



Software Quality Management

Lecture # 1: Introduction to Software Engineering

Topics covered

- ✧ Professional software development
 - What is meant by software engineering.
- ✧ Software engineering ethics
 - A brief introduction to ethical issues that affect software engineering.

Software engineering

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

Software costs

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

Software project failure

✧ *Increasing system complexity*

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

✧ *Failure to use software engineering methods*

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

Frequently asked questions about Software Engineering

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

Frequently asked questions about Software Engineering

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

Software products

✧ Generic products

- Stand-alone systems that are marketed and sold to any customer who wishes to buy them.
- Examples – PC software such as graphics programs, project management tools; CAD software; software for specific markets such as appointments systems for dentists.

✧ Customized (Bespoke) products

- Software that is commissioned by a specific customer to meet their own needs.
- Examples – embedded control systems, air traffic control software, traffic monitoring systems.

Product specification

✧ Generic products

- The specification of what the software should do is owned by the software developer and decisions on software change are made by the developer.

✧ Customized (Bespoke) products

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

Essential attributes of good software

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

Software Engineering

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

Importance of Software Engineering

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

Software Process activities

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

General issues that affect software

✧ Heterogeneity

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

✧ Business and social change

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

General issues that affect software

✧ Security and trust

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

✧ Scale

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

Software engineering diversity

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

Application types (1 / 3)

✧ 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 (2 / 3)

✧ 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 (3 /3)

✧ 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.

Web-based 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 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 have emerged that support the creation of rich interfaces within a web browser / mobile device.

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.

Key Points (1 / 2)

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

Key Points (2 / 2)

- ✧ There are many different types of system and each requires appropriate software engineering tools and techniques for their development.
- ✧ 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.



Software Quality Management

Lecture # 2: Introduction to Software Processes

Software Processes

- ✧ Software process models
- ✧ Process activities

The Software Process

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

Software Process descriptions

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

Plan-driven and Agile processes

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

Software Process models

Software Process models

✧ The Waterfall model

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

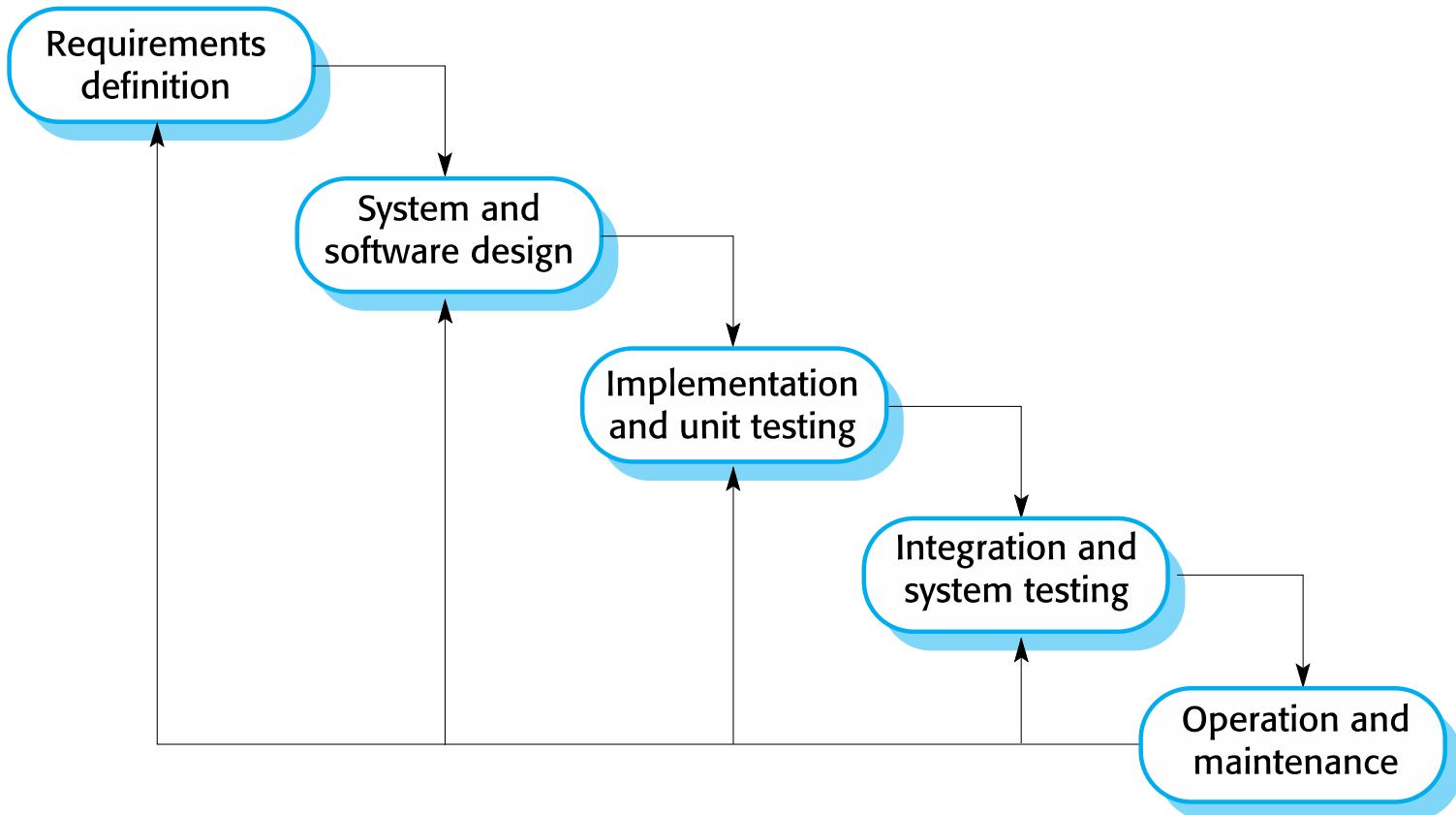
✧ Incremental development

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

✧ Integration and Configuration (Reuse-oriented)

- The system is assembled from existing configurable components. May be plan-driven or agile.
- ✧ In practice, most large systems are developed using a process that incorporates elements from all of these models.

The Waterfall model



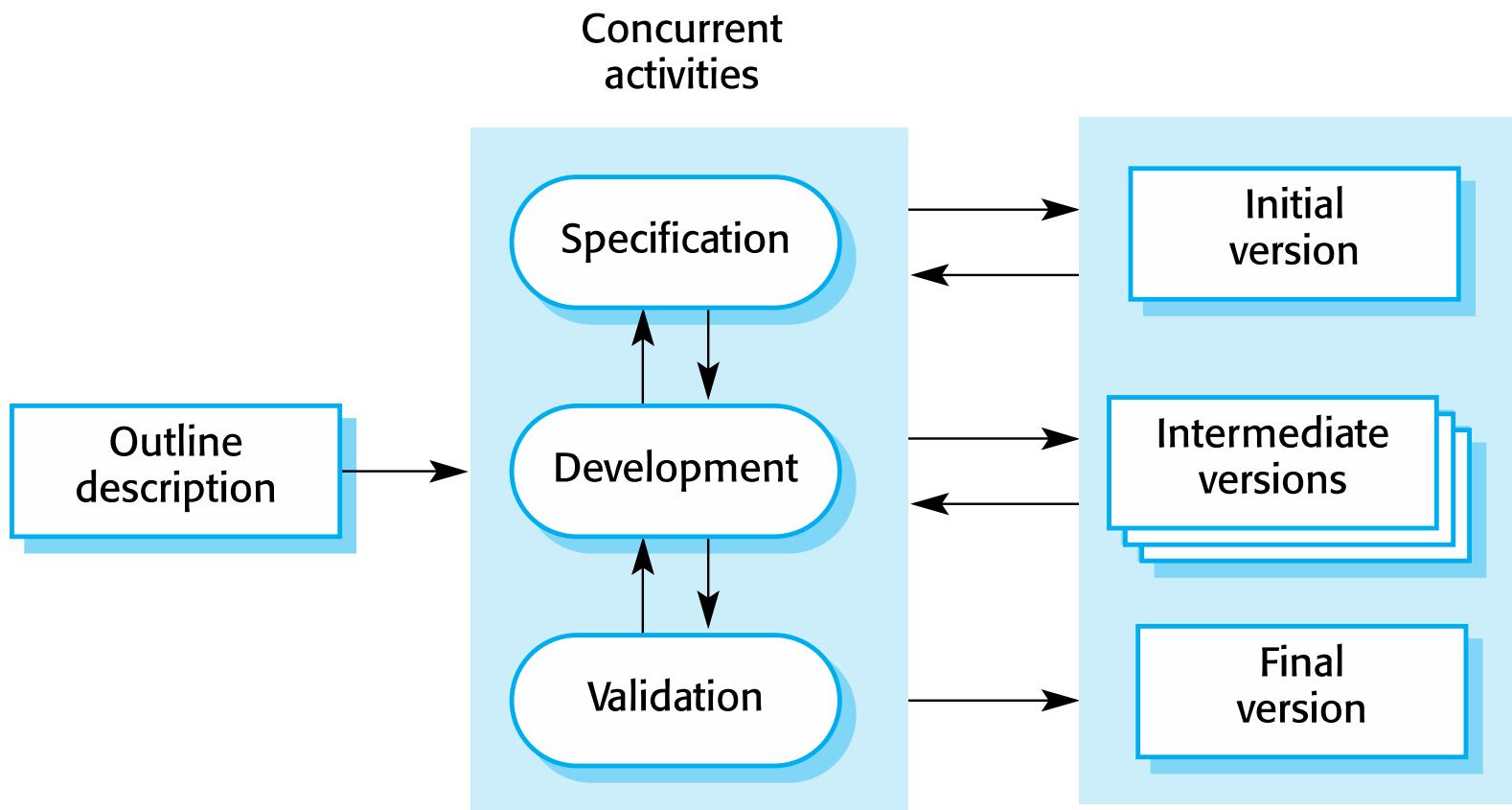
Waterfall model phases

- ✧ There are separate identified phases in the waterfall model:
 - Requirements Analysis and Definition
 - System and Software Design
 - Implementation and Unit Testing
 - Integration and System Testing
 - Operation and Maintenance
- ✧ The main drawback of the waterfall model is the difficulty of accommodating change after the process is underway. In principle, a phase has to be complete before moving onto the next phase.

Waterfall model problems

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

Incremental development



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.

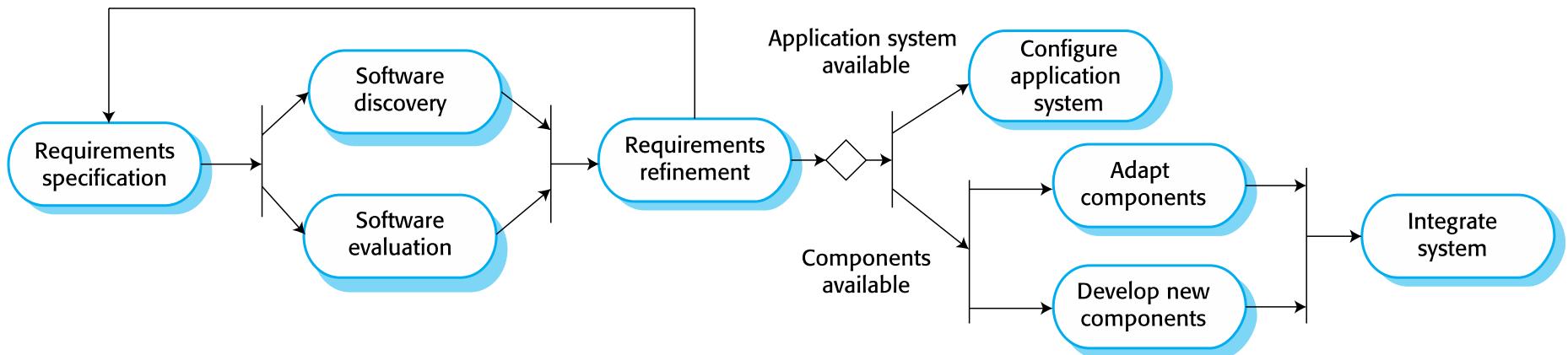
Integration and Configuration (Reuse-oriented)

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

Types of reusable software

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

Reuse-oriented software engineering



Key process stages of **Reuse-oriented development**

1. Requirements specification
2. Software discovery and evaluation
3. Requirements refinement
4. Application system configuration
5. Component adaptation and integration

Reuse-oriented development: Advantages and disadvantages

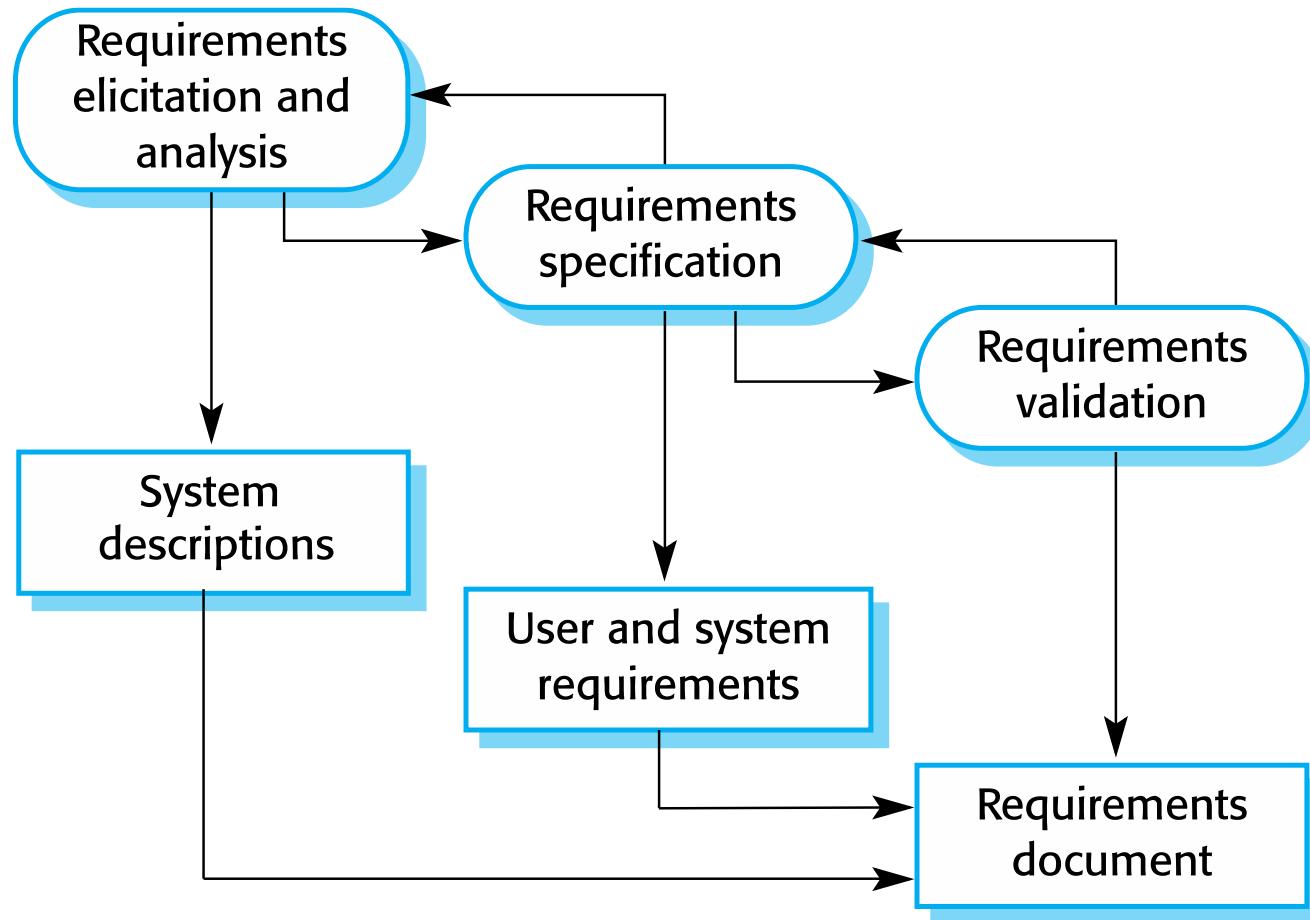
- ✧ Reduced costs and risks as less software is developed from scratch.
- ✧ Faster delivery and deployment of system.
- ✧ But requirements compromises are inevitable so system may not meet real needs of users.
- ✧ Loss of control over evolution of reused system elements.

Software Process activities

Software Process activities

- ✧ **Real software processes are inter-leaved sequences of technical, collaborative and managerial activities** with the overall goal of specifying, designing, implementing and testing a software system.
- ✧ The four basic process activities of **Specification, Development, Validation and Evolution** are organized differently in different development processes.
- ✧ For example, in the **Waterfall** model, they are organized in sequence, whereas in **Incremental development** they are interleaved.

The Requirements Engineering process



Software Specification

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

Software Design and Implementation

✧ The process of converting the system specification into an executable system.

✧ **Software Design**

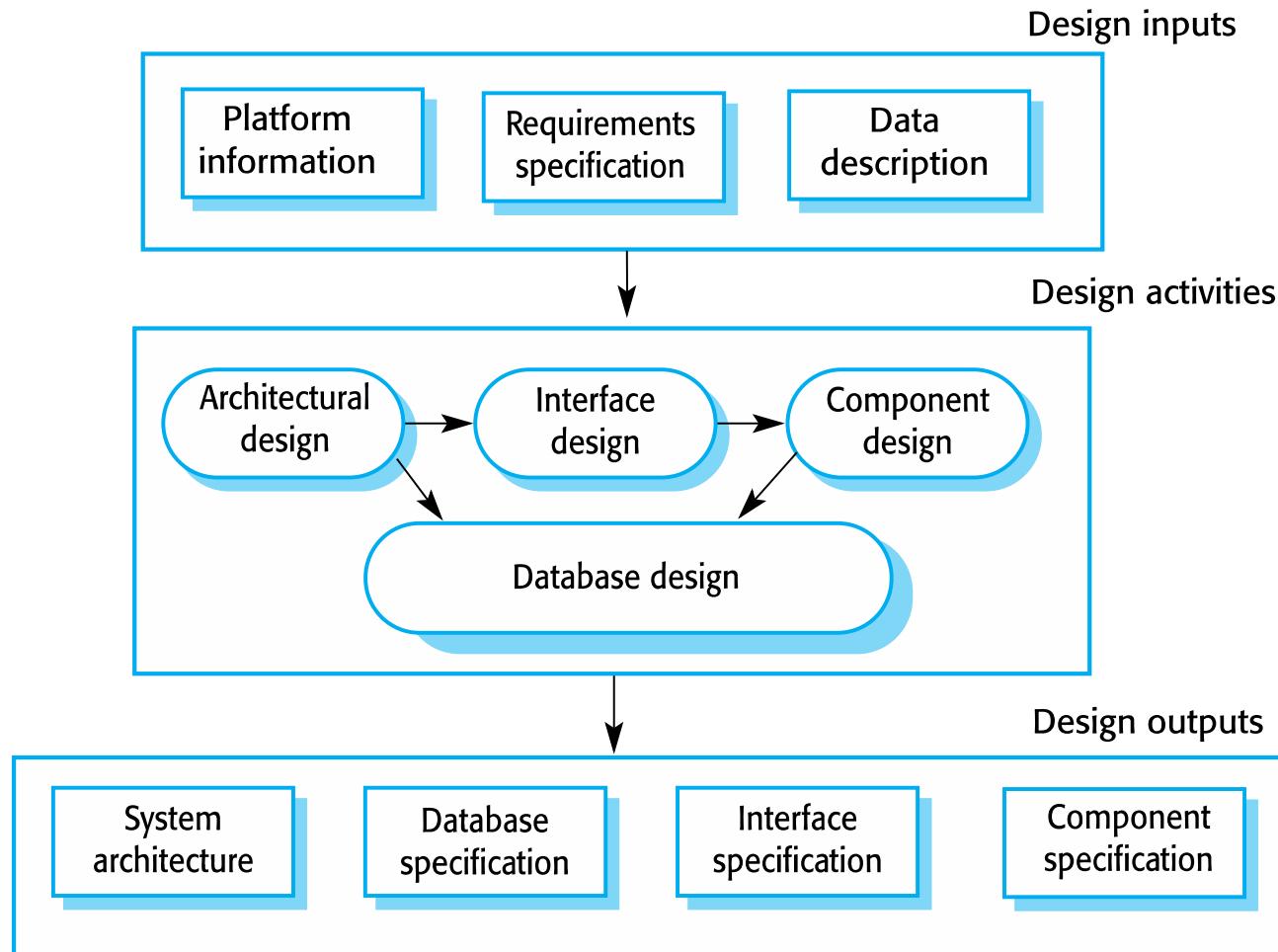
- Design a software structure that realises the specification;

✧ **Implementation**

- Translate this structure into an executable program;

✧ The activities of design and implementation are closely related and may be inter-leaved.

