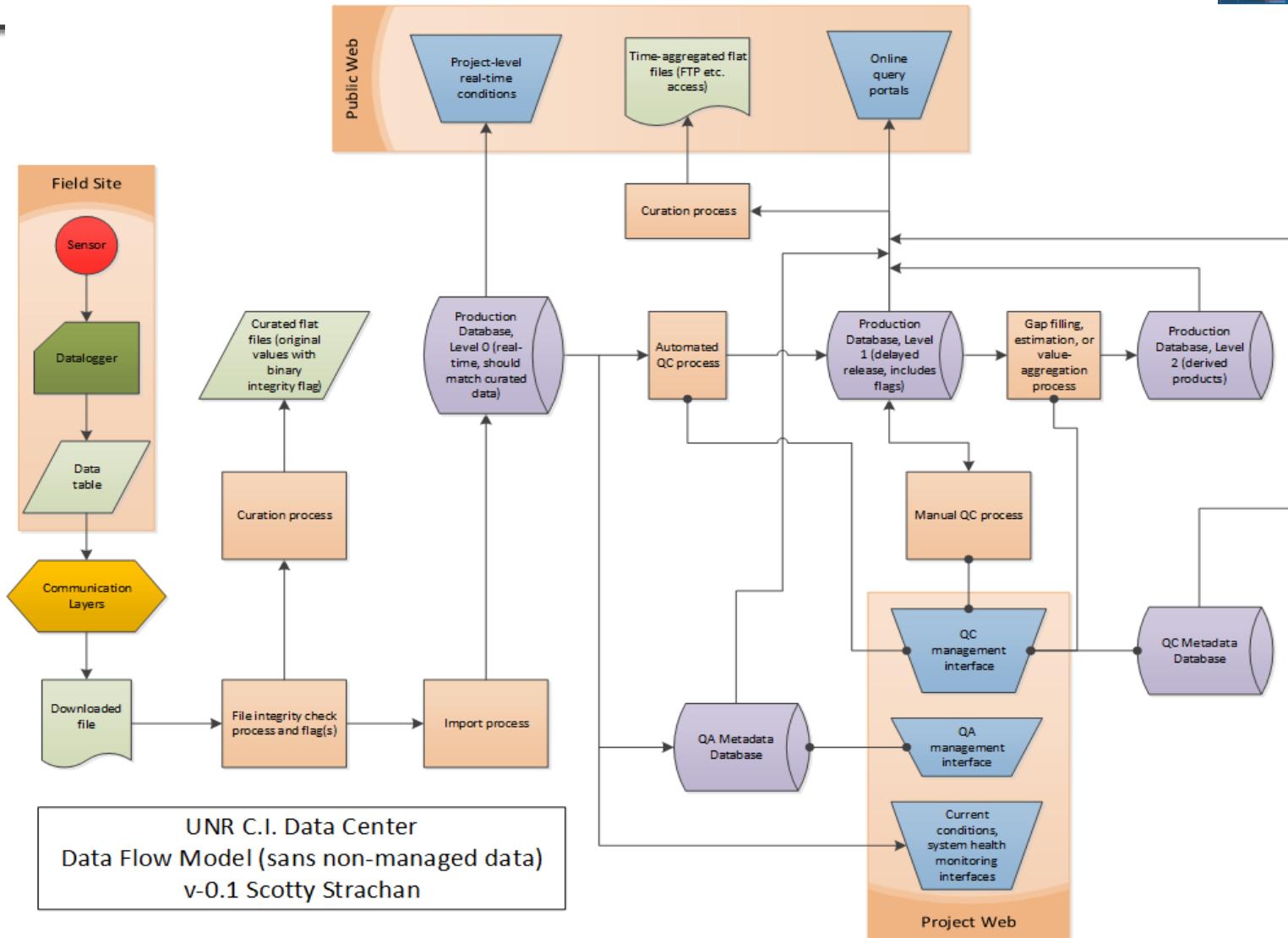


This management ***problem is universal*** for observational science, particularly the earth sciences.

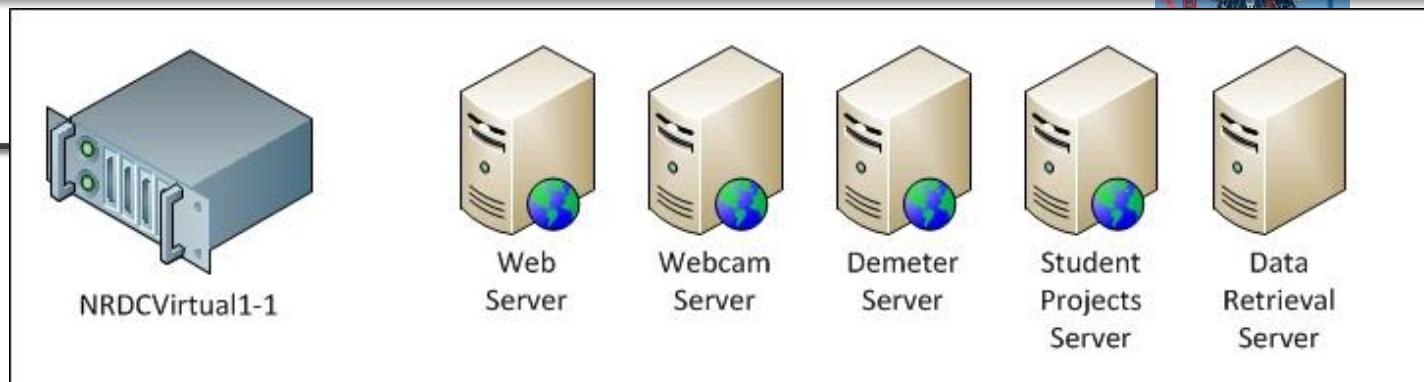
Many organizations are working on this problem, but ***no “one” solution exists*** yet.

Graphic: Wade Sheldon, ESIP Enviroensing Cluster

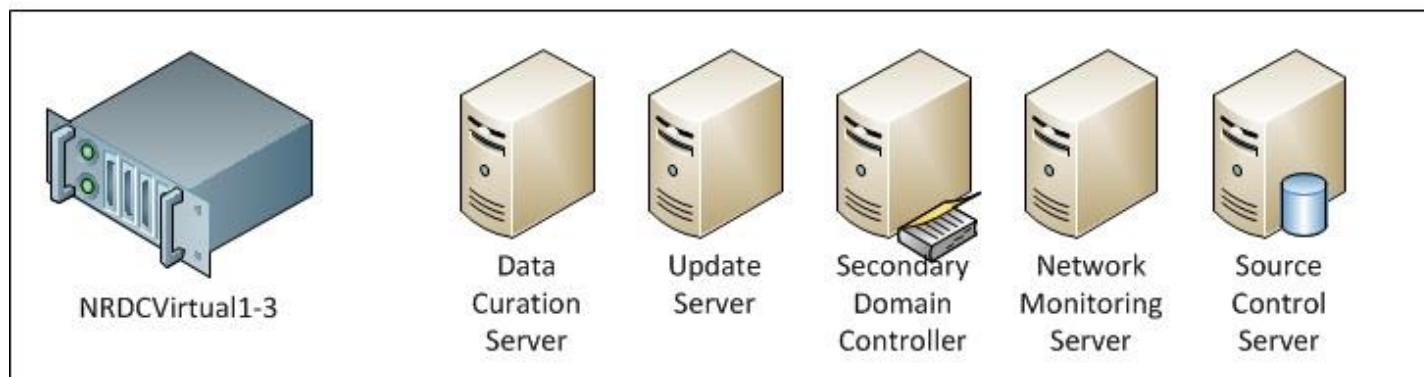
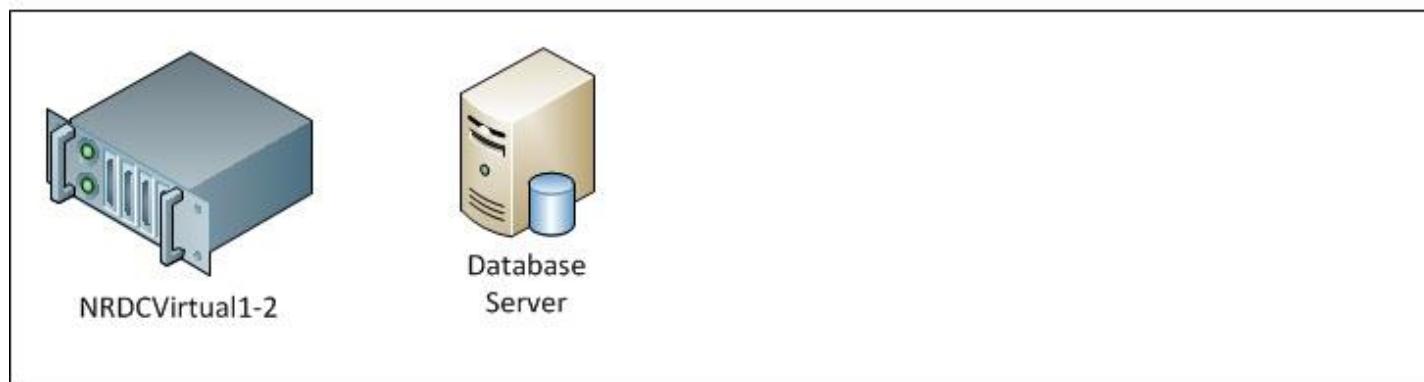
# The CI system: 3 - data flow & management



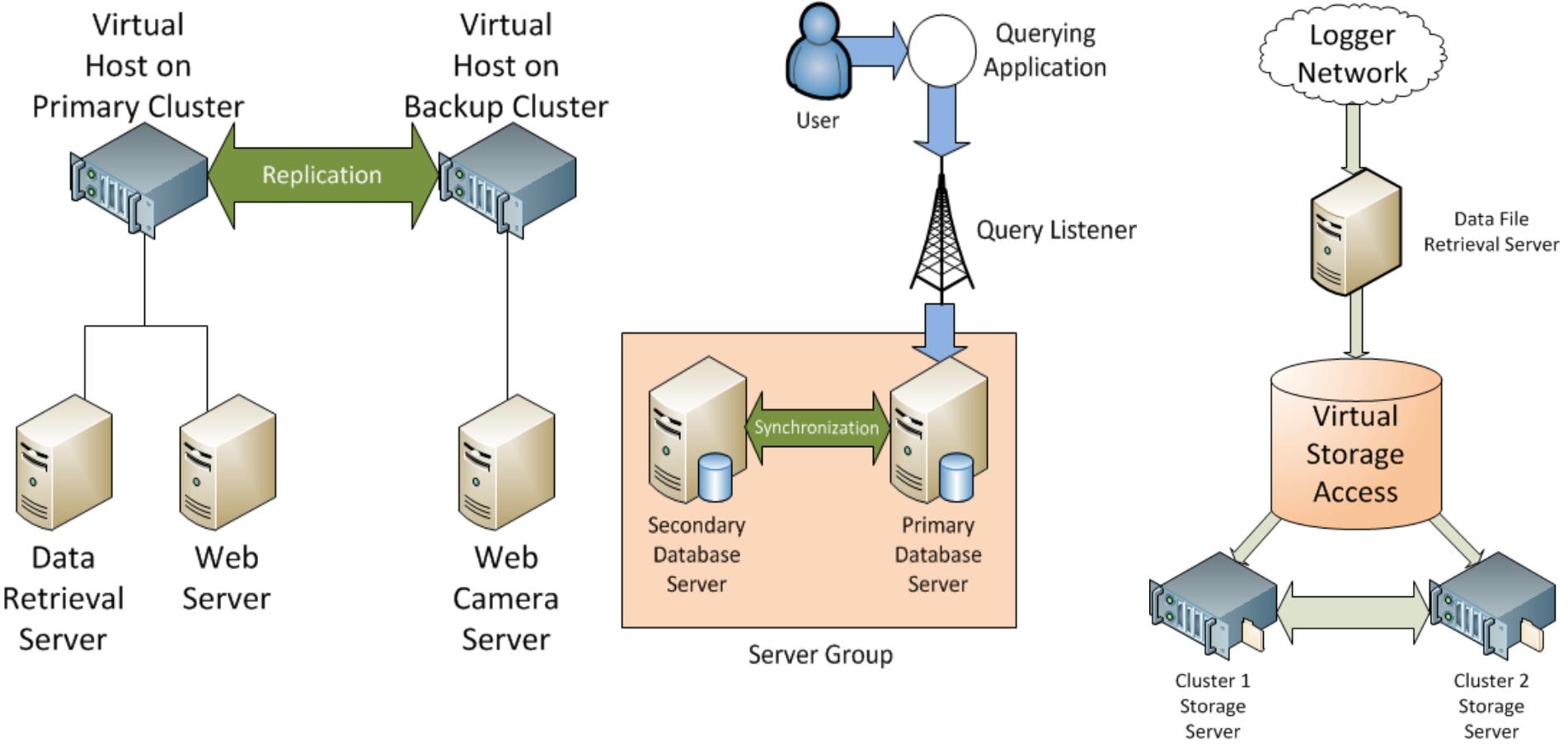
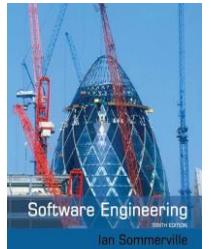
# The CI system: 3 - data center architecture



Physical and  
logical server  
architecture



# The CI system: 3 - data storage



# The CI system: 4 - data processing

## Nevada Research Data Center Acquire and Share Research Data and Results

### Featured Projects

1 2 3 4

#### NevCAN

NevCAN is a network of sites dedicated to the monitoring and recording of climate conditions in some of the most remote places in Nevada. The transects setup at each of these sites have a variety of sensors such as air temperature and humidity, imagery, precipitation, soil moisture, and tree sap flow. In addition, several efforts have been pursued in improving climate change education as well as climate modeling.



[Featured Projects](#)

[Suggest a Dataset](#)

[Submit Research Results](#)

[About Us](#)

Hosted by the University of Nevada, Reno

[Site Map](#)