A general model of the design process



Design activities

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

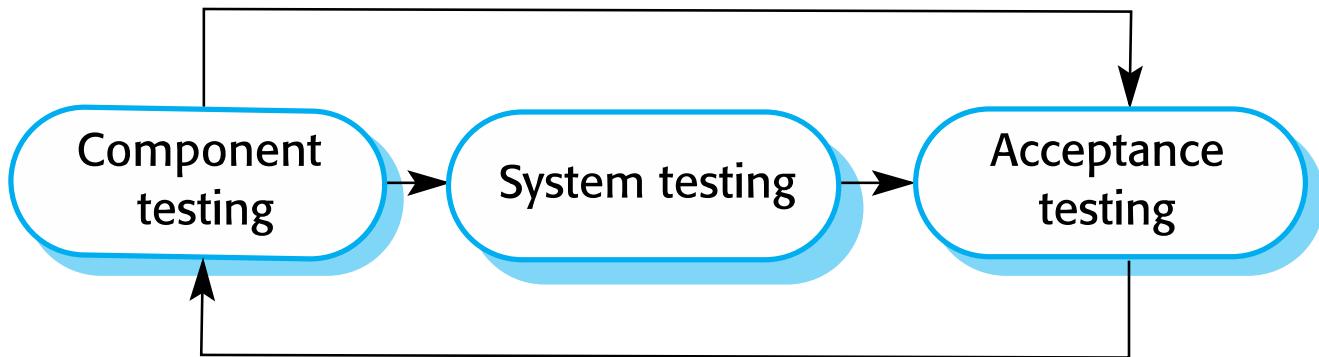
System Implementation

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

Software Validation

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

Stages of Testing



Testing stages

✧ Component testing

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

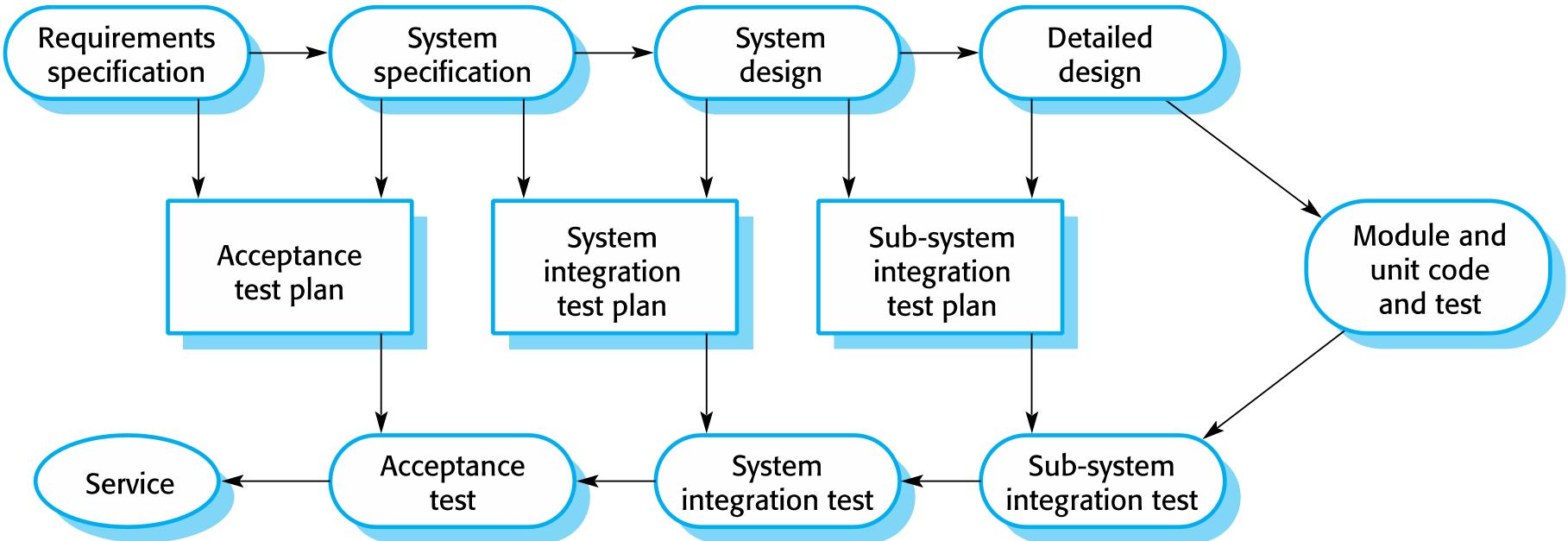
✧ System testing

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

✧ Customer testing

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

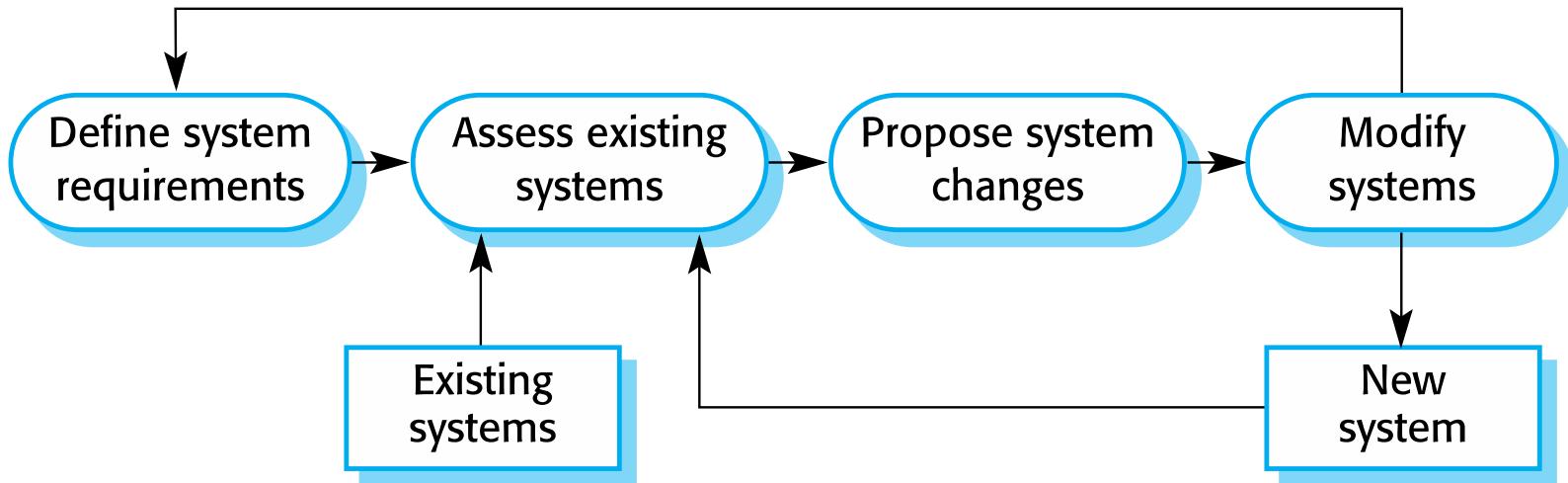
Testing phases in a Plan-driven software process (V-model)



Software evolution

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

System evolution



Software Processes: Key Points (1 / 2)

- ✧ **Software processes** are the activities involved in producing a software system.
- ✧ Software process models are abstract representations of these processes.
- ✧ **General process models** describe the organization of software processes.
 - Examples of these general models include the ‘Waterfall’ model, Incremental development, and Reuse-oriented development.
- ✧ **Requirements engineering** is the process of developing a software specification.

Software Processes: Key Points (2 / 2)

- ✧ **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.

Coping with change

Coping with change

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

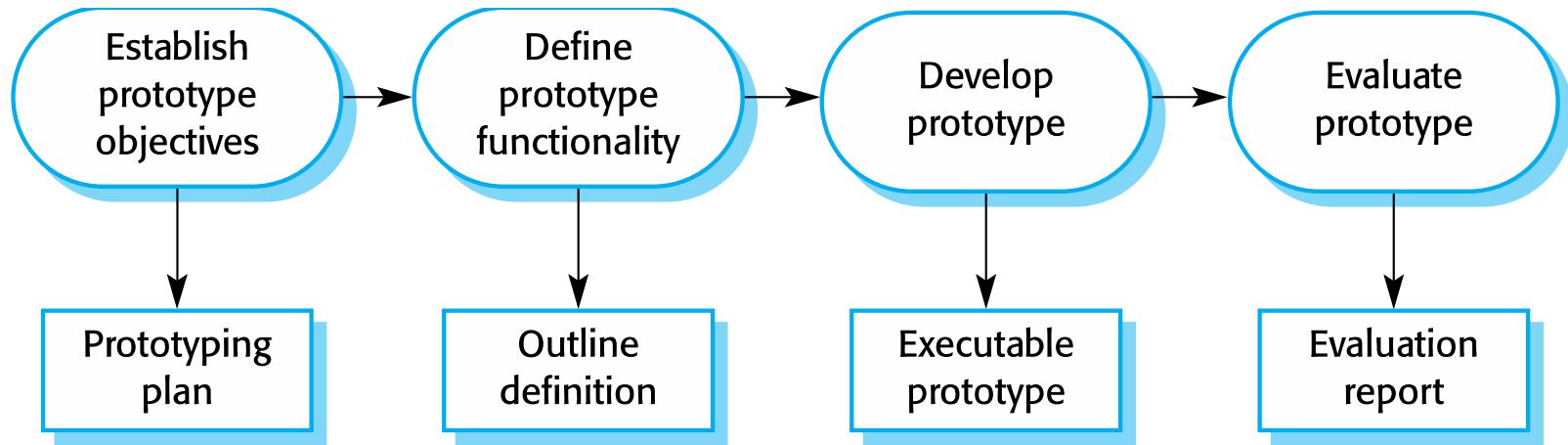
Software prototyping

- ✧ 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 prioritised** and the highest priority requirements are included in early increments.
- ✧ Once the development of an increment is started, the requirements are frozen though requirements for later increments can continue to evolve.

Incremental development and delivery

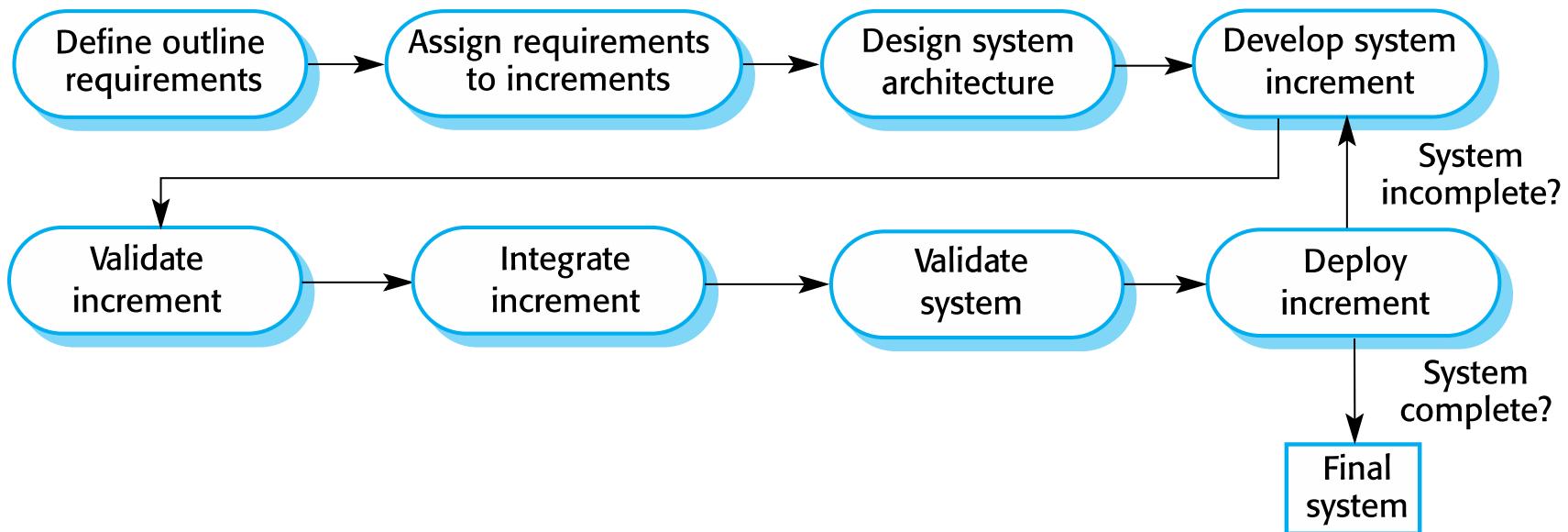
✧ Incremental development

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

✧ Incremental delivery

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

Incremental delivery



Incremental delivery: Advantages

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

Incremental delivery: Problems

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

Software Processes: Key Points (1 / 3)

- ✧ **Software processes** are the **activities** involved in producing a software system.
- ✧ Software process models are abstract representations of these processes.
- ✧ **General process models** describe the organization of software processes.
 - Examples of these general models include the ‘Waterfall’ model, Incremental development, and Reuse-oriented development.
- ✧ **Requirements engineering** is the process of developing a software specification.

Software Processes: Key Points (2 / 3)

- ✧ **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.

Software Processes: Key Points (3 / 3)

- ✧ Processes may be structured for iterative development and delivery so that changes may be made without disrupting the system as a whole.



Software Quality Management

Lecture #3: The Unified Process

The UNIFIED (SOFTWARE DEVELOPMENT) PROCESS

- The Unified Process
- Iteration and incrementation within the object-oriented paradigm
- The Requirements workflow
- The Analysis workflow
- The Design workflow
- The Implementation workflow
- The Test workflow

Unified Process: Overview (continued)

Slide 3.99

- Postdelivery maintenance
- Retirement
- The phases of the Unified Process
- One- versus two-dimensional life-cycle models
- Why two-dimensional process?

3.1 The Unified Process

Slide 3.100

- Until late 1990s, three of the most successful object-oriented methodologies were
 - ▶ Booch's method
 - ▶ Jacobson's Objectory
 - ▶ Rumbaugh's OMT

- In 1999, Booch, Jacobson, and Rumbaugh published a complete object-oriented analysis and design methodology that **unified** their three separate methodologies
 - ▶ Original name: *Rational Unified Process* (RUP)
 - ▶ Next name: *Unified Software Development Process*
 - ▶ Name used today: **Unified Process** (for brevity)

- The Unified Process ***is not a series of steps*** for constructing a software product
 - ▶ No such single “one size fits all” methodology could exist
 - ▶ There is a wide variety of different types of software
- The Unified Process is an adaptable methodology
 - ▶ It has to be modified for the specific software product to be developed

- Unified Modelling Language (UML) is graphical
 - ▶ A picture is worth a thousand words
- UML diagrams enable software engineers to communicate quickly and accurately

- The Unified Process is a **modeling** technique
 - ▶ A **model** is a set of UML diagrams that represent various aspects of the software product we want to develop
- UML stands for **Unified Modeling** Language
 - ▶ UML is the tool that we use to **represent** (model) the target software product

- The object-oriented paradigm is **iterative** and **incremental** in nature
 - ▶ There is no alternative to repeated **iteration** and **incrementation** until the UML diagrams are satisfactory.