Supported by NSF grant # IIA-1301726

# The CI system: 4 - data processing

NRDC

Data Projects Resources Connections Contact



Current Conditions

Snake Range West Subalpine

# NRDC website

Last recorded: 2/2/2015, 9:25:00 AM

## Last Recorded

Temperature 25.4 °F  
-3.7 °C

Wind Speed 5.3 mph  
2.4 m/s

Wind Direction 276.5 °North

Relative Humidity 81.7 %

## Last 6 Hours

Barometric Pressure Change +0.03 in Hg  
+1.13 mbar

Average Wind Direction 263.8 °North

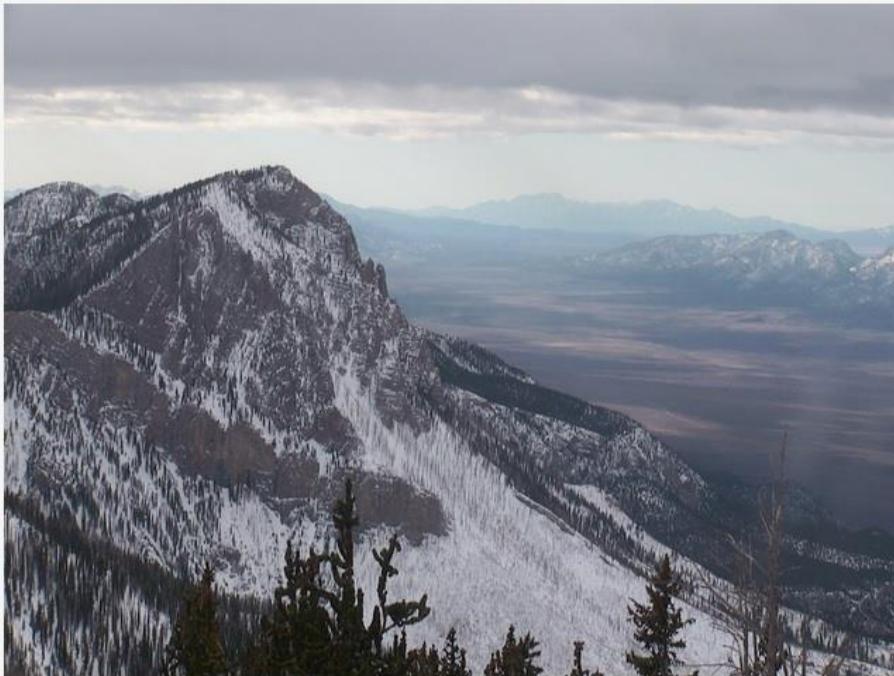
## Last 24 Hours

High Temperature 41.3 °F  
5.2 °C

Low Temperature 21.7 °F  
-5.7 °C

Maximum Wind Gust 27.5 mph  
12.3 m/s

Accumulated Precipitation 0.2 inch  
0.0 mm



Latest Webcam Image: South zoom  
2/2/2015, 9:03:45 AM

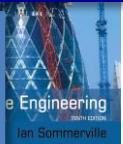
Hosted

Map

Supported by NSF grant # IIA-1301726

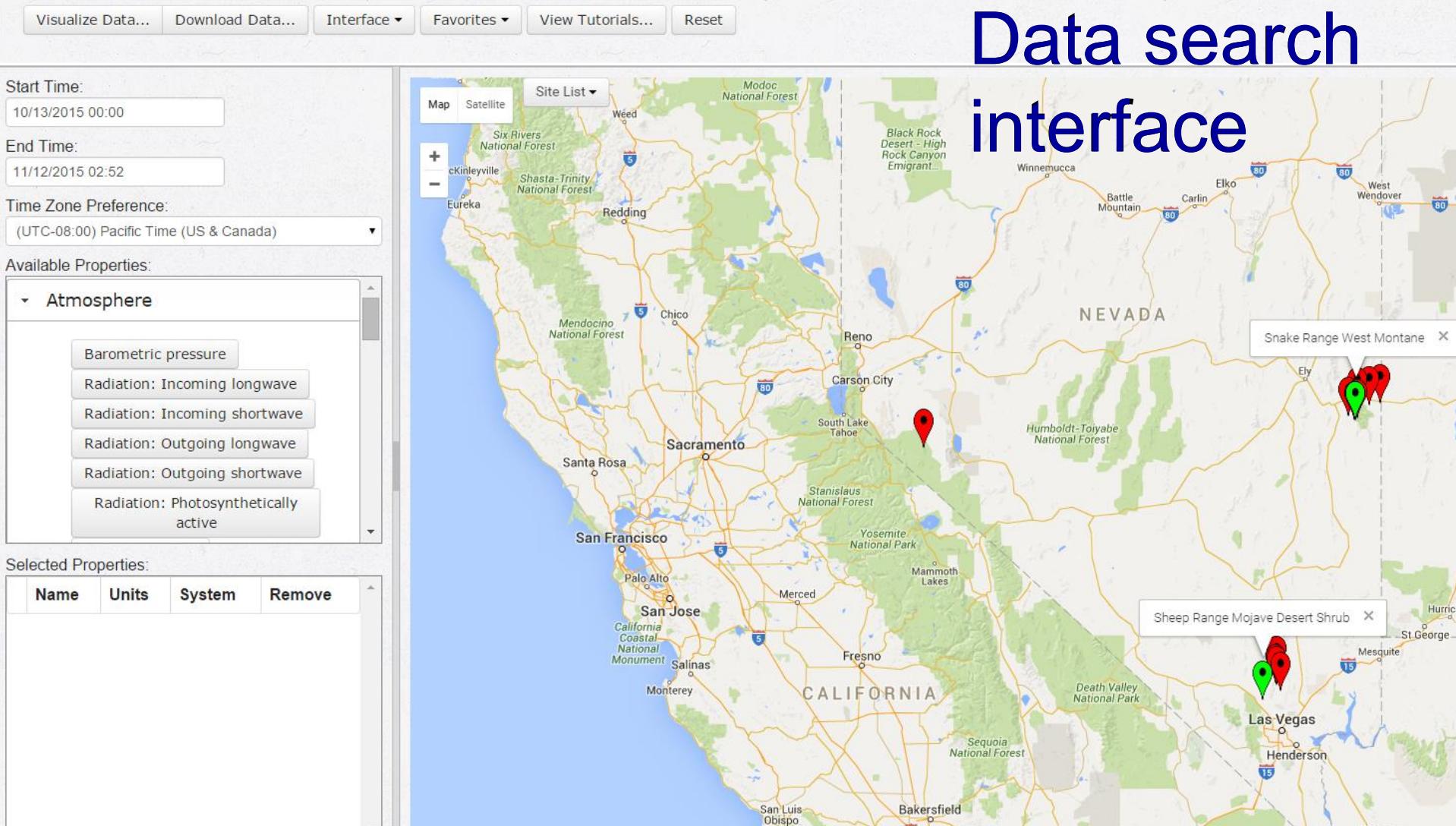
# The CI system: 4 - data processing

*Observation of ongoing phenomena*



12/04/2014 13:03:04 PST

# The CI system: 4 - data processing



# The CI system: 4 - data processing



[Scotty Strachan, Geography, UNR – *Sub-alpine mountain sublimation*]



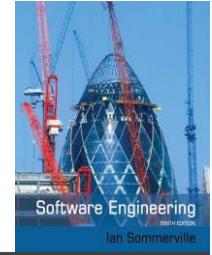
# Redundancy and diversity



# 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, it 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

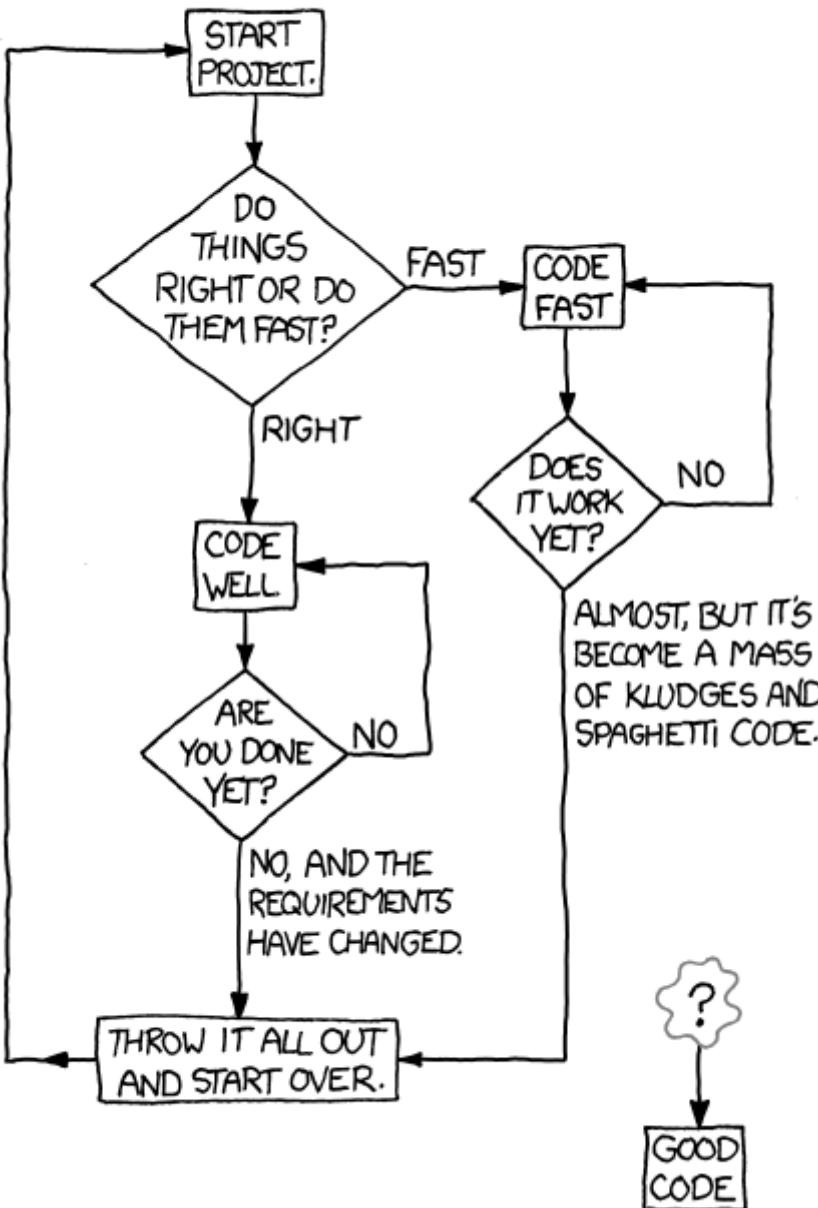
- Dependable software often requires certification so both **process and product documentation** has to be produced
- **Up-front requirements analysis** is also essential to discover requirements and requirements conflicts that may compromise the safety and security of the system
- These **conflict with 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**

## HOW TO WRITE GOOD CODE:





# 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

- **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 analyzing 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 a 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 specifications are **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 22 – Project Management

Ian Sommerville,

*Software Engineering*, 10<sup>th</sup> Edition

Pearson Education, Addison-Wesley

Note: These are a slightly modified version of Chapter 22 slides available from the author's site <http://iansommerville.com/software-engineering-book/>



# Topics covered

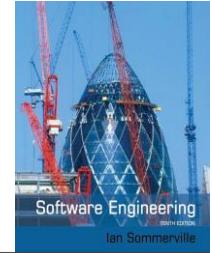
---

- ◊ What is software project management?
- ◊ 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 organizations 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 organization 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*

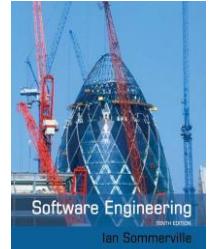
- Project managers are responsible for planning, estimating and scheduling project development and assigning people to tasks
- Covered in Chapter 23

## ◊ *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

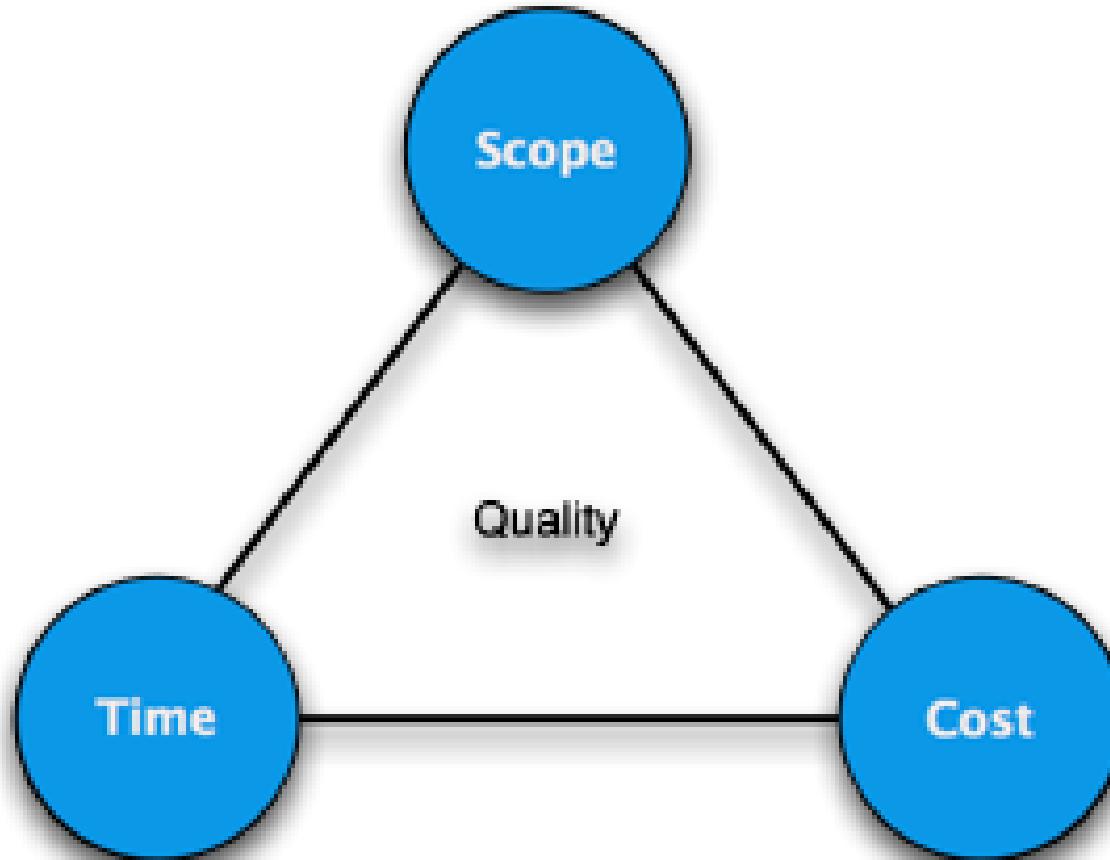
## ◊ *Proposal writing*

- 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

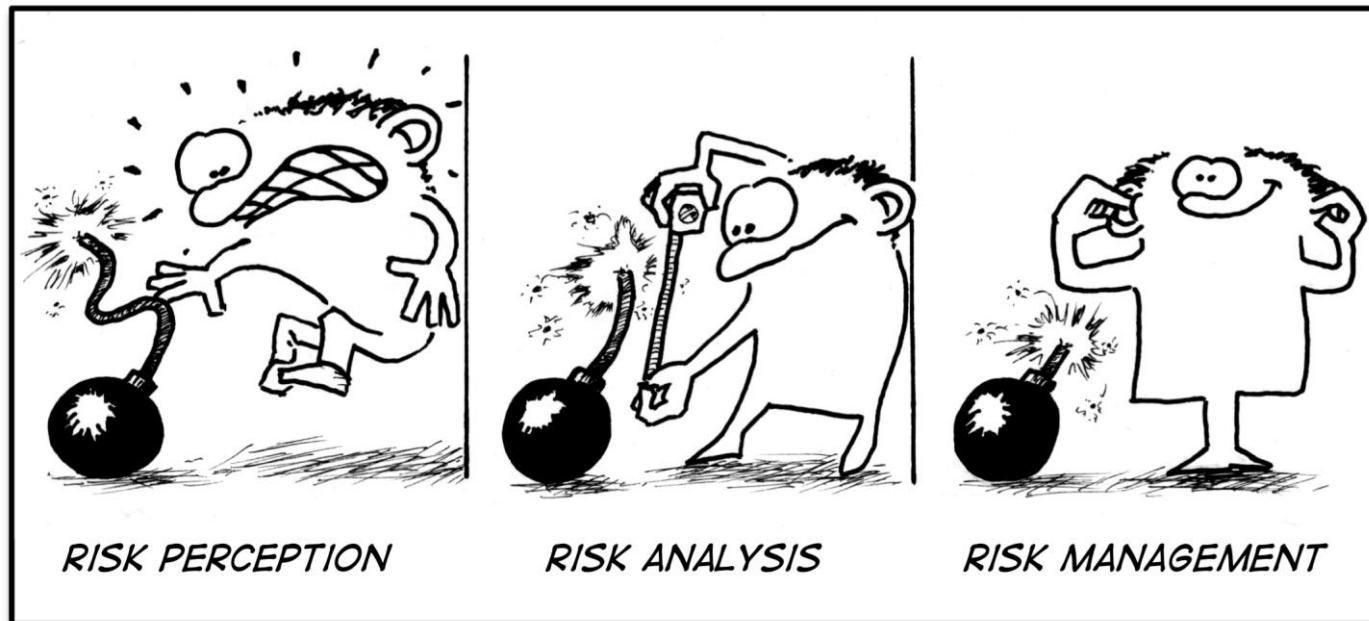


Software Engineering  
Ian Sommerville

# Project Triangle



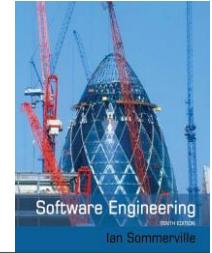
# Risk management





# 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, etc.)
  - 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 organization developing or procuring the software

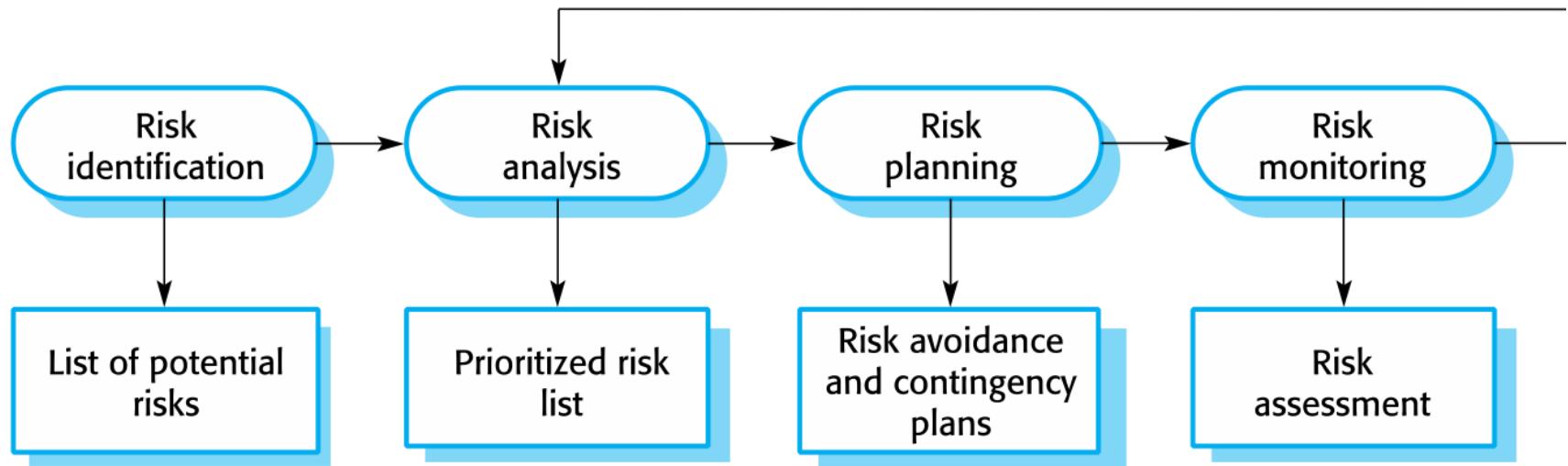


# Examples of project, product, and business risks

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



# The risk management process





# Risk identification

- ◊ 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

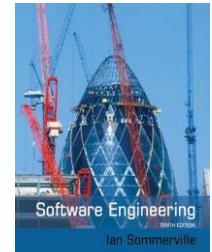
Risk	Probability	Effects
The time required to develop the software is underestimated (12).	High	Serious
Software tools cannot be integrated (9).	High	Tolerable
Customers fail to understand the impact of requirements changes (11).	Moderate	Tolerable
Required training for staff is not available (5).	Moderate	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 Register												
Risk Id	Risks	Current Risk			Status	Owner	Raised	Mitigation Strategies	Residual Risk			
		Likelihood	Impact	Severity					Likelihood	Impact	Severity	
Category 1: Project selection and Project finance												
RP-01	Financial attraction of project to investors	4	4	16	Open		01-march	<ul style="list-style-type: none"> <li>▪ Data collection</li> <li>▪ Information of financial capability of investor</li> <li>▪ Giving them assurance of tremendous future return.</li> </ul>	4	3	12	
RP-02	Availability of finance	3	4	12	Open		03-march	<ul style="list-style-type: none"> <li>▪ Own resources</li> <li>▪ Commitment with financial institution</li> <li>▪ Exclusive management of investor.</li> </ul>	3	3	9	
RP-03	Level of demand for project	3	3	9	Open		08-march	<ul style="list-style-type: none"> <li>▪ Making possibility and identification of low cost and best quality material.</li> <li>▪ Eradication of extra expenses from petty balance.</li> </ul>	2	3	6	
RP-04	Land acquisition (site availability)	3	3	9	Open		13-march	<ul style="list-style-type: none"> <li>▪ Making feasibilities</li> <li>▪ Analysis and interpretation of feasibilities.</li> <li>▪ Possession and legal obligation of land.</li> </ul>	2	2	4	
RP-05	_ High finance costs	2	2	4	Open		15-march	<ul style="list-style-type: none"> <li>▪ Lowering operational expenses and transportation expenses</li> <li>▪ Proper management of current expenses.</li> </ul>	1	2	2	



# 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.

Risk Register												
Risk Id	Risks	Current Risk			Status	Owner	Raised	Mitigation Strategies	Residual Risk			
		Likelihood	Impact	Severity					Likelihood	Impact	Severity	
Category 1: Project selection and Project finance												
RP-01	Financial attraction of project to investors	4	4	16	Open		01-march	<ul style="list-style-type: none"> <li>▪ Data collection</li> <li>▪ Information of financial capability of investor</li> <li>▪ Giving them assurance of tremendous future return.</li> </ul>	4	3	12	
RP-02	Availability of finance	3	4	12	Open		03-march	<ul style="list-style-type: none"> <li>▪ Own resources</li> <li>▪ Commitment with financial institution</li> <li>▪ Exclusive management of investor.</li> </ul>	3	3	9	
RP-03	Level of demand for project	3	3	9	Open		08-march	<ul style="list-style-type: none"> <li>▪ Making possibility and identification of low cost and best quality material.</li> <li>▪ Eradication of extra expenses from petty balance.</li> </ul>	2	3	6	
RP-04	Land acquisition (site availability)	3	3	9	Open		13-march	<ul style="list-style-type: none"> <li>▪ Making feasibilities</li> <li>▪ Analysis and interpretation of feasibilities.</li> <li>▪ Possession and legal obligation of land.</li> </ul>	2	2	4	
RP-05	_ High finance costs	2	2	4	Open		15-march	<ul style="list-style-type: none"> <li>▪ Lowering operational expenses and transportation expenses</li> <li>▪ Proper management of current expenses.</li> </ul>	1	2	2	