Iteration and Incrementation

Slide 3.106

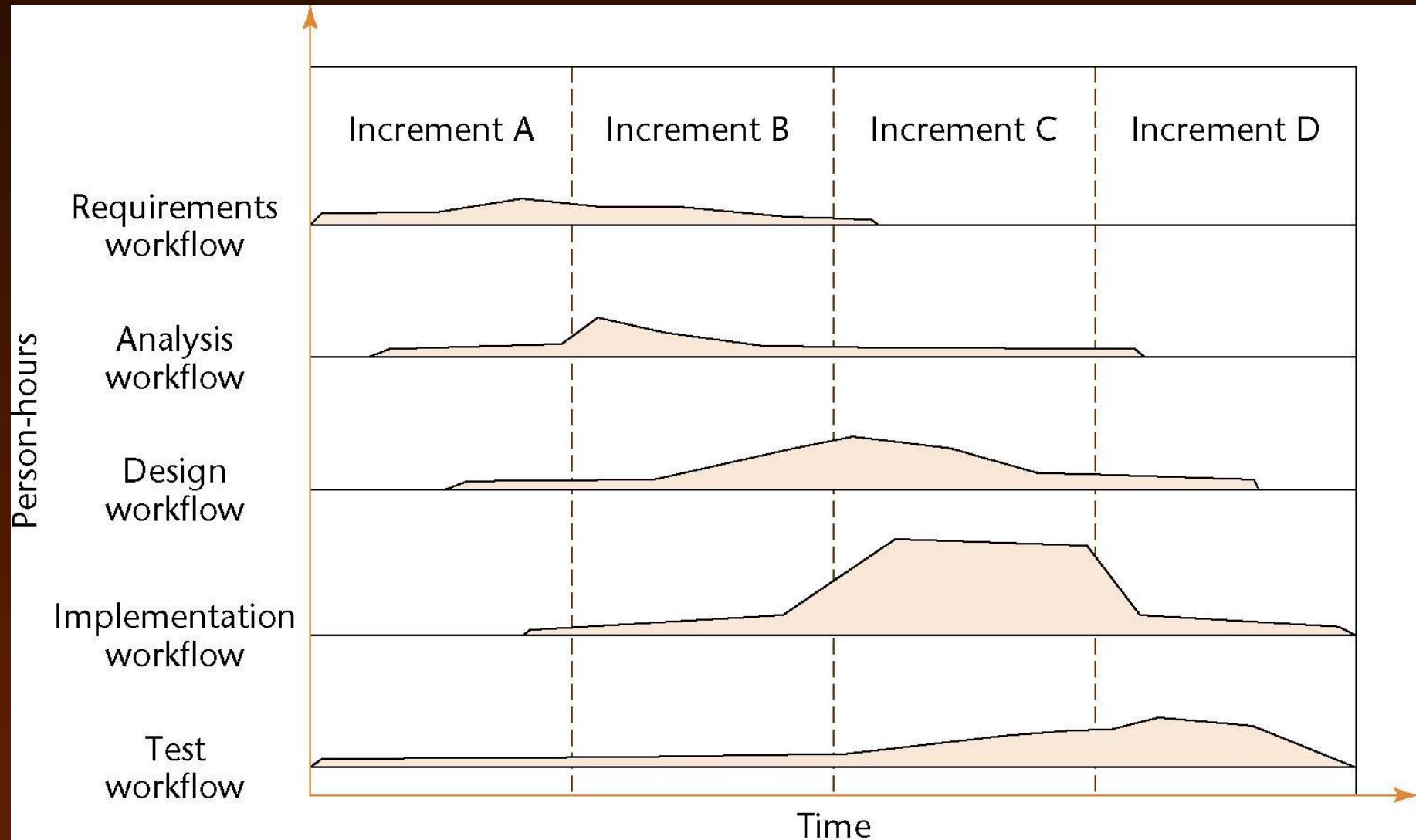


Figure 2.4

3.10 The Phases of the Unified Process

Slide 3.107

- The *increments* are identified as *phases*

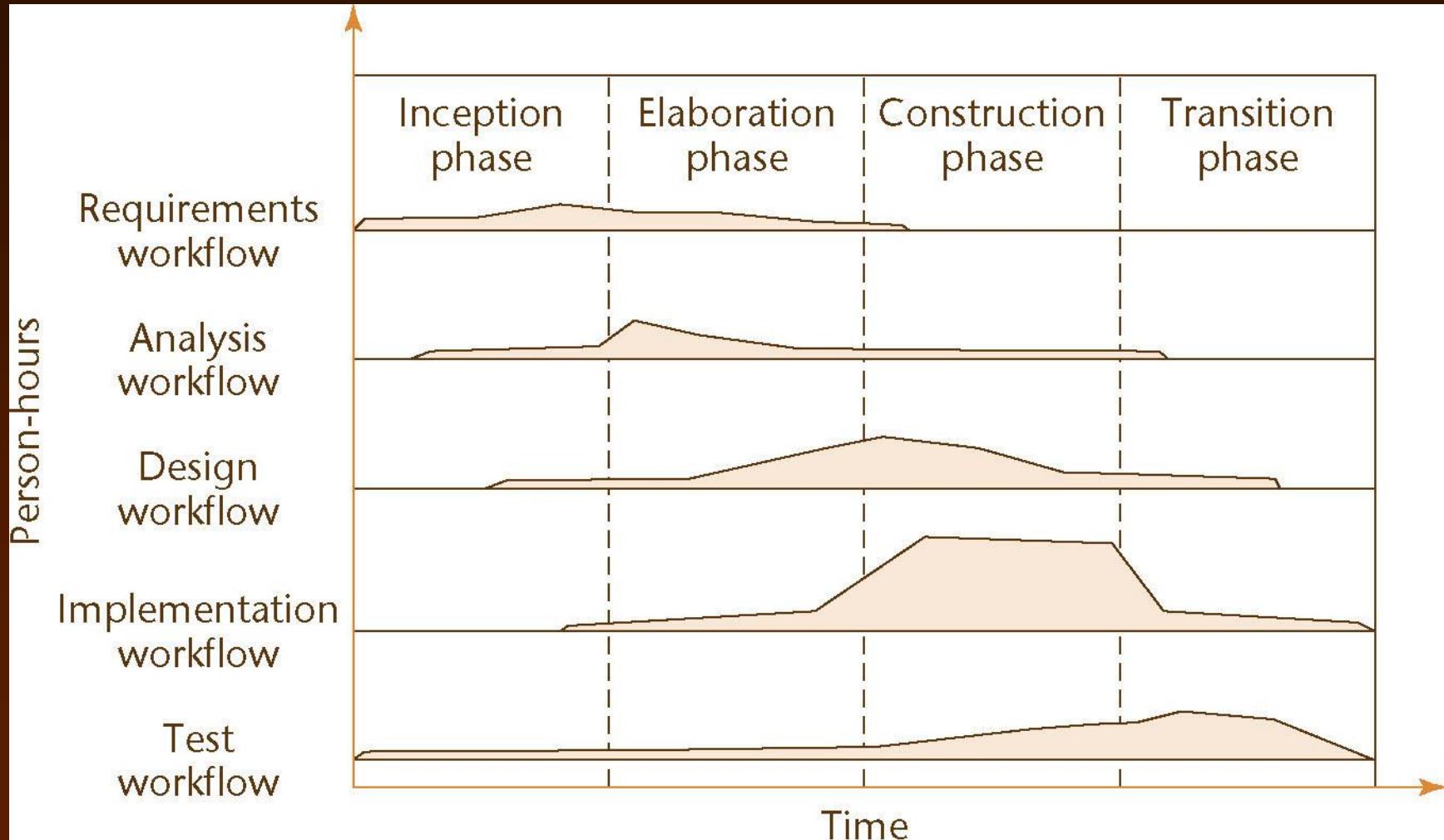


Figure 3.1

Iteration and Incrementation

Slide 3.108

- Iteration is performed during each incrementation

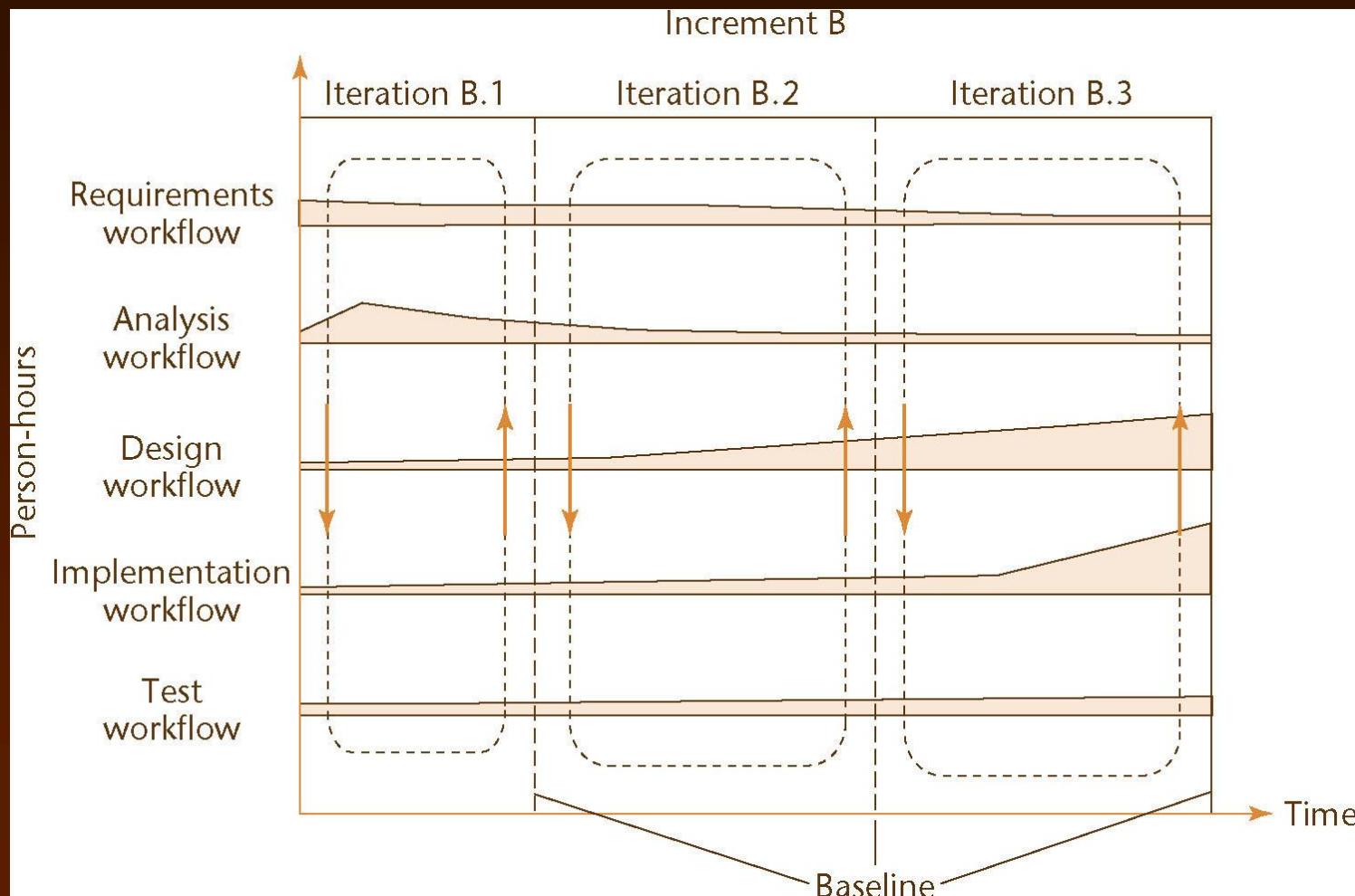


Figure 2.5

Iteration and Incrementation

Slide 3.109

- Iteration and incrementation are used in conjunction with one another
 - ▶ There is no single “requirements phase” or “design phase”
 - ▶ Instead, there are multiple instances of each phase

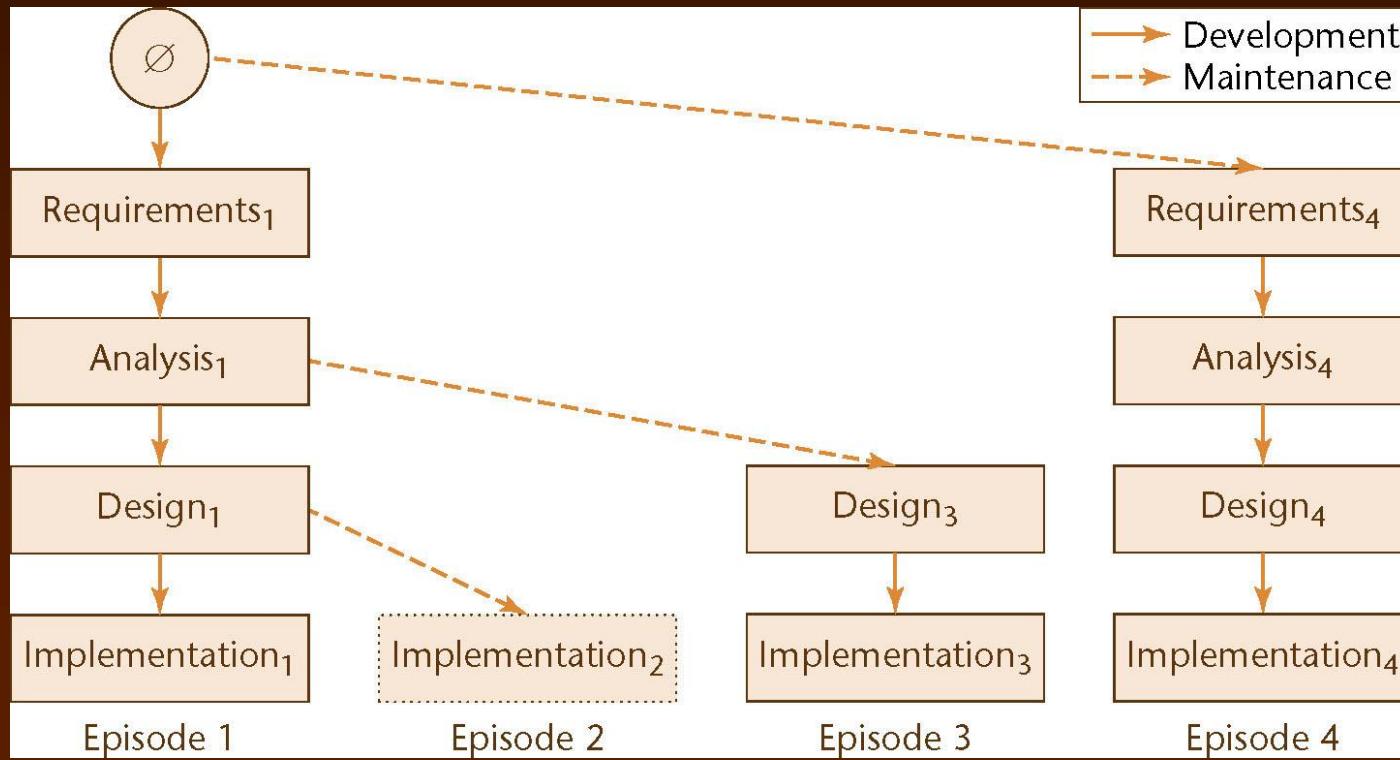


Figure 2.6

Iteration and Incrementation

Slide 3.110

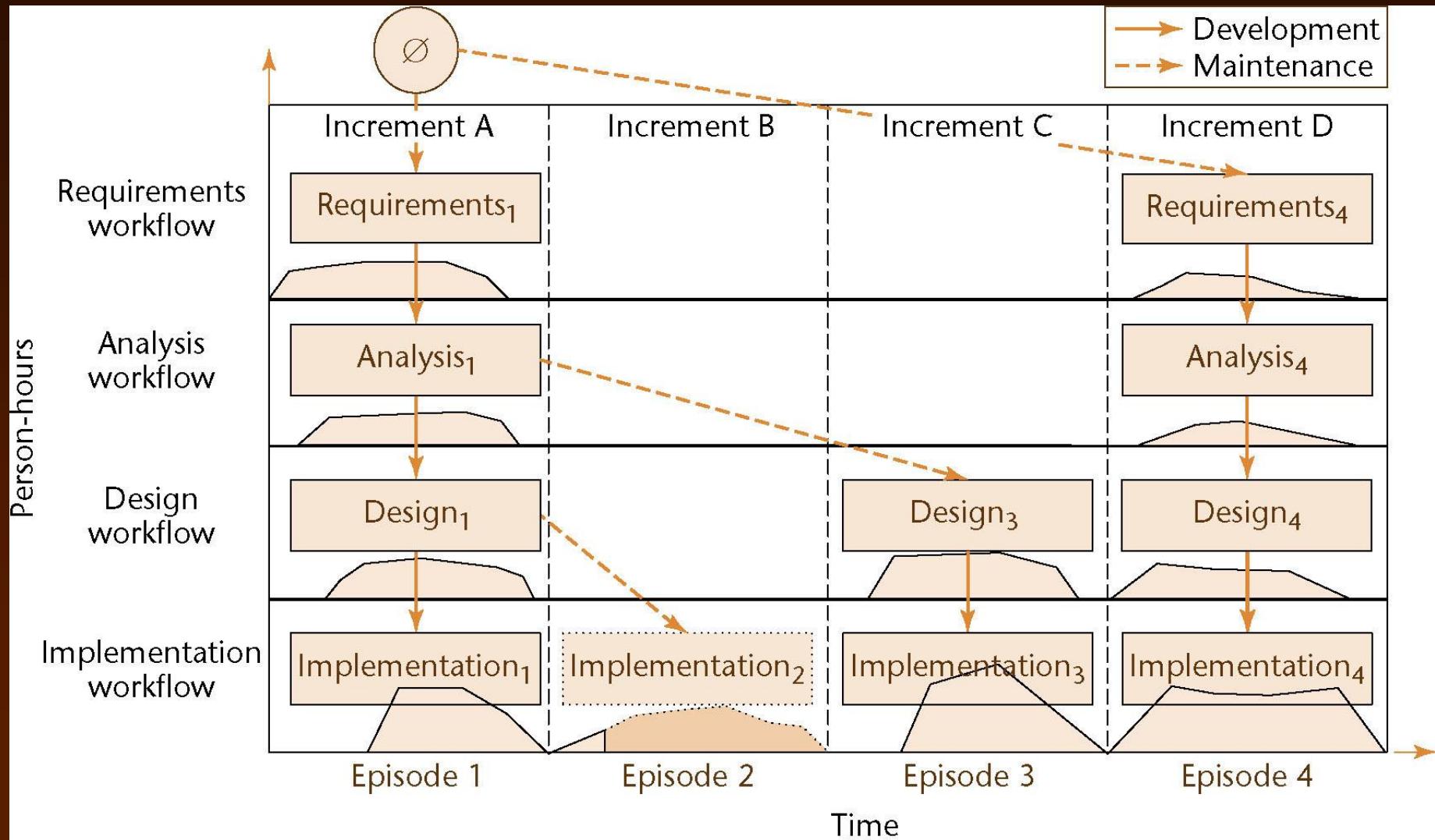


Figure 3.1

More on Incrementation

Slide 3.111

- Each episode corresponds to an **increment**
- Not every increment includes every workflow
- Increment B was not completed
- Dashed lines denote **maintenance**
 - ▶ Episodes 2, 3: Corrective maintenance
 - ▶ Episode 4: Perfective maintenance

Other Aspects of Iteration and Incrementation

Slide 3.112

- We can consider the project as a whole as a set of mini projects (increments)
- Each mini project extends the
 - ▶ Requirements artifacts
 - ▶ Analysis artifacts
 - ▶ Design artifacts
 - ▶ Implementation artifacts
 - ▶ Testing artifacts
- The final set of artifacts is the complete product

Other Aspects of Iteration and Incrementation

Slide 3.113

- During each mini project, we
 - ▶ Extend the artifacts (**incrementation**);
 - ▶ Check the artifacts (test workflow); and
 - ▶ If necessary, change the relevant artifacts (**iteration**);

- Each iteration can be viewed as a small but complete **waterfall life-cycle model**
- During each iteration we select a portion of the software product
- On that portion we perform the
 - ▶ Classical requirements phase
 - ▶ Classical analysis phase
 - ▶ Classical design phase
 - ▶ Classical implementation phase

- The Unified Process
- Iteration and incrementation within the object-oriented paradigm
- The Requirements workflow
- The Analysis workflow
- The Design workflow
- The Implementation workflow
- The Test workflow

3.3 The Requirements Workflow

Slide 3.116

- The aim of the Requirements Workflow
 - ▶ To determine the client's needs

Overview of the Requirements Workflow

Slide 3.117

- First, gain an understanding of the *application domain* (or *domain*, for short)
 - ▶ That is, the specific business environment in which the software product is to operate
- Second, build a **business model**
 - ▶ Use UML to describe the client's business processes
 - ▶ If at any time the client does not feel that the cost is justified, development terminates immediately

Overview of the Requirements Workflow (contd)

Slide 3.118

- It is vital to determine the client's constraints
 - ▶ Deadline
 - **Nowadays, software products are often mission critical**
 - ▶ Parallel running
 - ▶ Portability
 - ▶ Reliability
 - ▶ Rapid response time
 - ▶ Cost
 - **The client will rarely inform the developer how much money is available**
 - **A bidding procedure is used instead**

Overview of the Requirements Workflow (contd)

Slide 3.119

- The aim of this *concept exploration* is to determine
 - ▶ What the client needs
 - ▶ *Not* what the client wants

3.4 The Analysis Workflow

Slide 3.120

- The aim of the analysis workflow
 - ▶ To analyze and refine the requirements
- Why not do this during the requirements workflow?
 - ▶ The requirements artifacts must be totally comprehensible by the client
- The artifacts of the requirements workflow must therefore be expressed in a natural (human) language
 - ▶ All natural languages are imprecise

The Analysis Workflow (contd)

Slide 3.121

- Example from a manufacturing information system:
 - ▶ “A part record and a plant record are read from the database. If **it** contains the letter A directly followed by the letter Q, then calculate the cost of transporting that part to that plant”
- To what does **it** refer?
 - ▶ The part record?
 - ▶ The plant record?
 - ▶ Or the database?

The Analysis Workflow (contd)

Slide 3.122

- Two separate workflows are needed
 - ▶ The **requirements artifacts** must be expressed in the language of the client
 - ▶ The **analysis artifacts** must be precise, and complete enough for the designers

The Specification Document (contd)

Slide 3.123

- Specification document (“specifications”)
 - ▶ It constitutes a contract
 - ▶ It must not have imprecise phrases like “optimal,” or “98% complete”
- Having complete and correct specifications is essential for
 - ▶ Testing and
 - ▶ Maintenance

The Specification Document (continued)

Slide 3.124

- The specification document must not have
 - ▶ Ambiguities
 - ▶ Contradictions
 - ▶ Omissions
 - ▶ Incompleteness
 - ▶ Errors

Software Project Management Plan

Slide 3.125

- Once the client has signed off the specifications, detailed planning and estimating begins
- We draw up the software project management plan, including
 - Cost estimate
 - Duration estimate
 - Deliverables
 - Milestones
 - Budget
- This is the earliest possible time for the SPMP

3.5 The Design Workflow

Slide 3.126

- The aim of the Design Workflow is to refine the analysis workflow until the material is in a form that can be implemented by the programmers
 - ▶ Many **nonfunctional requirements** need to be finalized at this time, including
 - Choice of programming language
 - Reuse issues
 - Portability issues

- Architectural design
 - ▶ Decompose the product into modules
- Detailed design
 - ▶ Design each module:
 - Data structures
 - Algorithms

- Classes are extracted during the object-oriented analysis workflow and
 - ▶ Designed during the **design workflow**
- Accordingly
 - ▶ Classical architectural design corresponds to part of the object-oriented analysis workflow
 - ▶ Classical detailed design corresponds to part of the object-oriented design workflow

The Design Workflow (contd)

Slide 3.129

- Retain design decisions
 - ▶ For when a dead-end is reached
 - ▶ To prevent the maintenance team reinventing the wheel

3.6 The Implementation Workflow

Slide 3.130

- The aim of the Implementation Workflow is to implement the target software product in the selected implementation language
 - ▶ A large software product is partitioned into subsystems
 - ▶ The subsystems consist of *components* or *code artifacts*

3.7 The Test Workflow

Slide 3.131

- The **Test Workflow** is the responsibility of
 - ▶ *Every* developer and maintainer, and
 - ▶ The Software Quality Assurance group
- Traceability of artifacts is an important requirement for successful testing

3.7.1 Requirements Artifacts

Slide 3.132

- Every item in the analysis artifacts must be traceable to an item in the requirements artifacts
 - ▶ Similarly for the design and implementation artifacts

3.7.2 Analysis Artifacts

Slide 3.133

- The analysis artifacts should be checked by means of a review
 - ▶ Representatives of the client and analysis team must be present
- The Software Project Management Plan (SPMP) must be similarly checked
 - ▶ Pay special attention to the cost and duration estimates

3.7.3 Design Artifacts

Slide 3.134

- Design reviews are essential
 - ▶ A client representative is not usually present

3.7.4 Implementation Artifacts

Slide 3.135

- Each component is tested as soon as it has been implemented
 - ▶ *Unit testing*
- At the end of each iteration, the completed components are combined and tested
 - ▶ *Integration testing*
- When the product appears to be complete, it is tested as a whole
 - ▶ *Product testing*
- Once the completed product has been installed on the client's computer, the client tests it
 - ▶ *Acceptance testing*

- COTS software is released for testing by prospective clients
 - ▶ Alpha release
 - ▶ Beta release
- There are advantages and disadvantages to being an alpha or beta release site

3.8 Postdelivery Maintenance

Slide 3.137

- Postdelivery maintenance is an essential component of software development
 - ▶ More money is spent on postdelivery maintenance than on all other activities combined
- Problems can be caused by
 - ▶ Lack of documentation of all kinds

Postdelivery Maintenance (contd)

Slide 3.138

- Two types of testing are needed
 - ▶ Testing the changes made during postdelivery maintenance
 - ▶ Regression testing
- All previous test cases (and their expected outcomes) need to be retained

3.9 Retirement

Slide 3.139

- Software can be unmaintainable because
 - ▶ A drastic change in design has occurred
 - ▶ The product must be implemented on a totally new hardware/operating system
 - ▶ Documentation is missing or inaccurate
 - ▶ Hardware is to be changed — it may be cheaper to rewrite the software from scratch than to modify it
- These are instances of maintenance (rewriting of existing software)

- True retirement is a rare event
- It occurs when the client organization no longer needs the functionality provided by the product

The Phases of the Unified Process (contd)

Slide 3.141

- The phases of the Unified Process are the increments
- The ***four increments*** are labeled
 - ▶ Inception phase
 - ▶ Elaboration phase
 - ▶ Construction phase
 - ▶ Transition phase

The Phases of the Unified Process (contd)

Slide 3.142

- In theory, there could be any number of increments
 - ▶ In practice, development seems to consist of four increments. Hence, four phases are suggested.

Classical Phases versus Workflows

Slide 3.143

- Sequential phases do not exist in the real world
- Instead, the **five core workflows (activities)** are performed over the entire life cycle
 - ▶ Requirements workflow
 - ▶ Analysis workflow
 - ▶ Design workflow
 - ▶ Implementation workflow
 - ▶ Test workflow

The Phases of the Unified Process

Slide 3.144

- The *increments* are identified as *phases*

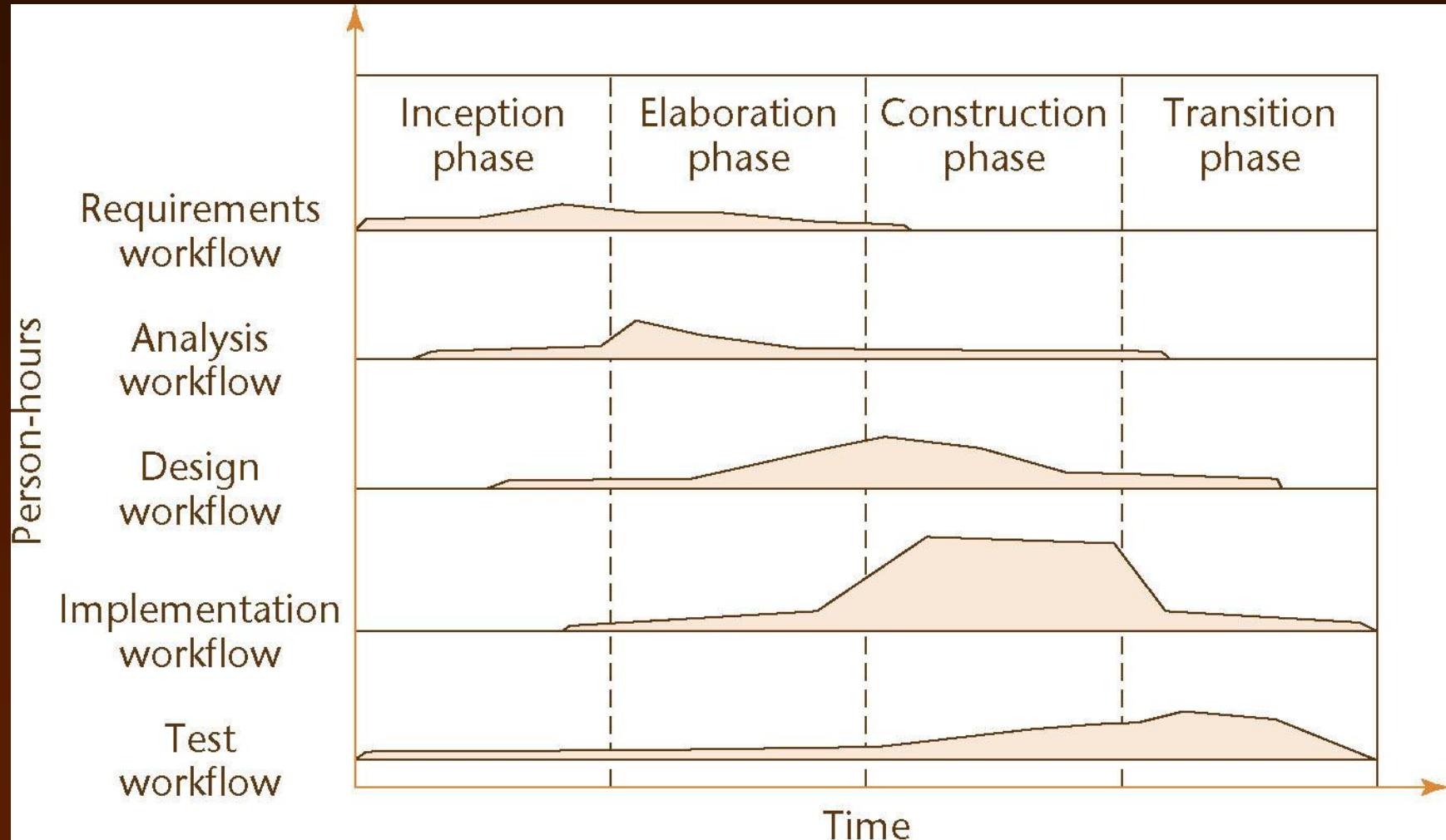


Figure 3.1

Iteration and Incrementation

Slide 3.145

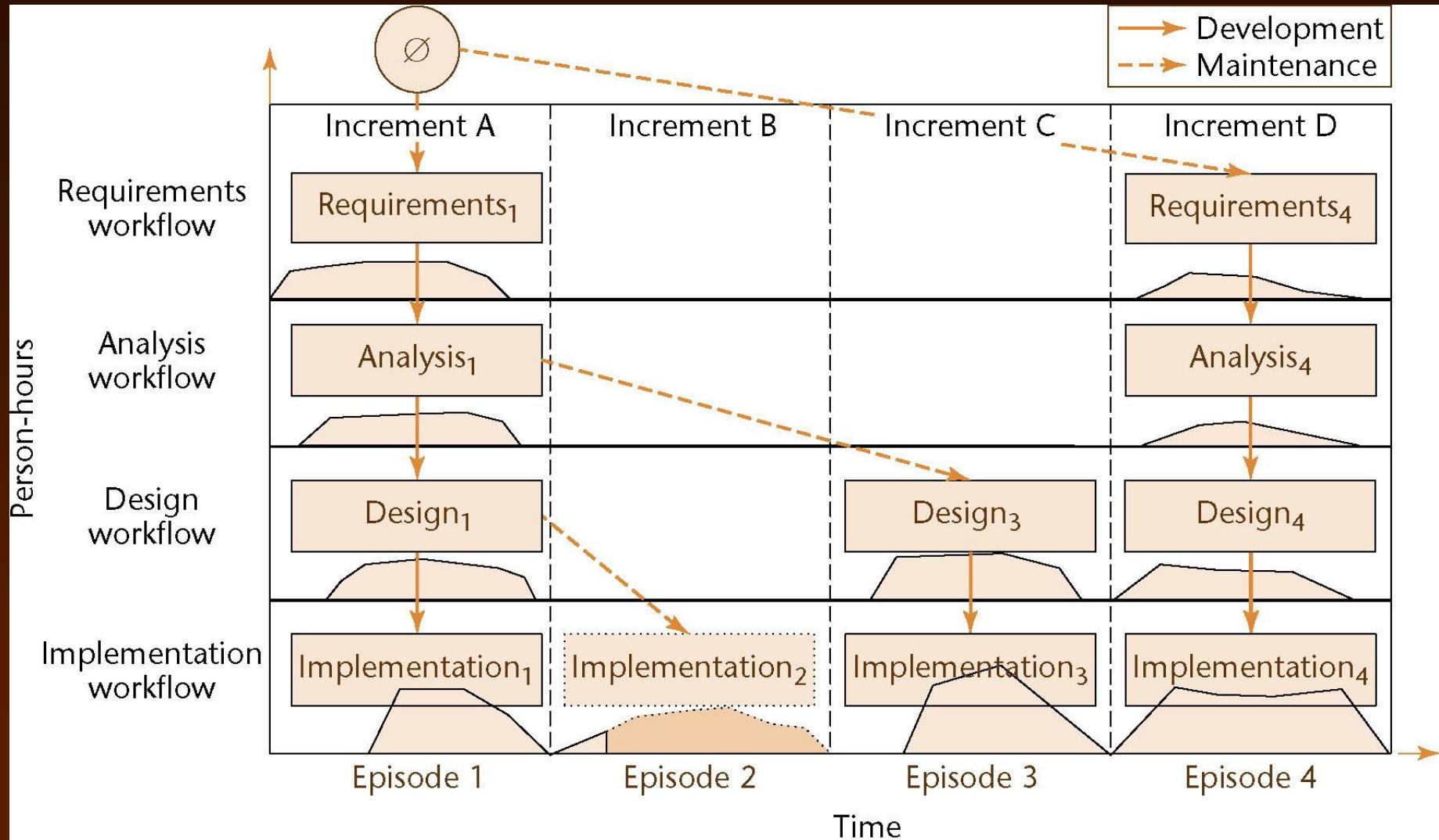


Figure 3.1

The Phases of the Unified Process (contd)

Slide 3.146

- The four **increments** are labeled
 - ▶ Inception phase
 - ▶ Elaboration phase
 - ▶ Construction phase
 - ▶ Transition phase
- The phases of the Unified Process are the increments

The Phases of the Unified Process (contd)

Slide 3.147

- In theory, there could be any number of increments
 - ▶ In practice, development seems to consist of four increments. Hence, four phases are suggested.

Classical Phases versus Workflows

Slide 3.148

- Sequential phases do not exist in the real world
- Instead, the **five core workflows (activities)** are performed over the entire life cycle
 - ▶ Requirements workflow
 - ▶ Analysis workflow
 - ▶ Design workflow
 - ▶ Implementation workflow
 - ▶ Test workflow

Iteration and Incrementation

Slide 3.149

- Iteration is performed during each incrementation

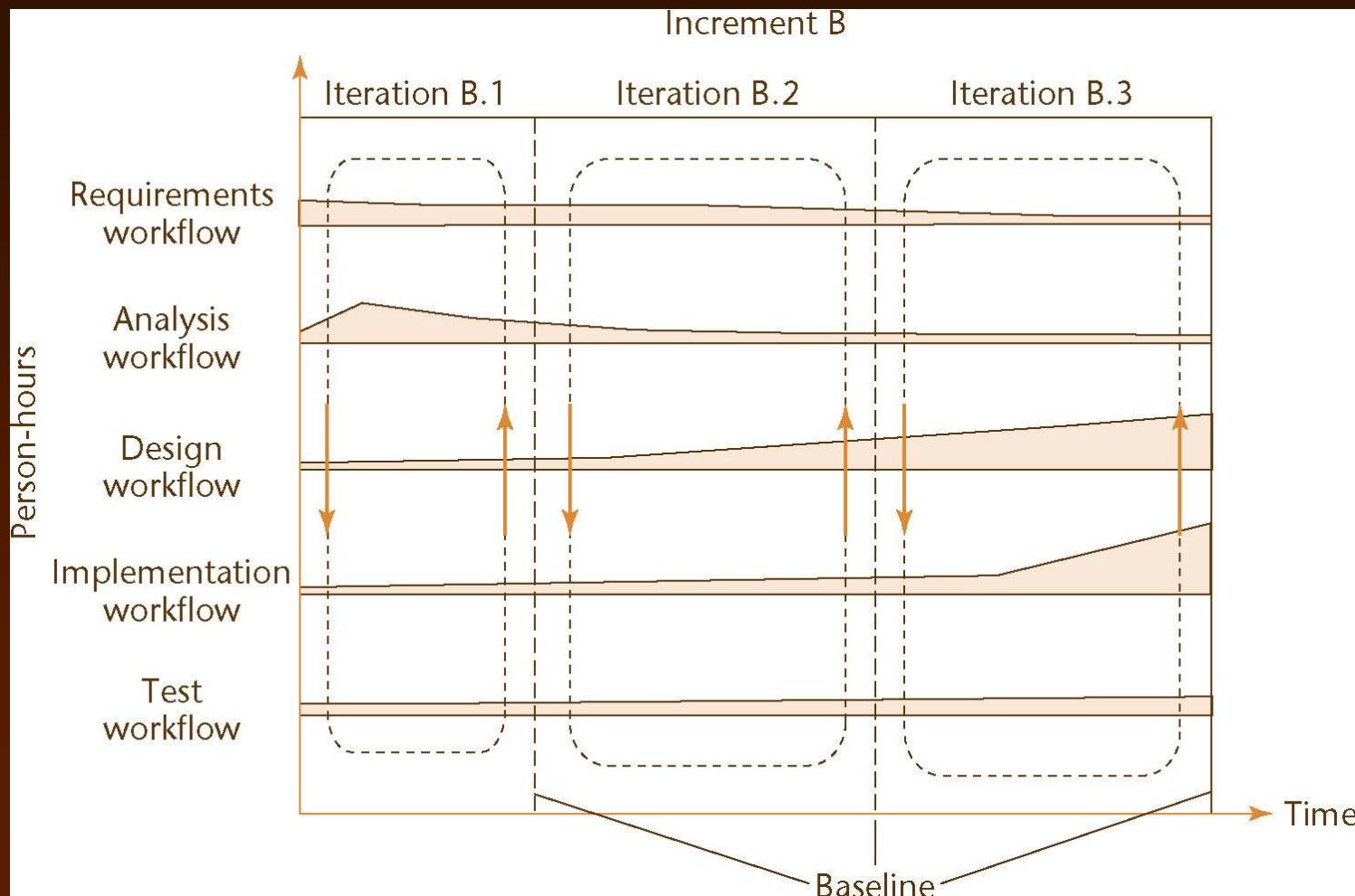


Figure 2.5

Other Aspects of Iteration and Incrementation

Slide 3.150

- We can consider the project as a whole as a set of mini projects (increments)
- Each mini project extends the
 - ▶ Requirements artifacts
 - ▶ Analysis artifacts
 - ▶ Design artifacts
 - ▶ Implementation artifacts
 - ▶ Testing artifacts
- The final set of artifacts is the complete product

Other Aspects of Iteration and Incrementation

Slide 3.151

- During each mini project, we
 - ▶ Extend the artifacts (incrementation);
 - ▶ Check the artifacts (test workflow); and
 - ▶ If necessary, change the relevant artifacts (iteration)

The Phases of the Unified Process

Slide 3.152

- Every step performed in the Unified Process falls into
 - ▶ One of the *five core workflows*, and *also*
 - ▶ One of the *four phases*
- Why does each step have to be considered twice?
- **Workflow**
 - ▶ Technical context of a step
- **Phase**
 - ▶ Business / Economical context of a step

3.10.1 The Inception Phase

Slide 3.153

- The aim of the **inception phase** is to determine whether the proposed software product is economically viable

- 1. Gain an understanding of the domain
- 2. Build the business model
- 3. Delimit the scope of the proposed project
 - ▶ Focus on the subset of the business model that is covered by the proposed software product
- 4. Begin to make the initial business case

The Inception Phase : The Initial Business Case

Slide 3.155

- Questions that need to be answered include:
 - ▶ Is the proposed software product cost effective?
 - ▶ How long will it take to obtain a return on investment?
 - ▶ Alternatively, what will be the cost if the company decides not to develop the proposed software product?
 - ▶ If the software product is to be sold in the marketplace, have the necessary marketing studies been performed?
 - ▶ Can the proposed software product be delivered in time?
 - ▶ If the software product is to be developed to support the client organization's own activities, what will be the impact if the proposed software product is delivered late?

The Inception Phase: The Initial Business Case

Slide 3.156

- What are the risks involved in developing the software product
- How can these risks be mitigated?
 - ▶ Does the team (who will develop the proposed software product) have the necessary experience?
 - ▶ Is new hardware needed for this software product?
 - ▶ If so, is there a risk that it will not be delivered in time?
 - ▶ If so, is there a way to mitigate that risk, perhaps by ordering back-up hardware from another supplier?
 - ▶ Are software tools (CASE Tools) needed?
 - ▶ Are they currently available?
 - ▶ Do they have all the necessary functionality?

The Inception Phase: The Initial Business Case

Slide 3.157

- Answers are needed by the end of the **inception phase** so that the **initial business case** can be made

The Inception Phase: Risks

Slide 3.158

- There are three major risk categories:
 - ▶ Technical risks
 - See earlier slide
 - ▶ The risk of not getting the requirements right
 - Mitigated by performing the requirements workflow correctly
 - ▶ The risk of not getting the architecture right
 - The architecture may not be sufficiently robust

The Inception Phase: Risks

Slide 3.159

- To mitigate all three classes of risks
 - ▶ The risks need to be ranked so that the critical risks are mitigated first
- This concludes the steps of the inception phase that fall under the requirements workflow

The Inception Phase: Analysis, Design Workflows

Slide 3.160

- A small amount of the analysis workflow may be performed during the inception phase
 - ▶ Information needed for the design of the architecture is extracted
- Accordingly, a small amount of the **design workflow** may be performed, too, in inception phase

The Inception Phase: Implementation Workflow

Slide 3.161

- Coding is generally not performed during the inception phase
- However, a *proof-of-concept prototype* is sometimes build to test the feasibility of constructing a part of the software product

The Inception Phase: Test Workflow

Slide 3.162

- The test workflow commences almost at the start of the inception phase
 - ▶ The aim is to ensure that the requirements have been accurately determined

The Inception Phase: Planning

Slide 3.163

- There is insufficient information at the beginning of the inception phase to plan the entire development
 - ▶ The only planning that is done at the start of the project is the planning for the inception phase itself
- For the same reason, the only planning that can be done at the end of the inception phase is the plan for just the next phase, the **elaboration phase**

- The **deliverables** of the inception phase include:
 - ▶ The initial version of the domain model
 - ▶ The initial version of the business model
 - ▶ The initial version of the requirements artifacts
 - ▶ A preliminary version of the analysis artifacts
 - ▶ A preliminary version of the architecture
 - ▶ The initial list of risks
 - ▶ The initial ordering of the use cases
 - ▶ The plan for the elaboration phase
 - ▶ The initial version of the business case

The Inception Phase: The Initial Business Case

Slide 3.165

- Obtaining the initial version of the business case is the overall aim of the inception phase
- This initial version incorporates
 - ▶ A description of the scope of the software product
 - ▶ Financial details
 - ▶ If the proposed software product is to be marketed, the business case will also include
 - **Revenue projections, market estimates, initial cost estimates**
 - ▶ If the software product is to be used in-house, the business case will include
 - **The initial cost–benefit analysis**

3.10.2 Elaboration Phase

Slide 3.166

- The aim of the **elaboration phase** is to refine the initial requirements
 - ▶ Refine the architecture
 - ▶ Monitor the risks and refine their priorities
 - ▶ Refine the business case
 - ▶ Produce the project management plan
- The major activities of the elaboration phase are refinements or elaborations of the previous phase

The Tasks of the Elaboration Phase

Slide 3.167

- The tasks of the **elaboration phase** correspond to:
 - ▶ All but completing the requirements workflow
 - ▶ Performing virtually the entire analysis workflow
 - ▶ Starting the design of the architecture

The Elaboration Phase: Documentation

Slide 3.168

- The **deliverables** of the elaboration phase include:
 - ▶ The completed domain model
 - ▶ The completed business model
 - ▶ The completed requirements artifacts
 - ▶ The completed analysis artifacts
 - ▶ An updated version of the architecture
 - ▶ An updated list of risks
 - ▶ The project management plan (for the rest of the project)
 - ▶ The completed business case

3.10.3 Construction Phase

Slide 3.169

- The aim of the **construction phase** is to produce the first operational-quality version of the software product
 - ▶ This is sometimes called the beta release.