# Managing people





# Managing people

---

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



# People management factors

## ◊ Consistency

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

## ◊ Respect

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

## ◊ Inclusion

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

## ◊ Honesty

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



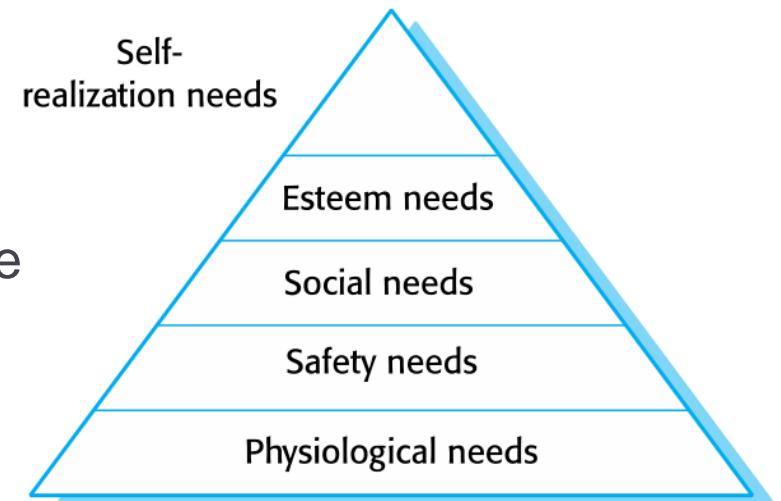
# Motivating people

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



# 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





# 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



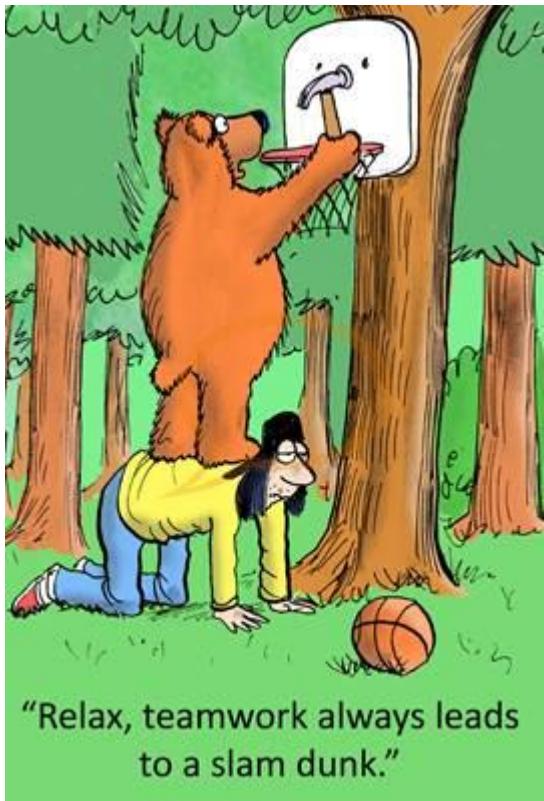
# Personality types

- ◊ 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, play tennis, travel, etc.
- ◊ Interaction-oriented
  - The principal motivation is the presence and actions of co-workers. People go to work because they like to go to work.



# Motivation balance

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



## Teamwork





# 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 organization 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 **organized 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 organization 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

# Project Manager's Next Semester



- ◊ Each team will be assigned a project manager
- ◊ Review project progress and raise issues
- ◊ Can advise on technical issues but cannot write any code
- ◊ Should have access to git-hub but don't need to attend all project meetings
- ◊ Grades for project managers grades are not reliant on senior project team and vice versa
- ◊ Project managers will reach out to teams the first week of the Spring 2023 semester

# Professional Tip of the Day: Junior Software Engineer (Software Engineer I) Average Salary



Reno, Nevada

- \$62,526 to \$88,910 - Average \$75,076

San Francisco, California

- \$77,846 to \$110,694 – Average \$93,471

New York, New York

- \$74,919 to \$106,532 – Average \$89,957

Chicago, Illinois

- \$66,069 to \$93,948 – Average \$79,330

Seattle, Washington

- \$68,094 to \$96,826 – Average \$81,761

Austin, Texas

- \$61,250 to \$87,094 – Average \$73,543

# Professional Tip of the Day: How to Time a Job Transition



- ◊ If you plan to quit your job, choose the date strategically
  - Don't quit and have your last day be before bonus day
  - Don't quit during the holiday season
    - Ideal day to quit is the Monday after New Year's day
  - If you have to quit because a job wants you to start at a particular date, ask them if you can move it
  - Consider choosing your last day around the end of a Sprint
- ◊ Take time off in-between jobs
- ◊ Take time off between graduation and starting your full time job



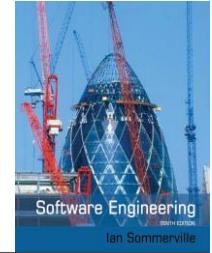
## Chapter 23 – Project planning

Ian Sommerville,

*Software Engineering*, 10<sup>th</sup> Edition

Pearson Education, Addison-Wesley

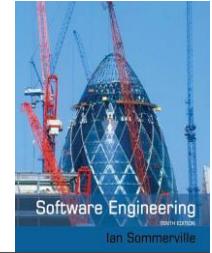
Note: These are a slightly modified version of Chapter 23 slides available from the author's site <http://iansommerville.com/software-engineering-book/>



# 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

## ◊ Planning Occurs During the Project:

- At the **proposal stage**, when you are bidding for a contract to develop or provide a software system.
- During the **project startup phase**(initiation), 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
- ◊ All these items are usually put into a project proposal

# Project startup planning

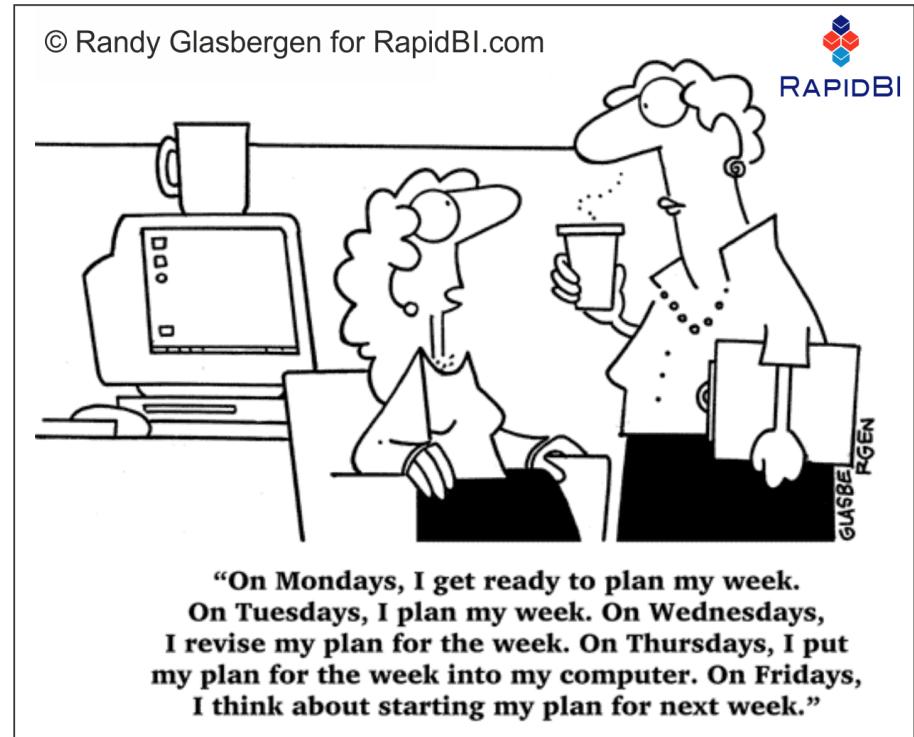


- ◊ At this stage, you know more about the system requirements but do not have design or implementation information
- ◊ Create a plan with enough detail to make decisions about the project budget and staffing.
  - This plan is the basis for project resource allocation
- ◊ The startup plan should also 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

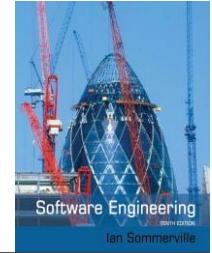


# Pricing to win

- ◊ 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

## Plan-driven development





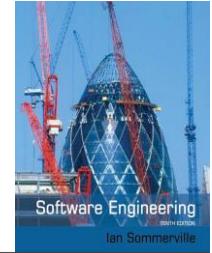
# Plan-driven development

- ◊ 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.
  - Allows for risk mitigation
- ◊ 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.