The Tasks of the Construction Phase

Slide 3.170

- The emphasis in this phase is on
 - ▶ Implementation and
 - ▶ Testing
 - Unit testing of modules
 - Integration testing of subsystems
 - Product testing of the overall system

The Construction Phase: Documentation

Slide 3.171

- The deliverables of the construction phase include:
 - ▶ The initial user manual and other manuals, as appropriate
 - ▶ All the artifacts (beta release versions)
 - ▶ The completed architecture
 - ▶ The updated risk list
 - ▶ The project management plan (for the remainder of the project)
 - ▶ If necessary, the updated business case

3.10.4 The Transition Phase

Slide 3.172

- The aim of the **transition phase** is to ensure that the client's requirements have indeed been met
 - ▶ Faults in the software product are corrected
 - ▶ All the manuals are completed
 - ▶ Attempts are made to discover any previously unidentified risks
- This phase is driven by feedback from the site(s) at which the beta release has been installed

The Transition Phase: Documentation

Slide 3.173

- The deliverables of the transition phase include:
 - ▶ All the artifacts (final versions)
 - ▶ The completed manuals

The Phases of the Unified Process

Slide 3.174

- The *increments* are identified as *phases*

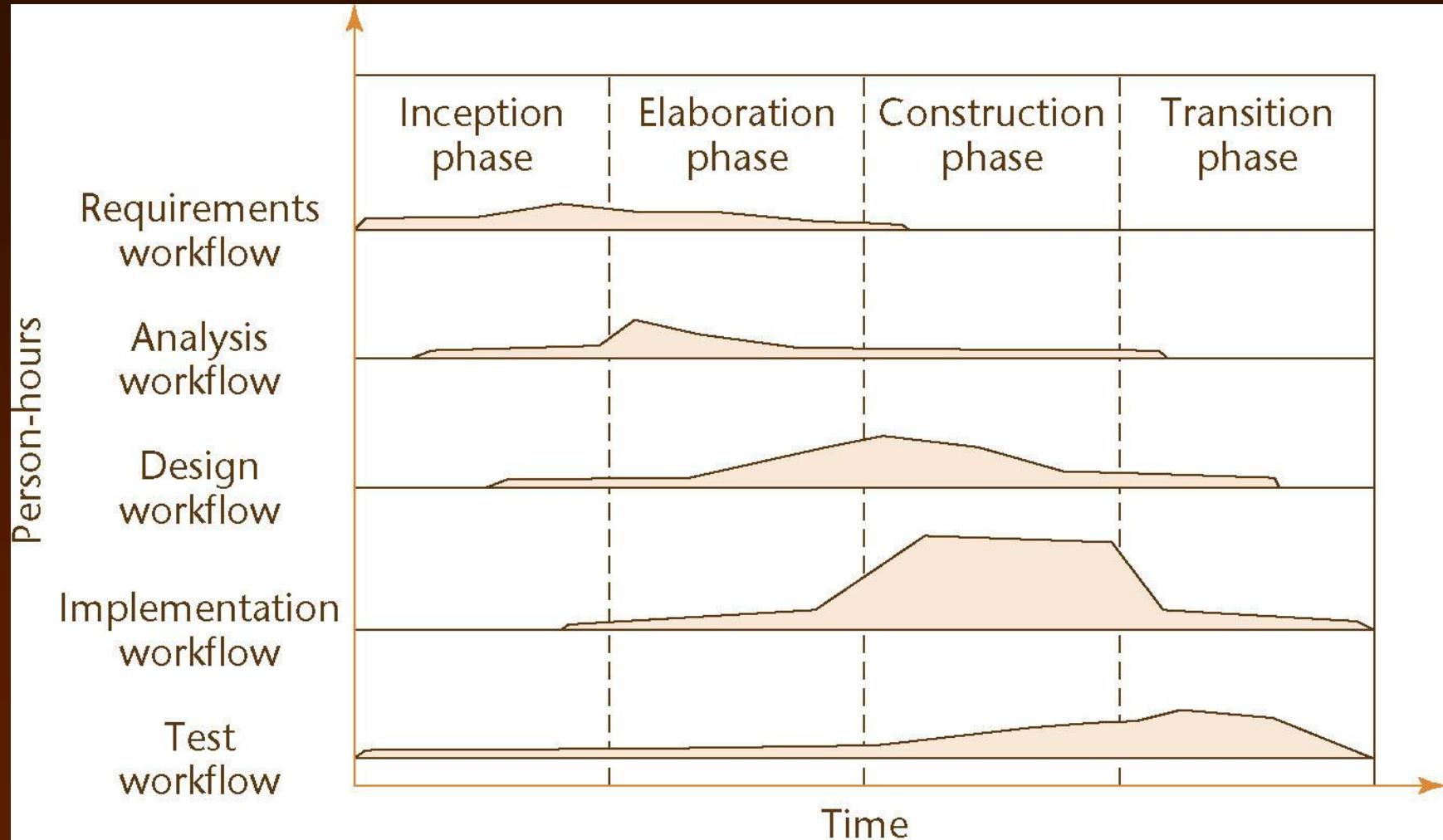


Figure 3.1

3.11 One- and Two-Dimensional Life-Cycle Models

Slide 3.175

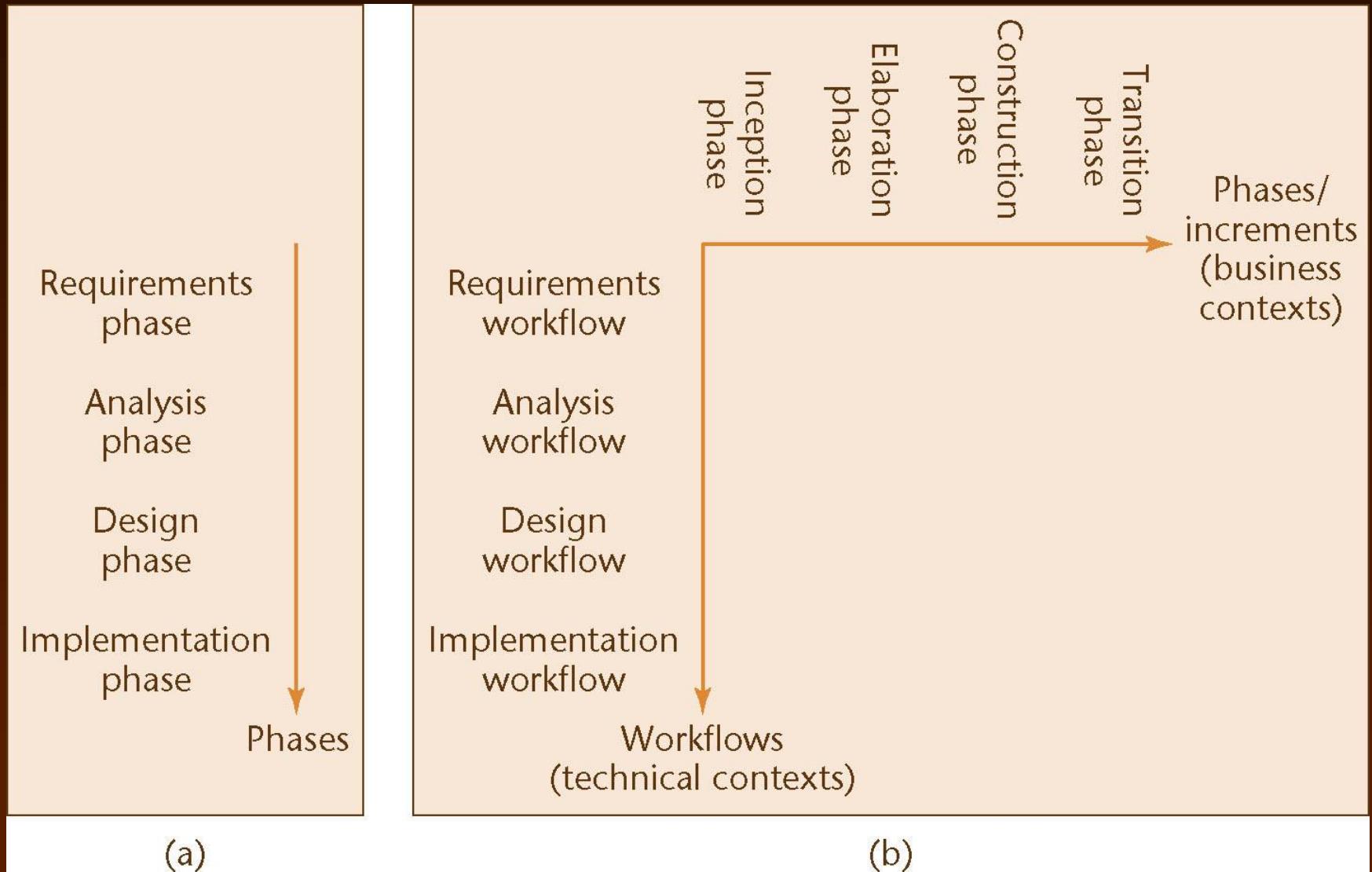


Figure 3.2

Why a Two-Dimensional Model?

Slide 3.176

- A traditional life cycle is a one-dimensional model
 - ▶ Represented by the single axis on the previous slide
 - Example: Waterfall model
- The Unified Process is a two-dimensional model
 - ▶ Represented by the two axes on the previous slide
- The two-dimensional figure shows
 - ▶ The workflows (**technical contexts**) and
 - ▶ The phases (**business contexts**)

Why a Two-Dimensional Model? (contd)

Slide 3.177

- The Waterfall model
- One-dimensional

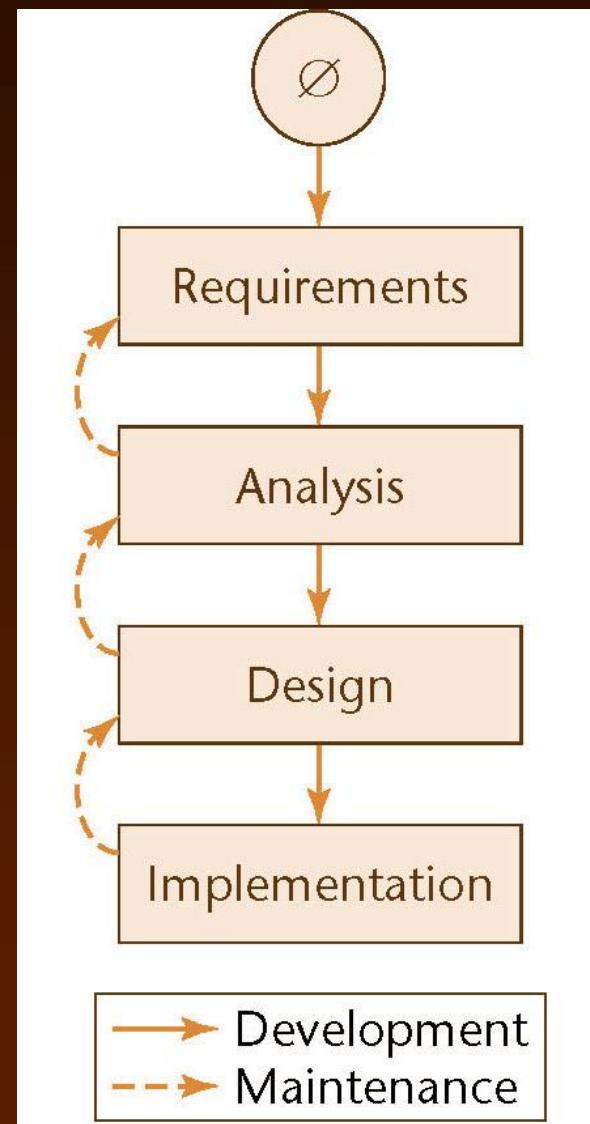


Figure 3.3

Why a Two-Dimensional Model? (contd)

Slide 3.178

- Evolution tree model
- Two-dimensional

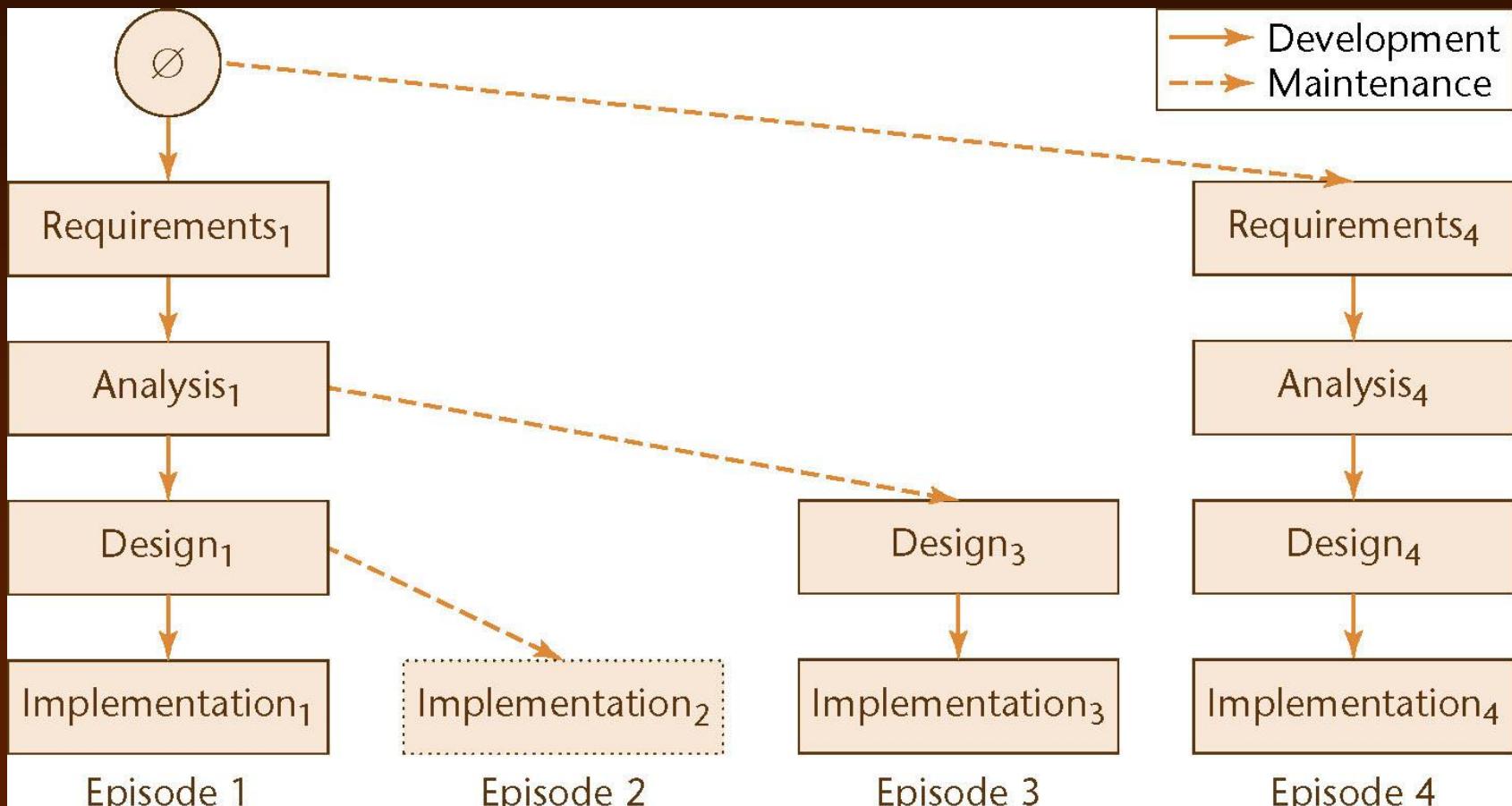


Figure 3.4

Why a Two-Dimensional Model? (contd)

Slide 3.179

- Are all the additional complications of the two-dimensional model necessary?
- In an ideal world, each workflow would be completed before the next workflow is started

Why a Two-Dimensional Model? (contd)

Slide 3.180

- In reality, the development task is too big for this
- As a consequence of Miller's Law
 - ▶ The development task has to be divided into increments (phases)
 - ▶ Within each increment, iteration is performed until the task is complete.

Why a Two-Dimensional Model? (contd)

Slide 3.181

- At the beginning of the process, there is not enough information about the software product to carry out the requirements workflow
 - ▶ Similarly for the other core workflows
- A software product has to be broken into subsystems
- Even subsystems can be too large at times
 - ▶ Components may be all that can be handled until a fuller understanding of all the parts of the product as a whole has been obtained

Why a Two-Dimensional Model? (contd)

Slide 3.182

- The Unified Process handles the inevitable changes well
 - ▶ The moving target problem
 - ▶ The inevitable mistakes
- The Unified Process is the best solution found to date for treating a large problem as a set of smaller, largely independent subproblems
 - ▶ It provides a framework for incrementation and iteration
 - ▶ In the future, it will inevitably be superseded by some better methodology.

Unified Process

Slide 3.183

Thank you.



Software Quality Management

Lecture # 3: The Unified Process

The UNIFIED (SOFTWARE DEVELOPMENT) PROCESS

- The Unified Process
- Iteration and incrementation within the object-oriented paradigm
- The Requirements workflow
- The Analysis workflow
- The Design workflow
- The Implementation workflow
- The Test workflow

- Postdelivery maintenance
- Retirement
- The phases of the Unified Process
- One- versus two-dimensional life-cycle models
- Why two-dimensional process?

3.1 The Unified Process

Slide 3.188

- Until late 1990s, three of the most successful object-oriented methodologies were
 - ▶ Booch's method
 - ▶ Jacobson's Objectory
 - ▶ Rumbaugh's OMT

- In 1999, Booch, Jacobson, and Rumbaugh published a complete object-oriented analysis and design methodology that **unified** their three separate methodologies
 - ▶ Original name: *Rational Unified Process* (RUP)
 - ▶ Next name: *Unified Software Development Process*
 - ▶ Name used today: **Unified Process** (for brevity)

- The Unified Process ***is not a series of steps*** for constructing a software product
 - ▶ No such single “one size fits all” methodology could exist
 - ▶ There is a wide variety of different types of software
- The Unified Process is an adaptable methodology
 - ▶ It has to be modified for the specific software product to be developed

- Unified Modelling Language (UML) is graphical
 - ▶ A picture is worth a thousand words
- UML diagrams enable software engineers to communicate quickly and accurately

- The Unified Process is a **modeling** technique
 - ▶ A **model** is a set of UML diagrams that represent various aspects of the software product we want to develop
- UML stands for **Unified Modeling** Language
 - ▶ UML is the tool that we use to **represent** (model) the target software product

- The object-oriented paradigm is **iterative** and **incremental** in nature
 - ▶ There is no alternative to repeated **iteration** and **incrementation** until the UML diagrams are satisfactory.