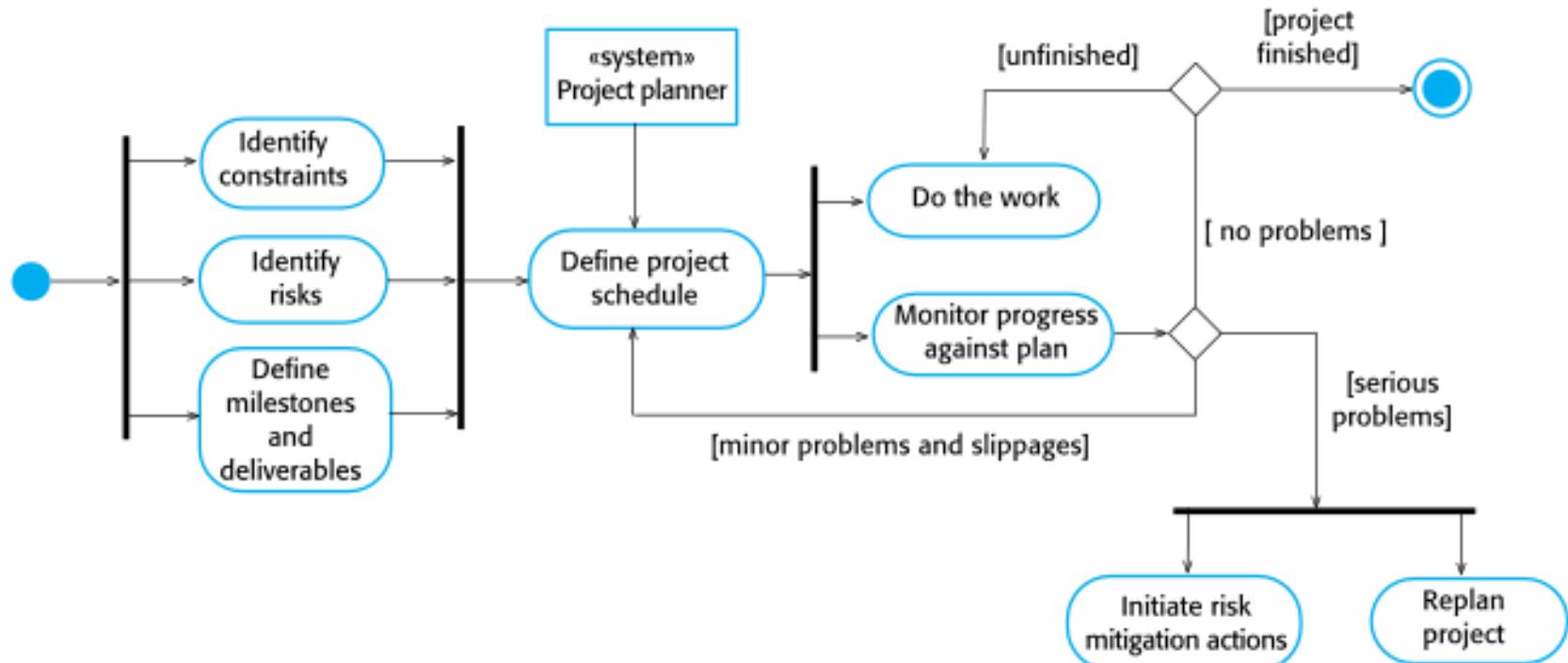
# The planning process

- ◊ 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

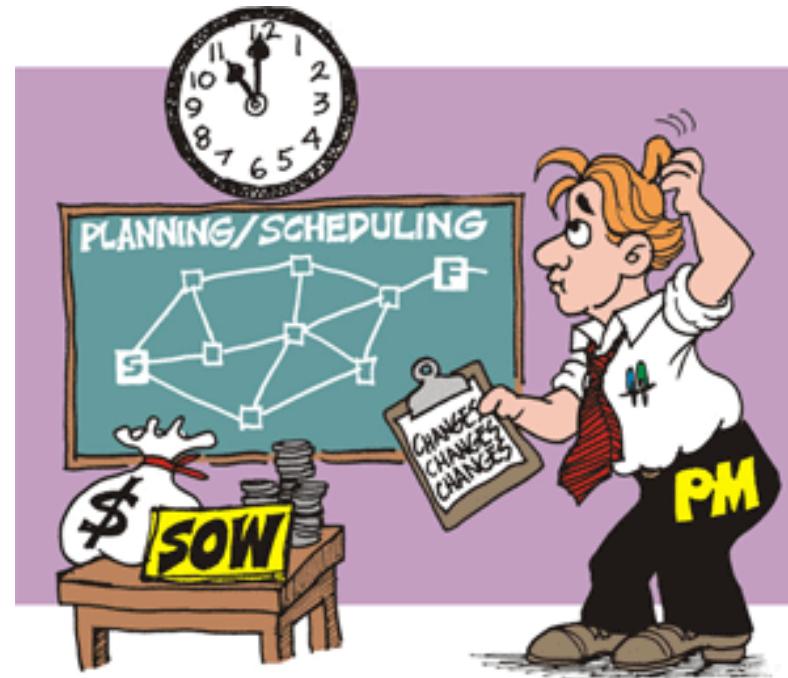
- ◊ You should make realistic rather than optimistic assumptions when you are defining a project plan.
- ◊ Problems of some description always arise during a project, and these lead to project delays.
- ◊ Your initial assumptions and scheduling should therefore take unexpected problems into account.
- ◊ 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.

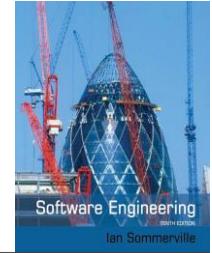
# Project scheduling





# Project scheduling

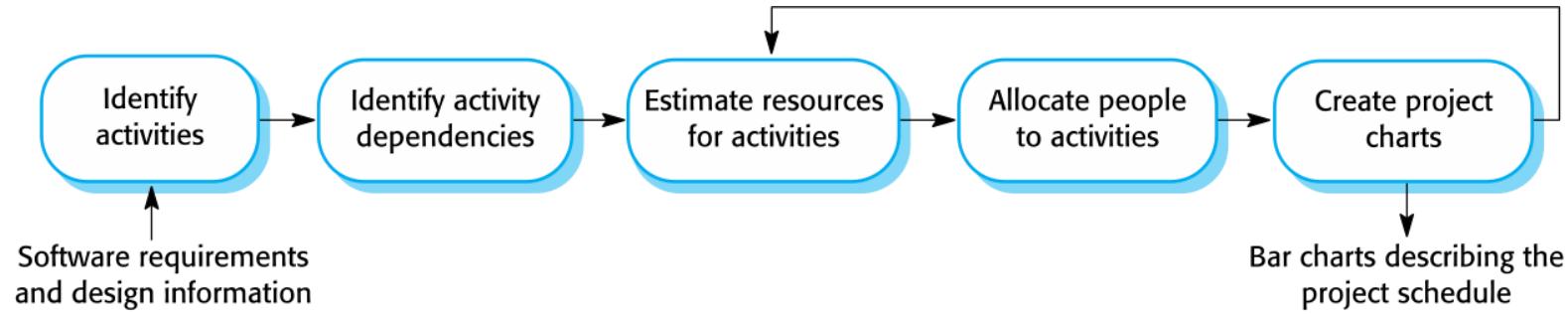
- ◊ Project scheduling is the process of deciding how the work in a project will be organized as separate tasks, and when and how these tasks will be executed.
- ◊ You estimate the calendar time needed to complete each task, the effort required and who will work on the tasks that have been identified.
- ◊ 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





# 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

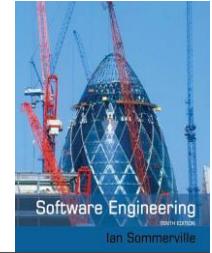
- ◊ Graphical notations are normally used to illustrate the project schedule.
- ◊ These show the project breakdown into tasks. Tasks should not be too small. They should take about a week or two.
- ◊ Calendar-based
  - Bar charts are the most commonly used representation for project schedules. They show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

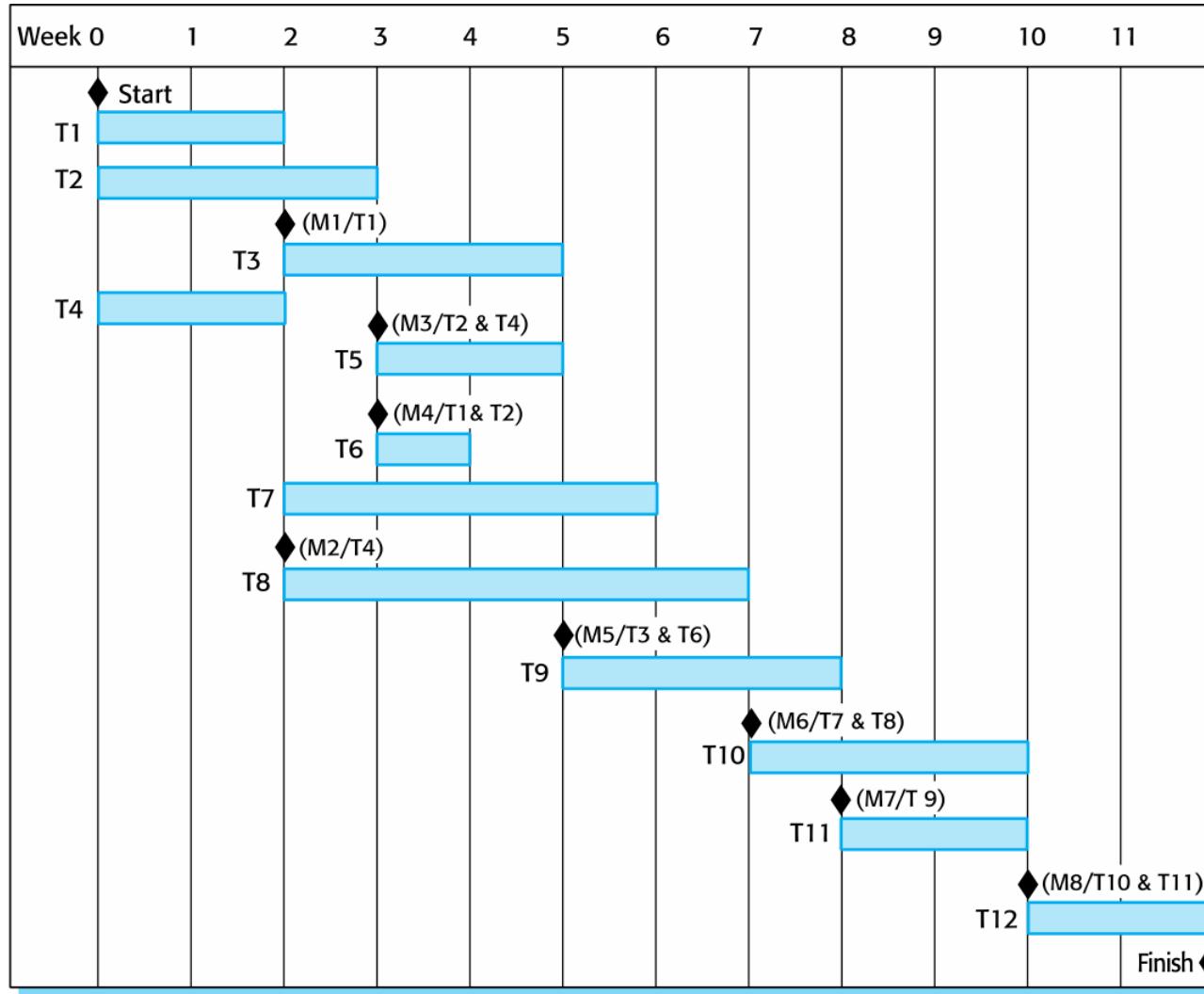


# Tasks, durations, and dependencies

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)

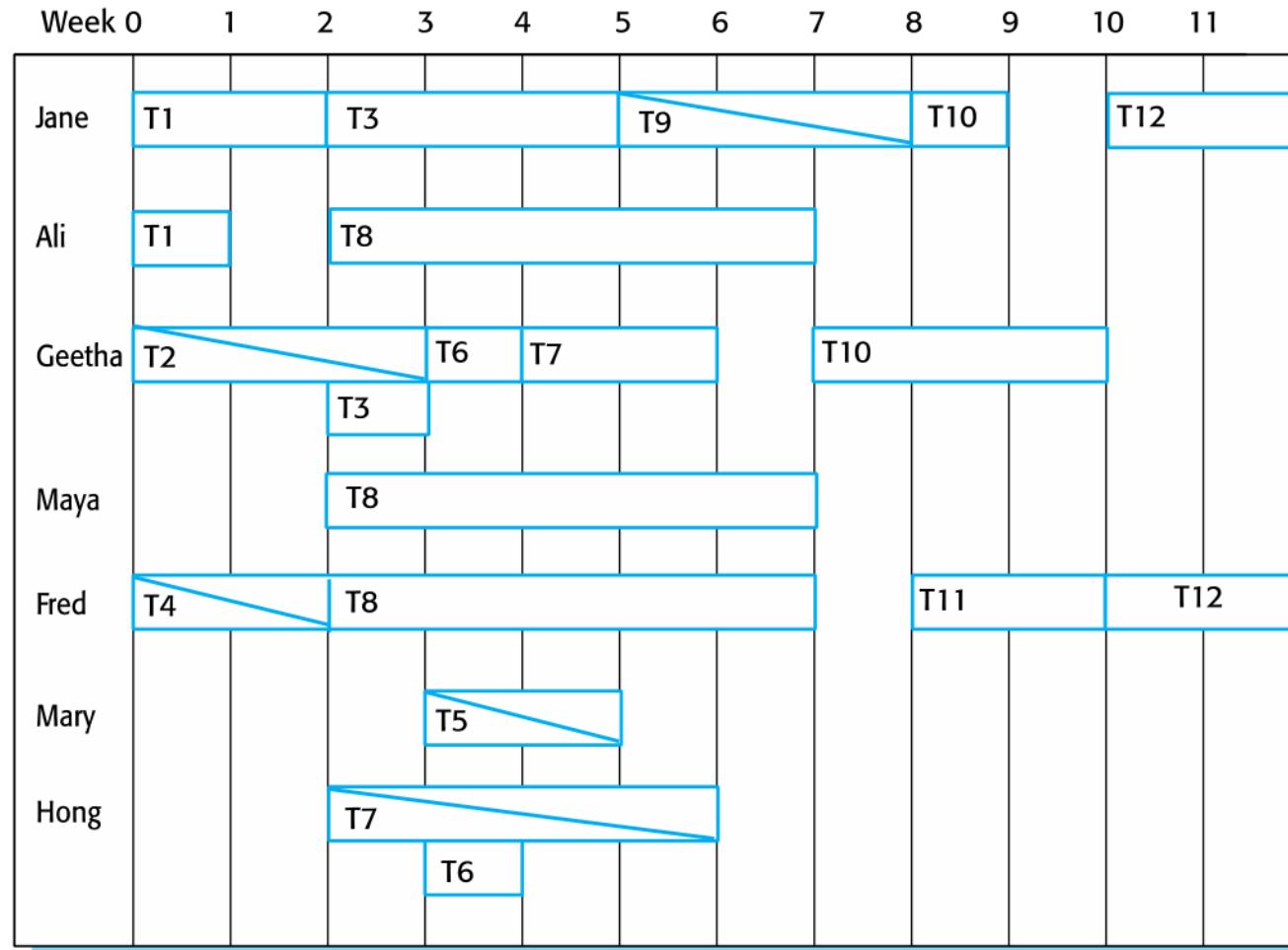


# Activity bar chart





# Staff allocation chart



## Agile planning





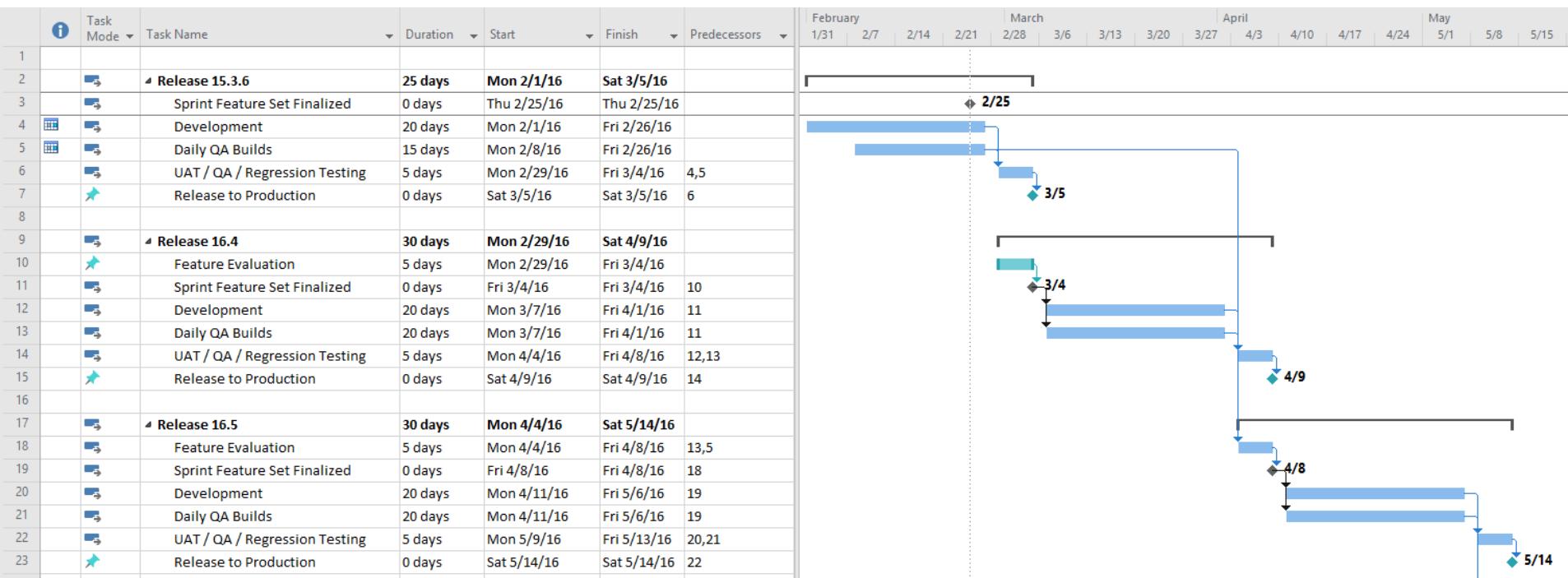
# Agile planning

- ◊ 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.