Iteration and Incrementation

Slide 3.194

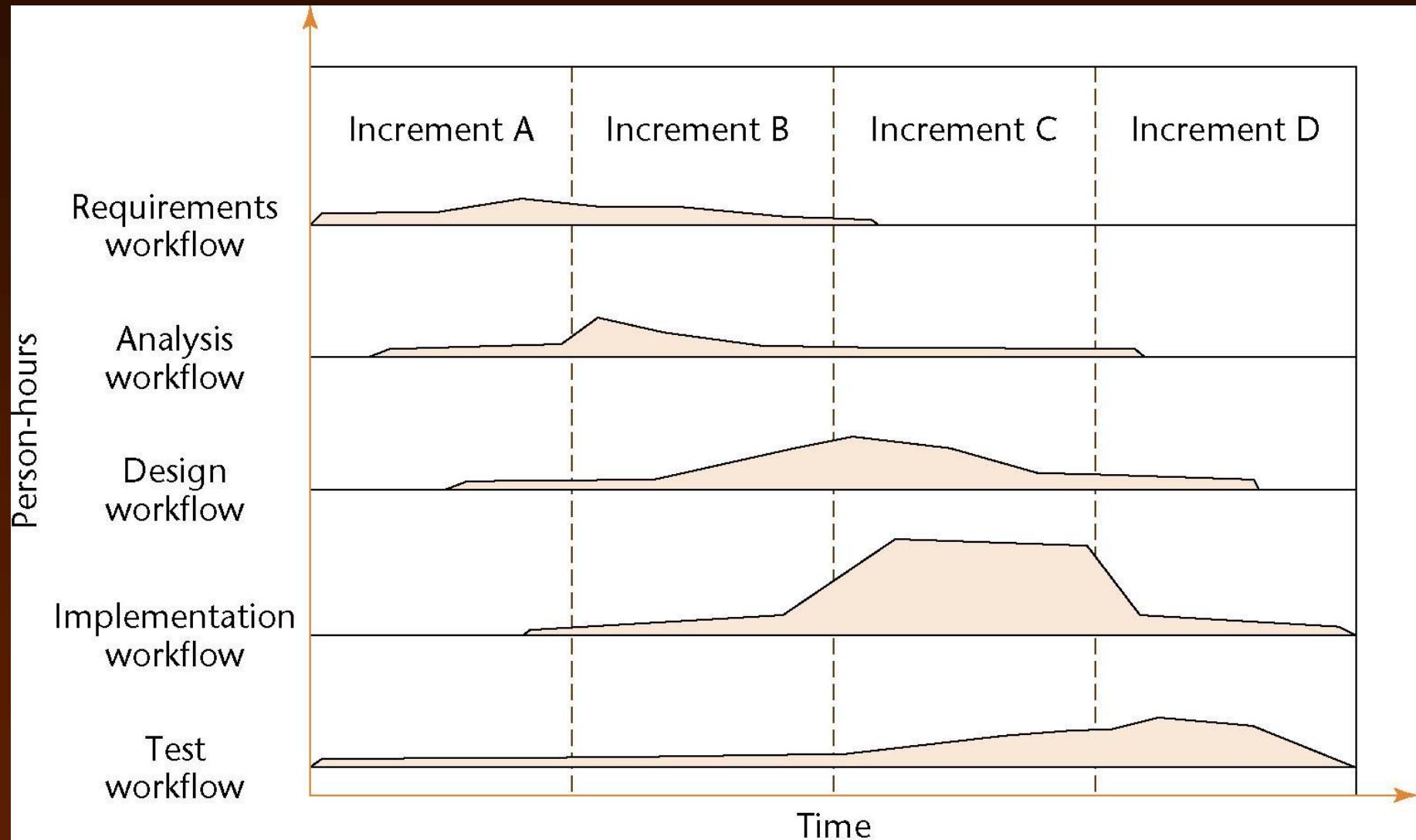


Figure 2.4

3.10 The Phases of the Unified Process

Slide 3.195

- The *increments* are identified as *phases*

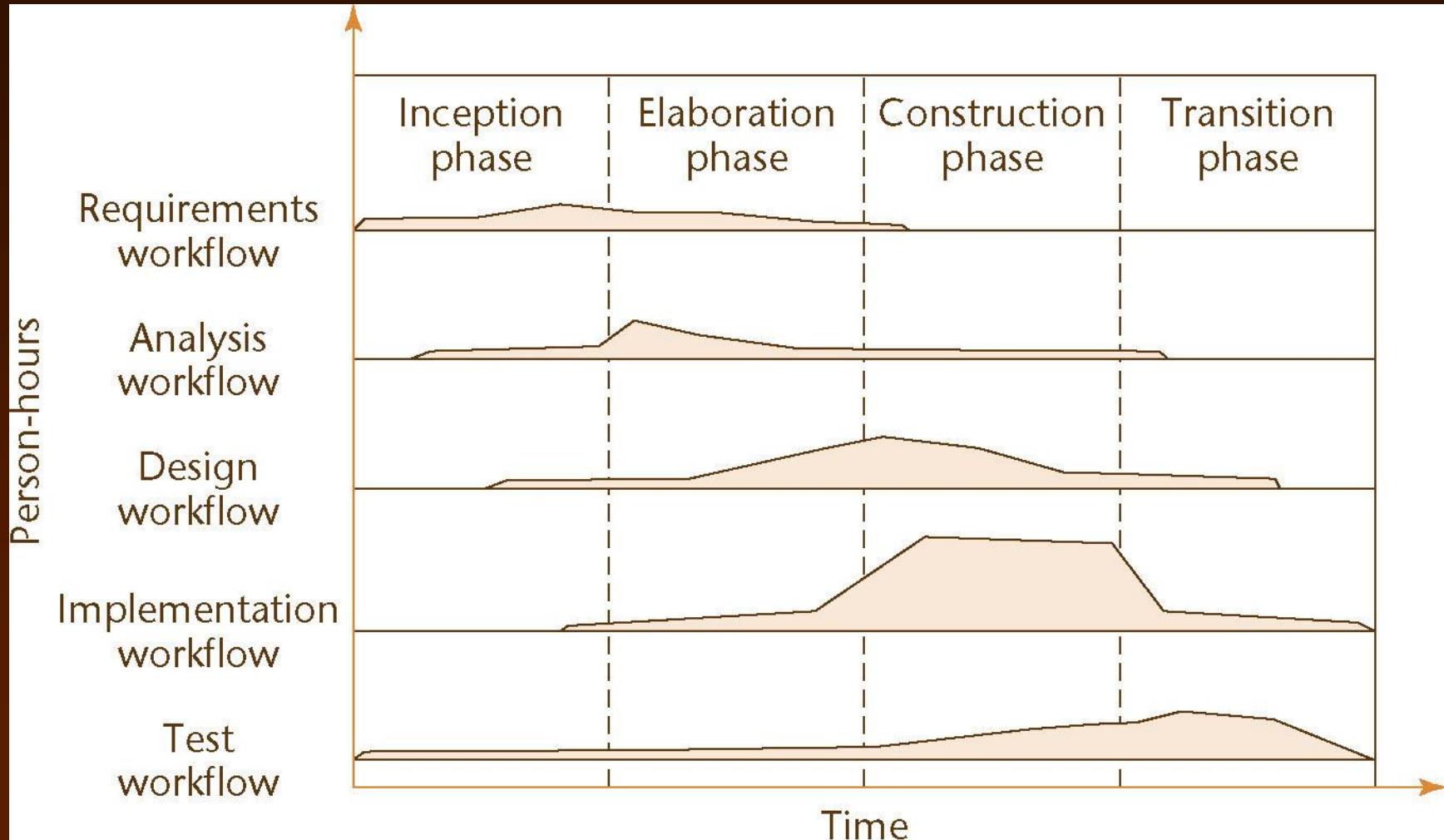


Figure 3.1

Iteration and Incrementation

Slide 3.196

- Iteration is performed during each incrementation

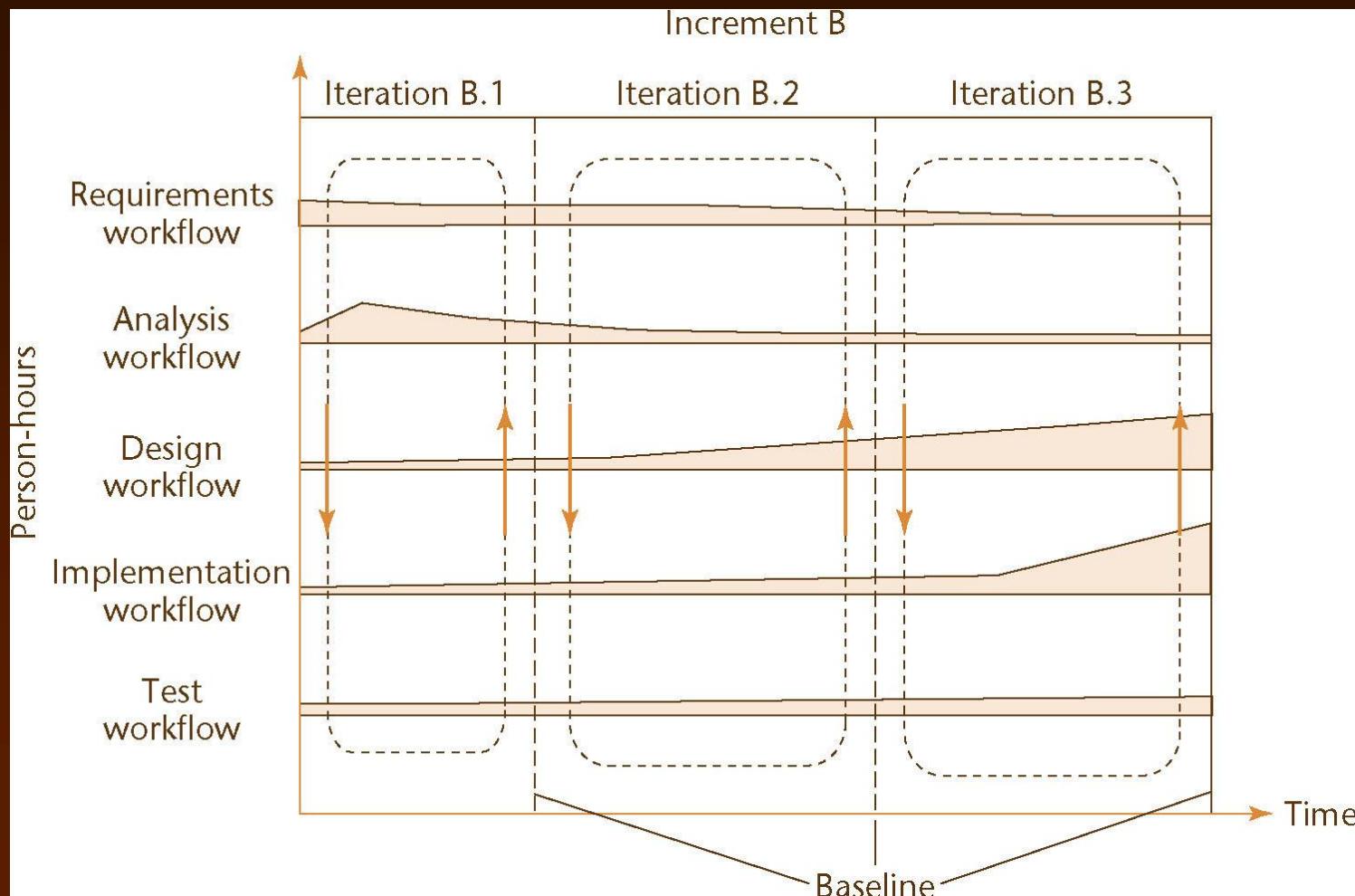


Figure 2.5

Iteration and Incrementation

Slide 3.197

- Iteration and incrementation are used in conjunction with one another
 - ▶ There is no single “requirements phase” or “design phase”
 - ▶ Instead, there are multiple instances of each phase

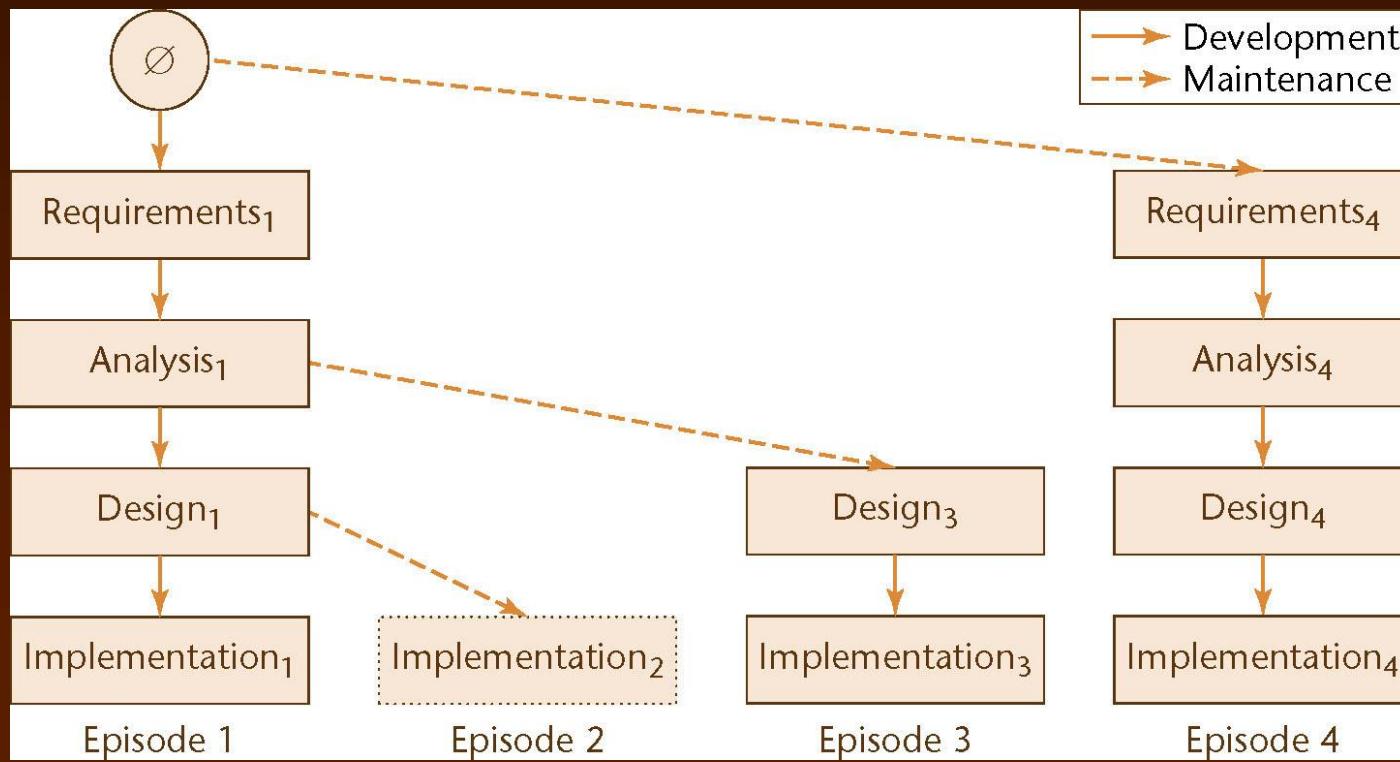


Figure 2.6

Iteration and Incrementation

Slide 3.198

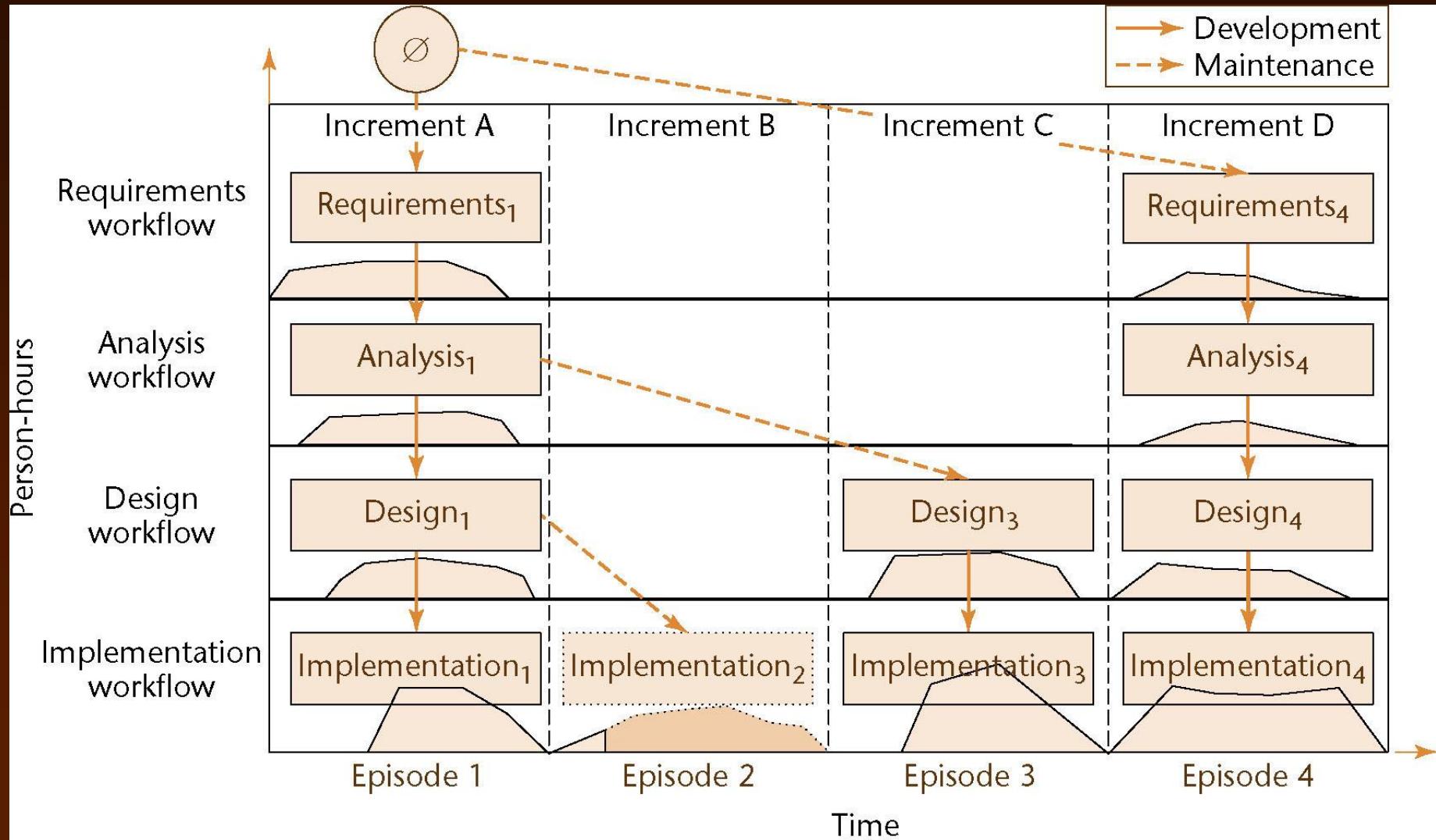


Figure 3.1

More on Incrementation

Slide 3.199

- Each episode corresponds to an **increment**
- Not every increment includes every workflow
- Increment B was not completed
- Dashed lines denote **maintenance**
 - ▶ Episodes 2, 3: Corrective maintenance
 - ▶ Episode 4: Perfective maintenance

Other Aspects of Iteration and Incrementation

Slide 3.200

- We can consider the project as a whole as a set of mini projects (increments)
- Each mini project extends the
 - ▶ Requirements artifacts
 - ▶ Analysis artifacts
 - ▶ Design artifacts
 - ▶ Implementation artifacts
 - ▶ Testing artifacts
- The final set of artifacts is the complete product

Other Aspects of Iteration and Incrementation

Slide 3.201

- During each mini project, we
 - ▶ Extend the artifacts (**incrementation**);
 - ▶ Check the artifacts (test workflow); and
 - ▶ If necessary, change the relevant artifacts (**iteration**);

- Each iteration can be viewed as a small but complete **waterfall life-cycle model**
- During each iteration we select a portion of the software product
- On that portion we perform the
 - ▶ Classical requirements phase
 - ▶ Classical analysis phase
 - ▶ Classical design phase
 - ▶ Classical implementation phase

- The Unified Process
- Iteration and incrementation within the object-oriented paradigm
- The Requirements workflow
- The Analysis workflow
- The Design workflow
- The Implementation workflow
- The Test workflow

3.3 The Requirements Workflow

Slide 3.204

- The aim of the Requirements Workflow
 - ▶ To determine the client's needs

Overview of the Requirements Workflow

Slide 3.205

- First, gain an understanding of the *application domain* (or *domain*, for short)
 - ▶ That is, the specific business environment in which the software product is to operate
- Second, build a **business model**
 - ▶ Use UML to describe the client's business processes
 - ▶ If at any time the client does not feel that the cost is justified, development terminates immediately

Overview of the Requirements Workflow (contd)

Slide 3.206

- It is vital to determine the client's constraints
 - ▶ Deadline
 - **Nowadays, software products are often mission critical**
 - ▶ Parallel running
 - ▶ Portability
 - ▶ Reliability
 - ▶ Rapid response time
 - ▶ Cost
 - **The client will rarely inform the developer how much money is available**
 - **A bidding procedure is used instead**

Overview of the Requirements Workflow (contd)

Slide 3.207

- The aim of this *concept exploration* is to determine
 - ▶ What the client needs
 - ▶ *Not* what the client wants

3.4 The Analysis Workflow

Slide 3.208

- The aim of the analysis workflow
 - ▶ To analyze and refine the requirements
- Why not do this during the requirements workflow?
 - ▶ The requirements artifacts must be totally comprehensible by the client
- The artifacts of the requirements workflow must therefore be expressed in a natural (human) language
 - ▶ All natural languages are imprecise

The Analysis Workflow (contd)

Slide 3.209

- Example from a manufacturing information system:
 - ▶ “A part record and a plant record are read from the database. If **it** contains the letter A directly followed by the letter Q, then calculate the cost of transporting that part to that plant”
- To what does **it** refer?
 - ▶ The part record?
 - ▶ The plant record?
 - ▶ Or the database?