# Normal Monthly Release

Monthly Sprint 1

Monthly Sprint 2

Monthly Sprint 3

Product and BA Evaluation

Development

Dev Resolution of QA Found Issues

User Acceptance Testing

Daily QA Builds and Testing

QA Regression Testing

Development

Dev Resolution of QA Found Issues

User Acceptance Testing

Daily QA Builds and Testing

QA Regression Testing

Product and BA Evaluation

Development

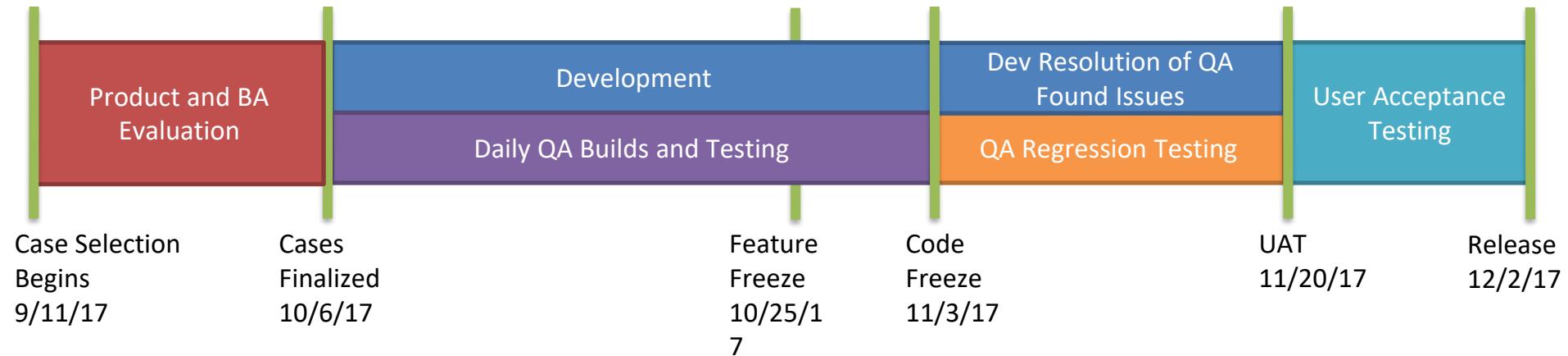
Daily QA Builds and Testing

Product and BA Evaluation

# Sprint Example



Monthly Sprint Example 17.12 (December)





# 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



# What is a User Story?

- ◊ A description of a software feature from an end-user perspective
- ◊ It describes who the user is and what they want and why
- ◊ A user story helps create a simplified description of a requirement and what it should achieve

## RECOMMENDED FORMAT

User stories typically follow a simple template:

*As a <type of user>, I want <some goal> so that <some reason>.*

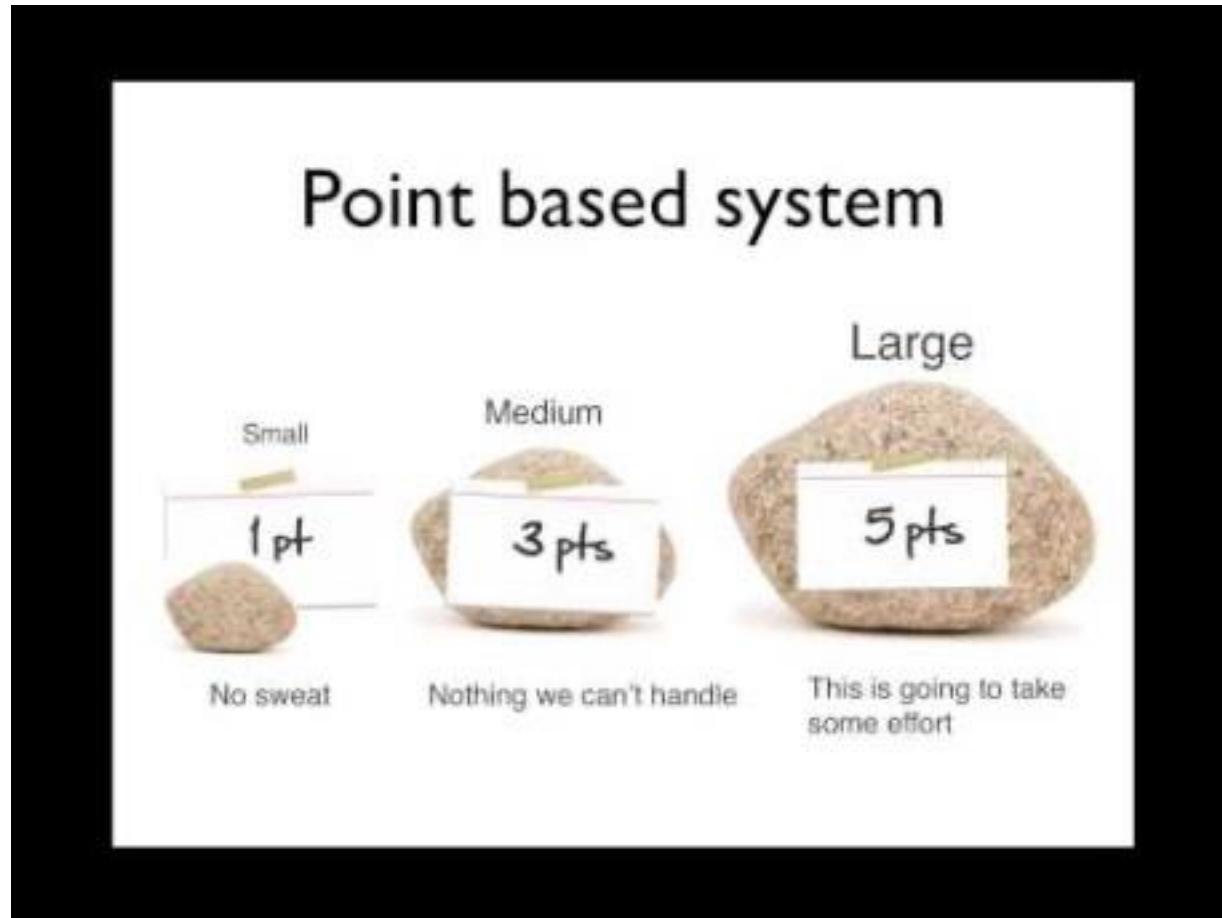


# Story-based planning

- ◊ 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



# Video: Estimation the Fine Art of Guessing



# The planning game





# 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



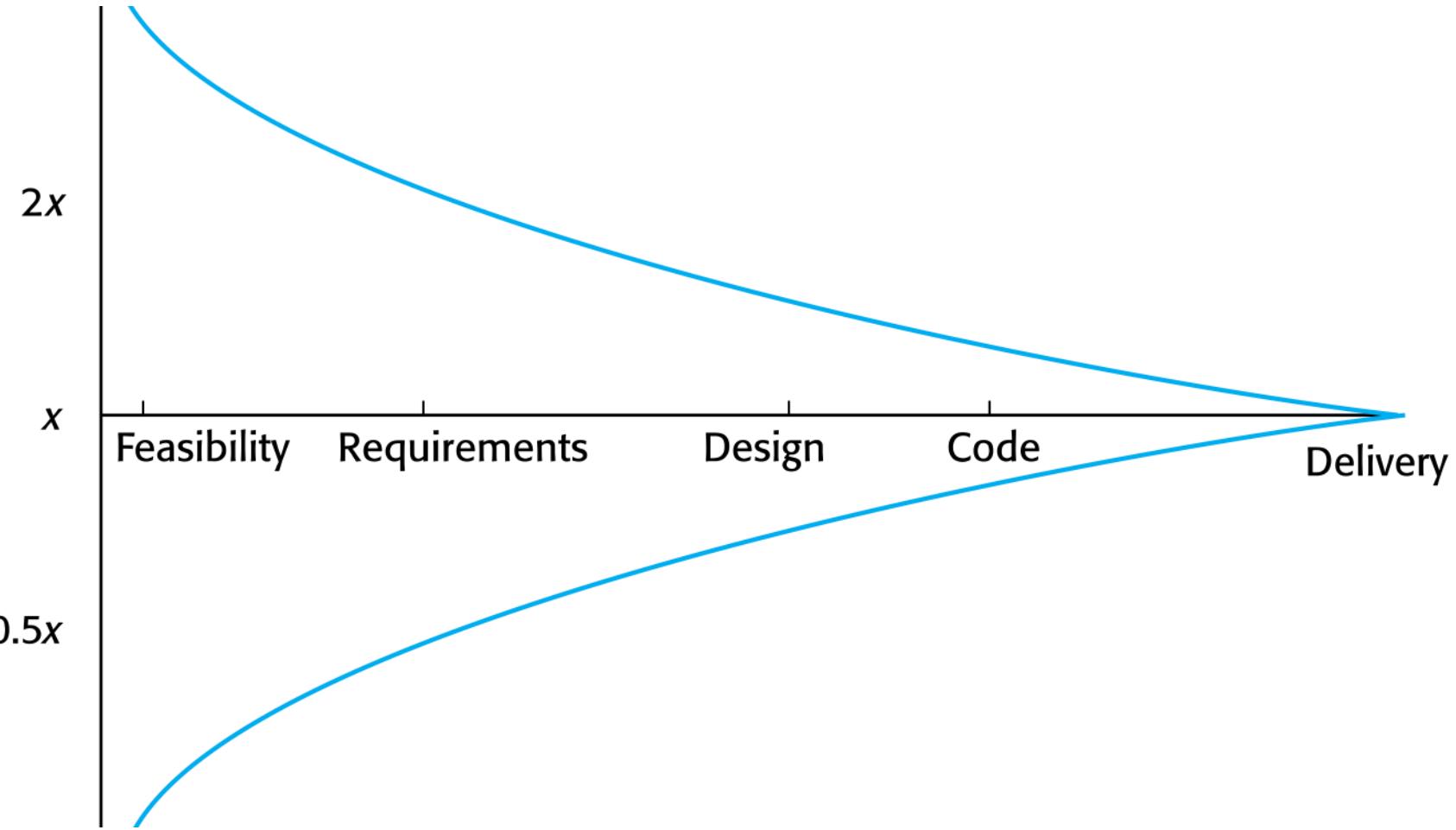


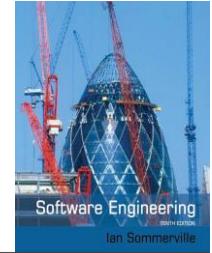
# Estimation techniques

- ◊ 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.



# 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.

# Professional Tip of the Day: The 70/20/10 Rule for Post Tax Income