The Analysis Workflow (contd)

Slide 3.210

- Two separate workflows are needed
 - ▶ The **requirements artifacts** must be expressed in the language of the client
 - ▶ The **analysis artifacts** must be precise, and complete enough for the designers

The Specification Document (contd)

Slide 3.211

- Specification document (“specifications”)
 - ▶ It constitutes a contract
 - ▶ It must not have imprecise phrases like “optimal,” or “98% complete”
- Having complete and correct specifications is essential for
 - ▶ Testing and
 - ▶ Maintenance

The Specification Document (continued)

Slide 3.212

- The specification document must not have
 - ▶ Ambiguities
 - ▶ Contradictions
 - ▶ Omissions
 - ▶ Incompleteness
 - ▶ Errors

Software Project Management Plan

Slide 3.213

- Once the client has signed off the specifications, detailed planning and estimating begins
- We draw up the software project management plan, including
 - Cost estimate
 - Duration estimate
 - Deliverables
 - Milestones
 - Budget
- This is the earliest possible time for the SPMP

3.5 The Design Workflow

Slide 3.214

- The aim of the Design Workflow is to refine the analysis workflow until the material is in a form that can be implemented by the programmers
 - ▶ Many **nonfunctional requirements** need to be finalized at this time, including
 - Choice of programming language
 - Reuse issues
 - Portability issues

- Architectural design
 - ▶ Decompose the product into modules
- Detailed design
 - ▶ Design each module:
 - Data structures
 - Algorithms

- Classes are extracted during the object-oriented analysis workflow and
 - ▶ Designed during the **design workflow**
- Accordingly
 - ▶ Classical architectural design corresponds to part of the object-oriented analysis workflow
 - ▶ Classical detailed design corresponds to part of the object-oriented design workflow

The Design Workflow (contd)

Slide 3.217

- Retain design decisions
 - ▶ For when a dead-end is reached
 - ▶ To prevent the maintenance team reinventing the wheel

3.6 The Implementation Workflow

Slide 3.218

- The aim of the Implementation Workflow is to implement the target software product in the selected implementation language
 - ▶ A large software product is partitioned into subsystems
 - ▶ The subsystems consist of *components* or *code artifacts*

3.7 The Test Workflow

Slide 3.219

- The **Test Workflow** is the responsibility of
 - ▶ *Every* developer and maintainer, and
 - ▶ The Software Quality Assurance group
- Traceability of artifacts is an important requirement for successful testing

3.7.1 Requirements Artifacts

Slide 3.220

- Every item in the analysis artifacts must be traceable to an item in the requirements artifacts
 - ▶ Similarly for the design and implementation artifacts

3.7.2 Analysis Artifacts

Slide 3.221

- The analysis artifacts should be checked by means of a review
 - ▶ Representatives of the client and analysis team must be present
- The Software Project Management Plan (SPMP) must be similarly checked
 - ▶ Pay special attention to the cost and duration estimates

3.7.3 Design Artifacts

Slide 3.222

- Design reviews are essential
 - ▶ A client representative is not usually present

3.7.4 Implementation Artifacts

Slide 3.223

- Each component is tested as soon as it has been implemented
 - ▶ *Unit testing*
- At the end of each iteration, the completed components are combined and tested
 - ▶ *Integration testing*
- When the product appears to be complete, it is tested as a whole
 - ▶ *Product testing*
- Once the completed product has been installed on the client's computer, the client tests it
 - ▶ *Acceptance testing*

- COTS software is released for testing by prospective clients
 - ▶ Alpha release
 - ▶ Beta release
- There are advantages and disadvantages to being an alpha or beta release site

3.8 Postdelivery Maintenance

Slide 3.225

- Postdelivery maintenance is an essential component of software development
 - ▶ More money is spent on postdelivery maintenance than on all other activities combined
- Problems can be caused by
 - ▶ Lack of documentation of all kinds

Postdelivery Maintenance (contd)

Slide 3.226

- Two types of testing are needed
 - ▶ Testing the changes made during postdelivery maintenance
 - ▶ Regression testing
- All previous test cases (and their expected outcomes) need to be retained

3.9 Retirement

Slide 3.227

- Software can be unmaintainable because
 - ▶ A drastic change in design has occurred
 - ▶ The product must be implemented on a totally new hardware/operating system
 - ▶ Documentation is missing or inaccurate
 - ▶ Hardware is to be changed — it may be cheaper to rewrite the software from scratch than to modify it
- These are instances of maintenance (rewriting of existing software)

- True retirement is a rare event
- It occurs when the client organization no longer needs the functionality provided by the product

The Phases of the Unified Process (contd)

Slide 3.229

- The phases of the Unified Process are the increments
- The ***four increments*** are labeled
 - ▶ Inception phase
 - ▶ Elaboration phase
 - ▶ Construction phase
 - ▶ Transition phase

The Phases of the Unified Process (contd)

Slide 3.230

- In theory, there could be any number of increments
 - ▶ In practice, development seems to consist of four increments. Hence, four phases are suggested.

Classical Phases versus Workflows

Slide 3.231

- Sequential phases do not exist in the real world
- Instead, the **five core workflows (activities)** are performed over the entire life cycle
 - ▶ Requirements workflow
 - ▶ Analysis workflow
 - ▶ Design workflow
 - ▶ Implementation workflow
 - ▶ Test workflow

The Phases of the Unified Process

Slide 3.232

- The *increments* are identified as *phases*

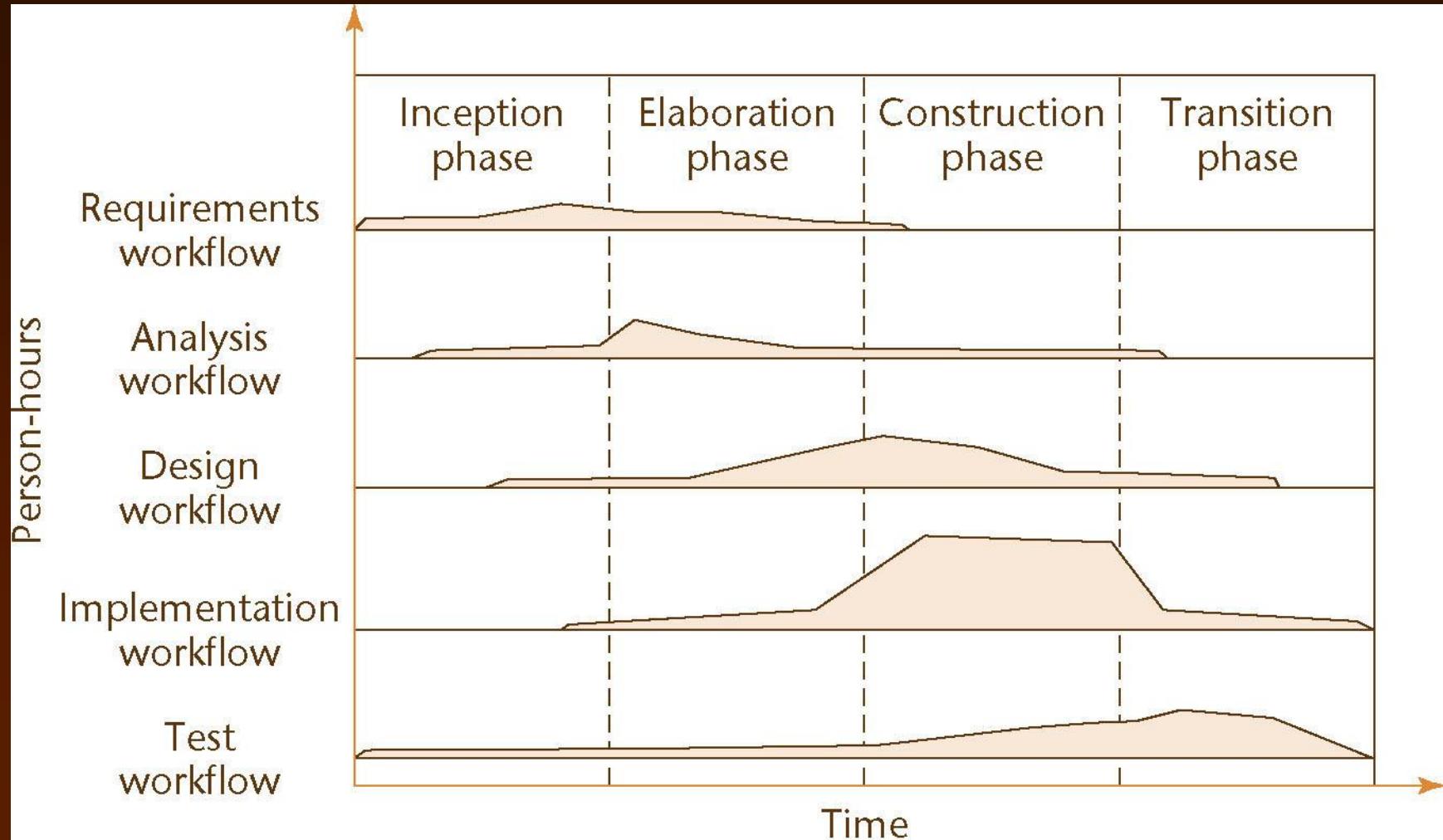


Figure 3.1

Iteration and Incrementation

Slide 3.233

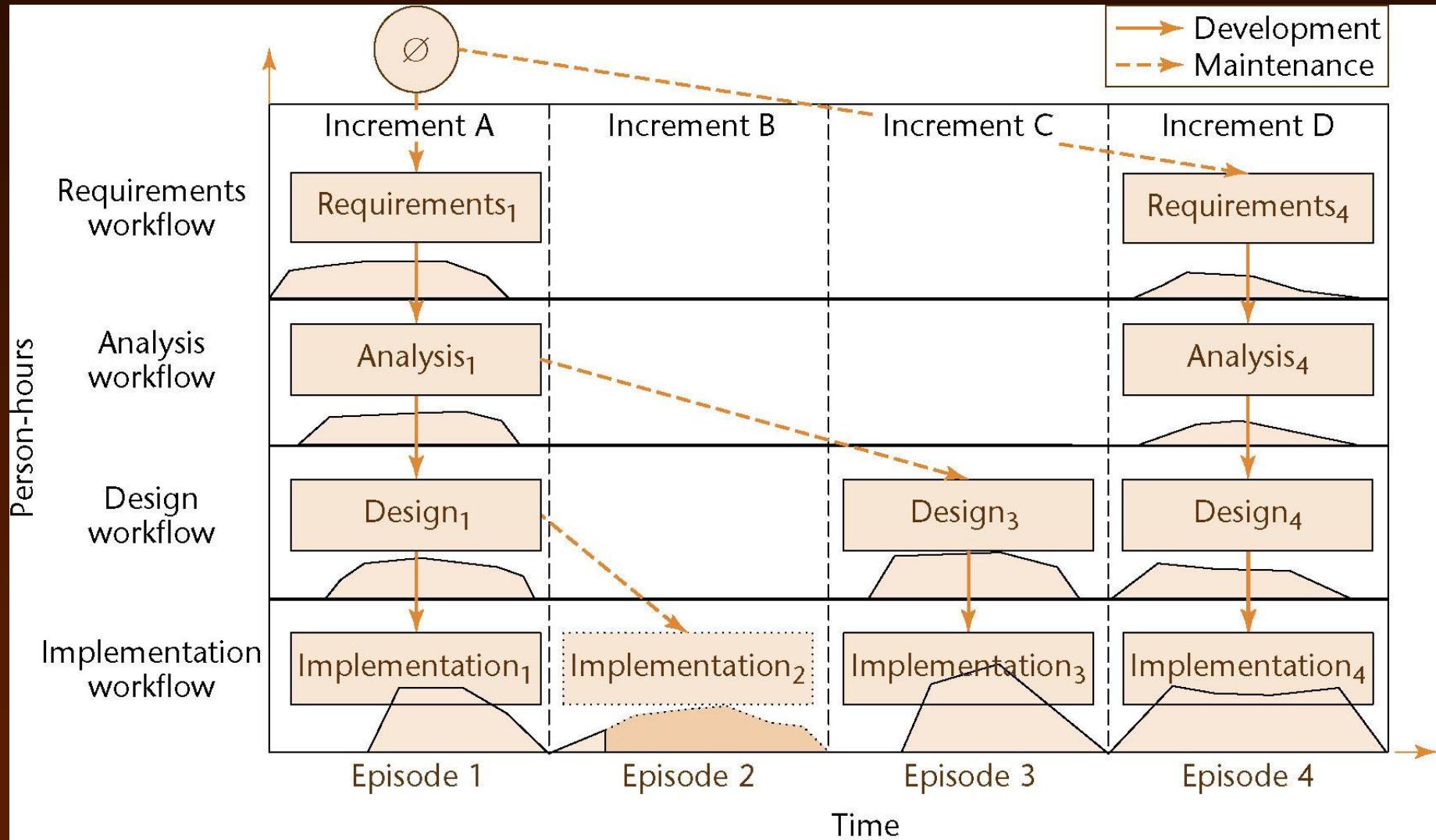


Figure 3.1

The Phases of the Unified Process (contd)

Slide 3.234

- The four ***increments*** are labeled
 - ▶ Inception phase
 - ▶ Elaboration phase
 - ▶ Construction phase
 - ▶ Transition phase
- The **phases** of the Unified Process **are the increments**

The Phases of the Unified Process (contd)

Slide 3.235

- In theory, there could be any number of increments
 - ▶ In practice, development seems to consist of four increments. Hence, four phases are suggested.

Classical Phases versus Workflows

Slide 3.236

- Sequential phases do not exist in the real world
- Instead, the **five core workflows (activities)** are performed over the entire life cycle
 - ▶ Requirements workflow
 - ▶ Analysis workflow
 - ▶ Design workflow
 - ▶ Implementation workflow
 - ▶ Test workflow

Iteration and Incrementation

Slide 3.237

- Iteration is performed during each incrementation

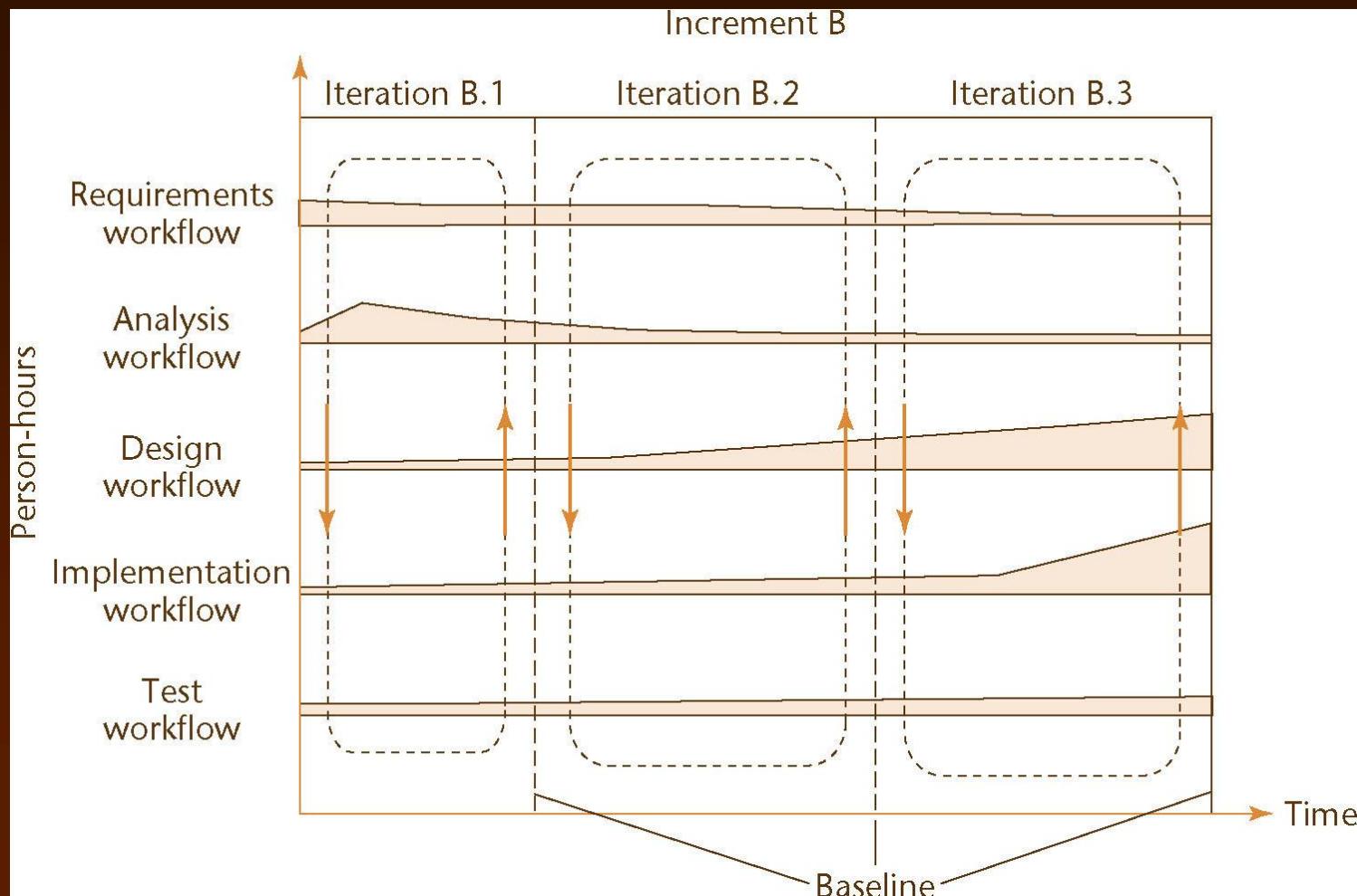


Figure 2.5

Other Aspects of Iteration and Incrementation

Slide 3.238

- We can consider the project as a whole as a set of mini projects (increments)
- Each mini project extends the
 - ▶ Requirements artifacts
 - ▶ Analysis artifacts
 - ▶ Design artifacts
 - ▶ Implementation artifacts
 - ▶ Testing artifacts
- The final set of artifacts is the complete product

Other Aspects of Iteration and Incrementation

Slide 3.239

- During each mini project, we
 - ▶ Extend the artifacts (incrementation);
 - ▶ Check the artifacts (test workflow); and
 - ▶ If necessary, change the relevant artifacts (iteration)

The Phases of the Unified Process

Slide 3.240

- Every step performed in the Unified Process falls into
 - ▶ One of the *five core workflows*, and *also*
 - ▶ One of the *four phases*
- Why does each step have to be considered twice?
- **Workflow**
 - ▶ Technical context of a step
- **Phase**
 - ▶ Business / Economical context of a step

3.10.1 The Inception Phase

Slide 3.241

- The aim of the **inception phase** is to determine whether the proposed software product is economically viable

The Inception Phase (contd)

Slide 3.242

- 1. Gain an understanding of the domain
- 2. Build the business model
- 3. Delimit the scope of the proposed project
 - ▶ Focus on the subset of the business model that is covered by the proposed software product
- 4. Begin to make the initial business case

The Inception Phase : The Initial Business Case

Slide 3.243

- Questions that need to be answered include:
 - ▶ Is the proposed software product cost effective?
 - ▶ How long will it take to obtain a return on investment?
 - ▶ Alternatively, what will be the cost if the company decides not to develop the proposed software product?
 - ▶ If the software product is to be sold in the marketplace, have the necessary marketing studies been performed?
 - ▶ Can the proposed software product be delivered in time?
 - ▶ If the software product is to be developed to support the client organization's own activities, what will be the impact if the proposed software product is delivered late?

The Inception Phase: The Initial Business Case

Slide 3.244

- What are the risks involved in developing the software product
- How can these risks be mitigated?
 - ▶ Does the team (who will develop the proposed software product) have the necessary experience?
 - ▶ Is new hardware needed for this software product?
 - ▶ If so, is there a risk that it will not be delivered in time?
 - ▶ If so, is there a way to mitigate that risk, perhaps by ordering back-up hardware from another supplier?
 - ▶ Are software tools (CASE Tools) needed?
 - ▶ Are they currently available?
 - ▶ Do they have all the necessary functionality?

The Inception Phase: The Initial Business Case

Slide 3.245

- Answers are needed by the end of the **inception phase** so that the **initial business case** can be made

The Inception Phase: Risks

Slide 3.246

- There are three major risk categories:
 - ▶ Technical risks
 - See earlier slide
 - ▶ The risk of not getting the requirements right
 - Mitigated by performing the requirements workflow correctly
 - ▶ The risk of not getting the architecture right
 - The architecture may not be sufficiently robust

The Inception Phase: Risks

Slide 3.247

- To mitigate all three classes of risks
 - ▶ The risks need to be ranked so that the critical risks are mitigated first
- This concludes the steps of the inception phase that fall under the requirements workflow

The Inception Phase: Analysis, Design Workflows

Slide 3.248

- A small amount of the analysis workflow may be performed during the inception phase
 - ▶ Information needed for the design of the architecture is extracted
- Accordingly, a small amount of the **design workflow** may be performed, too, in inception phase

The Inception Phase: Implementation Workflow

Slide 3.249

- Coding is generally not performed during the inception phase
- However, a *proof-of-concept prototype* is sometimes build to test the feasibility of constructing a part of the software product

The Inception Phase: Test Workflow

Slide 3.250

- The test workflow commences almost at the start of the inception phase
 - ▶ The aim is to ensure that the requirements have been accurately determined

The Inception Phase: Planning

Slide 3.251

- There is insufficient information at the beginning of the inception phase to plan the entire development
 - ▶ The only planning that is done at the start of the project is the planning for the inception phase itself
- For the same reason, the only planning that can be done at the end of the inception phase is the plan for just the next phase, the **elaboration phase**

The Inception Phase: Documentation

Slide 3.252

- The **deliverables** of the inception phase include:
 - ▶ The initial version of the domain model
 - ▶ The initial version of the business model
 - ▶ The initial version of the requirements artifacts
 - ▶ A preliminary version of the analysis artifacts
 - ▶ A preliminary version of the architecture
 - ▶ The initial list of risks
 - ▶ The initial ordering of the use cases
 - ▶ The plan for the elaboration phase
 - ▶ The initial version of the business case

The Inception Phase: The Initial Business Case

Slide 3.253

- Obtaining the initial version of the business case is the overall aim of the inception phase
- This initial version incorporates
 - ▶ A description of the scope of the software product
 - ▶ Financial details
 - ▶ If the proposed software product is to be marketed, the business case will also include
 - **Revenue projections, market estimates, initial cost estimates**
 - ▶ If the software product is to be used in-house, the business case will include
 - **The initial cost–benefit analysis**

3.10.2 Elaboration Phase

Slide 3.254

- The aim of the **elaboration phase** is to refine the initial requirements
 - ▶ Refine the architecture
 - ▶ Monitor the risks and refine their priorities
 - ▶ Refine the business case
 - ▶ Produce the project management plan
- The major activities of the elaboration phase are refinements or elaborations of the previous phase

The Tasks of the Elaboration Phase

Slide 3.255

- The tasks of the **elaboration phase** correspond to:
 - ▶ All but completing the requirements workflow
 - ▶ Performing virtually the entire analysis workflow
 - ▶ Starting the design of the architecture

The Elaboration Phase: Documentation

Slide 3.256

- The **deliverables** of the elaboration phase include:
 - ▶ The completed domain model
 - ▶ The completed business model
 - ▶ The completed requirements artifacts
 - ▶ The completed analysis artifacts
 - ▶ An updated version of the architecture
 - ▶ An updated list of risks
 - ▶ The project management plan (for the rest of the project)
 - ▶ The completed business case

3.10.3 Construction Phase

Slide 3.257

- The aim of the **construction phase** is to produce the first operational-quality version of the software product
 - ▶ This is sometimes called the beta release.

The Tasks of the Construction Phase

Slide 3.258

- The emphasis in this phase is on
 - ▶ Implementation and
 - ▶ Testing
 - Unit testing of modules
 - Integration testing of subsystems
 - Product testing of the overall system

The Construction Phase: Documentation

Slide 3.259

- The deliverables of the construction phase include:
 - ▶ The initial user manual and other manuals, as appropriate
 - ▶ All the artifacts (beta release versions)
 - ▶ The completed architecture
 - ▶ The updated risk list
 - ▶ The project management plan (for the remainder of the project)
 - ▶ If necessary, the updated business case

3.10.4 The Transition Phase

Slide 3.260

- The aim of the **transition phase** is to ensure that the client's requirements have indeed been met
 - ▶ Faults in the software product are corrected
 - ▶ All the manuals are completed
 - ▶ Attempts are made to discover any previously unidentified risks
- This phase is driven by feedback from the site(s) at which the beta release has been installed

The Transition Phase: Documentation

Slide 3.261

- The deliverables of the transition phase include:
 - ▶ All the artifacts (final versions)
 - ▶ The completed manuals

The Phases of the Unified Process

Slide 3.262

- The *increments* are identified as *phases*

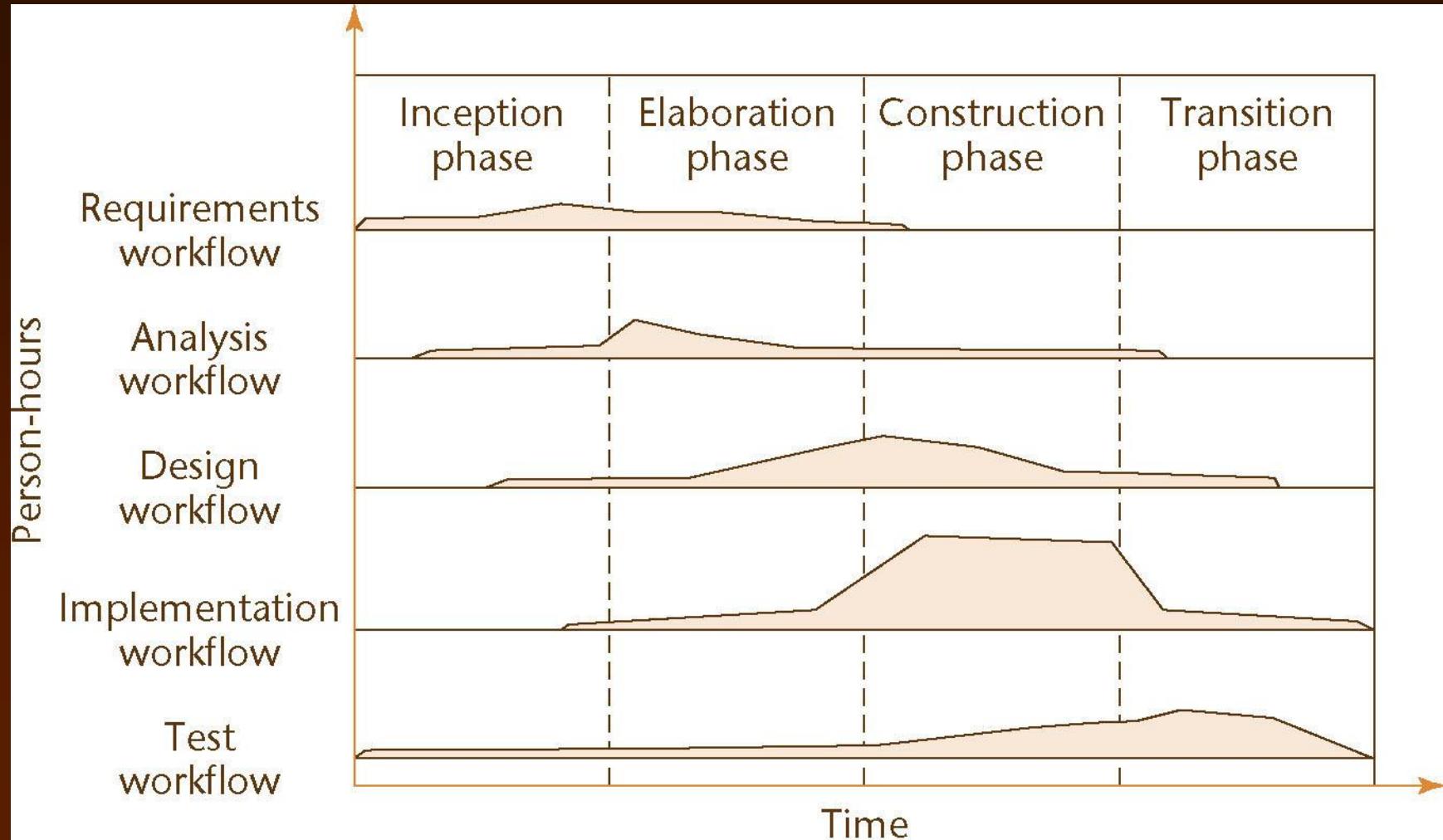


Figure 3.1

3.11 One- and Two-Dimensional Life-Cycle Models

Slide 3.263

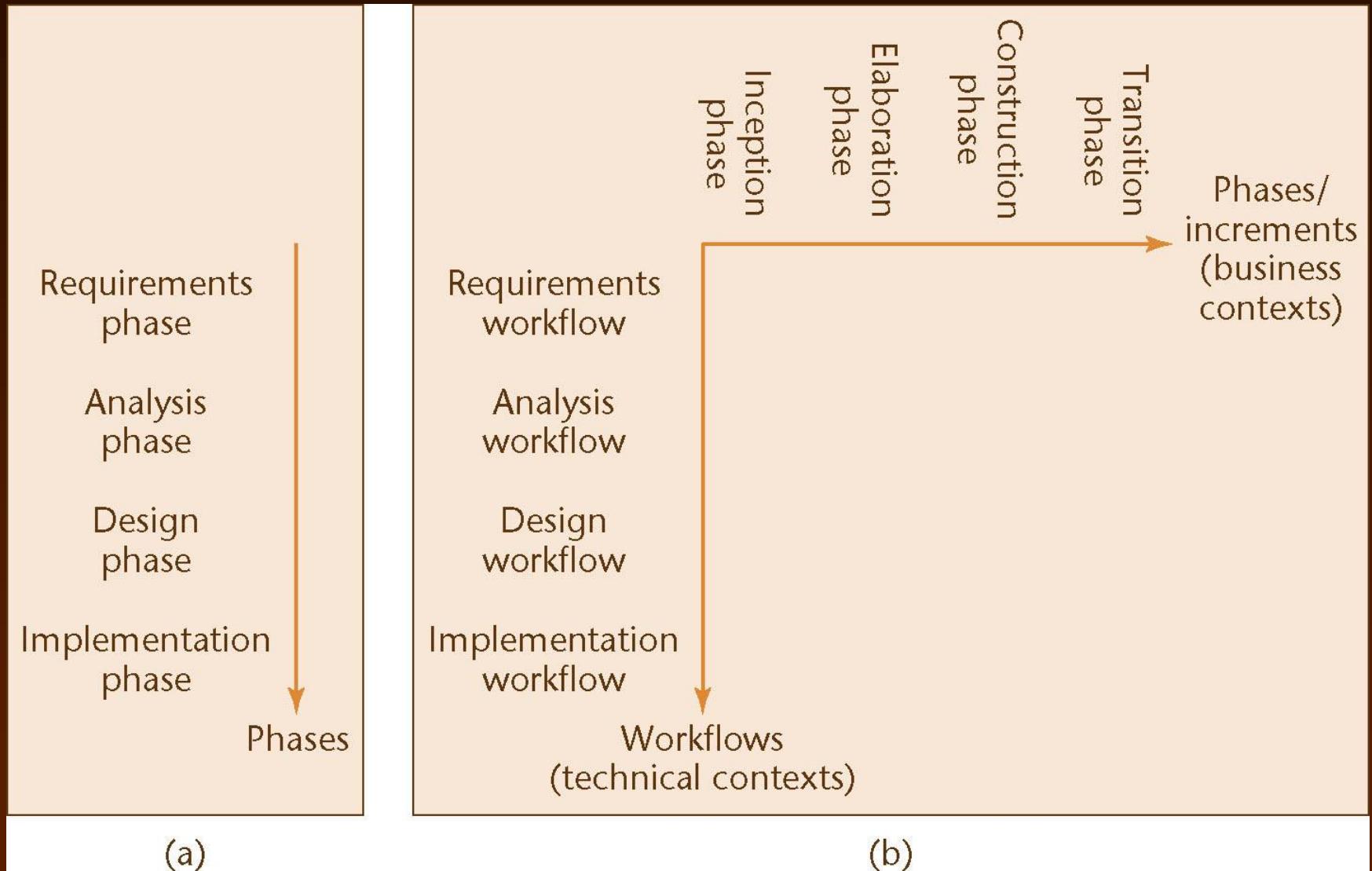


Figure 3.2

Why a Two-Dimensional Model?

Slide 3.264

- A traditional life cycle is a one-dimensional model
 - ▶ Represented by the single axis on the previous slide
 - Example: Waterfall model
- The Unified Process is a two-dimensional model
 - ▶ Represented by the two axes on the previous slide
- The two-dimensional figure shows
 - ▶ The workflows (**technical contexts**) and
 - ▶ The phases (**business contexts**)

Why a Two-Dimensional Model? (contd)

Slide 3.265

- The Waterfall model
- One-dimensional

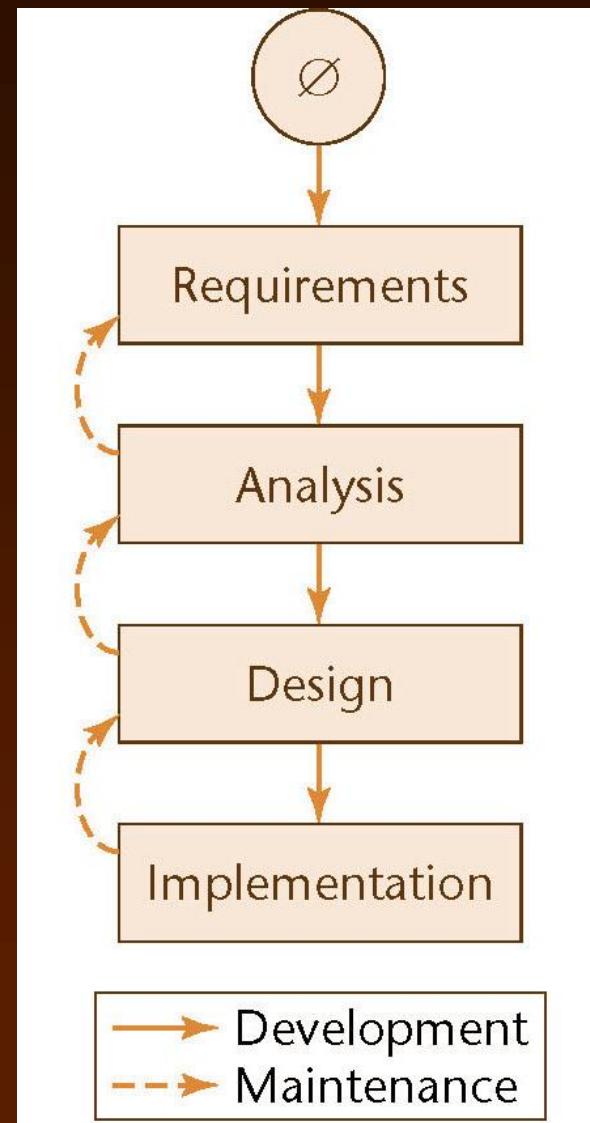


Figure 3.3

Why a Two-Dimensional Model? (contd)

Slide 3.266

- Evolution tree model
- Two-dimensional

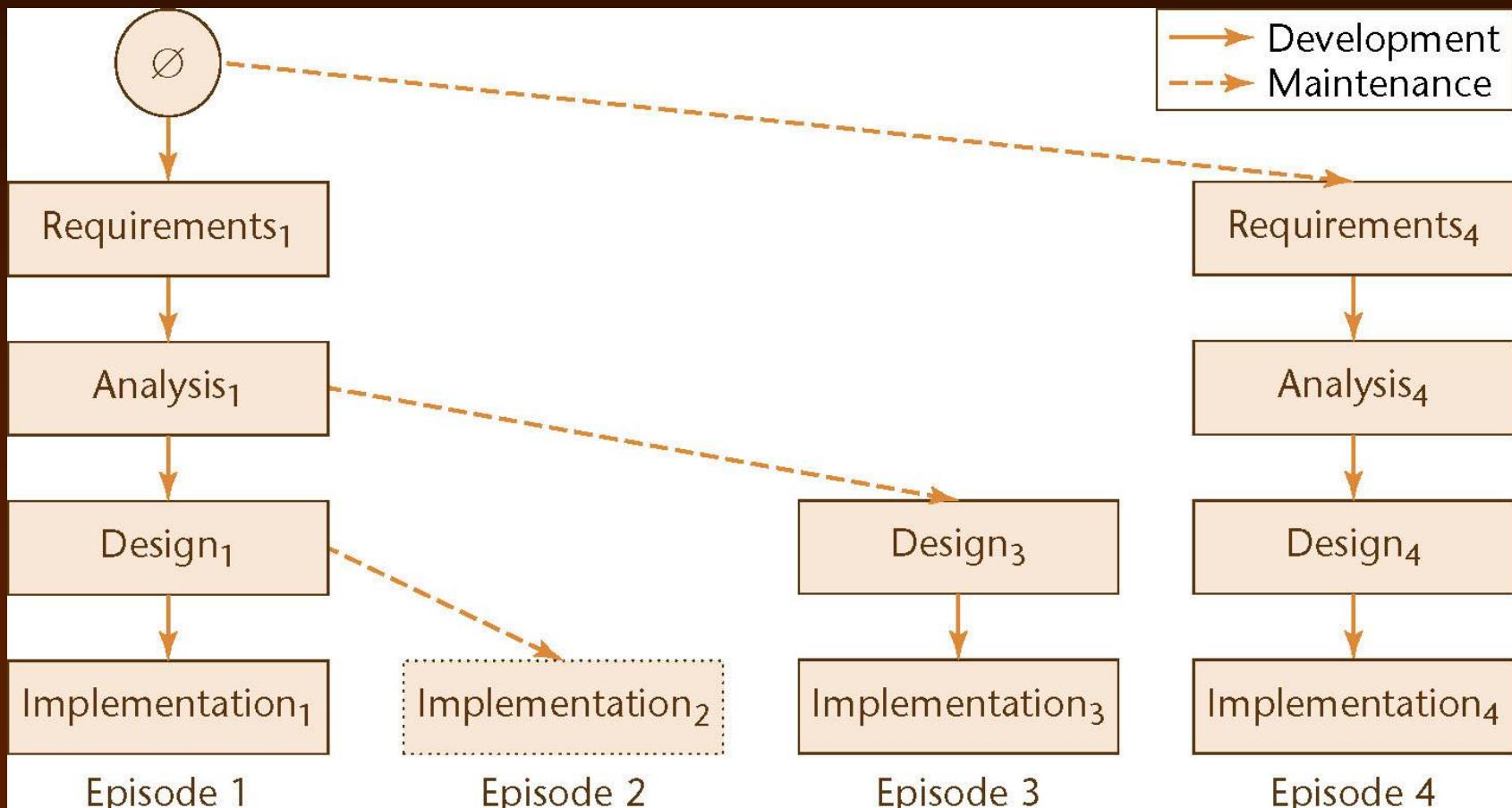


Figure 3.4

Why a Two-Dimensional Model? (contd)

Slide 3.267

- Are all the additional complications of the two-dimensional model necessary?
- In an ideal world, each workflow would be completed before the next workflow is started

Why a Two-Dimensional Model? (contd)

Slide 3.268

- In reality, the development task is too big for this
- As a consequence of Miller's Law
 - ▶ The development task has to be divided into increments (phases)
 - ▶ Within each increment, iteration is performed until the task is complete.

Why a Two-Dimensional Model? (contd)

Slide 3.269

- At the beginning of the process, there is not enough information about the software product to carry out the requirements workflow
 - ▶ Similarly for the other core workflows
- A software product has to be broken into subsystems
- Even subsystems can be too large at times
 - ▶ Components may be all that can be handled until a fuller understanding of all the parts of the product as a whole has been obtained

Why a Two-Dimensional Model? (contd)

Slide 3.270

- The Unified Process handles the inevitable changes well
 - ▶ The moving target problem
 - ▶ The inevitable mistakes
- The Unified Process is the best solution found to date for treating a large problem as a set of smaller, largely independent subproblems
 - ▶ It provides a framework for incrementation and iteration
 - ▶ In the future, it will inevitably be superseded by some better methodology.

Unified Process

Slide 3.271

Thank you.



Software Quality Management

Lecture # 4: Requirements Engineering

Requirements Engineering

- User requirements
- System requirements
- Functional requirements
- 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.

What is a Requirement?

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

Requirements Abstraction (Alan Davis)

“If a company wishes to let a contract for a large software development project, it must define its needs in a sufficiently abstract way that a solution is not pre-defined.

The requirements must be written so that several contractors can bid for the contract, offering, perhaps, different ways of meeting the client organization’s needs.

Once a contract has been awarded, the contractor must write a system definition for the client in more detail so that the client understands and can validate what the software will do.

Both of these documents may be called the requirements document for the system.”

Types of Requirement

- **User requirements**
 - Statements in natural language plus diagrams of the services the system provides and its operational constraints. Written for customers.
- **System requirements**
 - A structured document setting out detailed descriptions of the system's functions, services and operational constraints.
 - Defines what should be implemented so may be part of a contract between client and contractor.

User and System Requirements

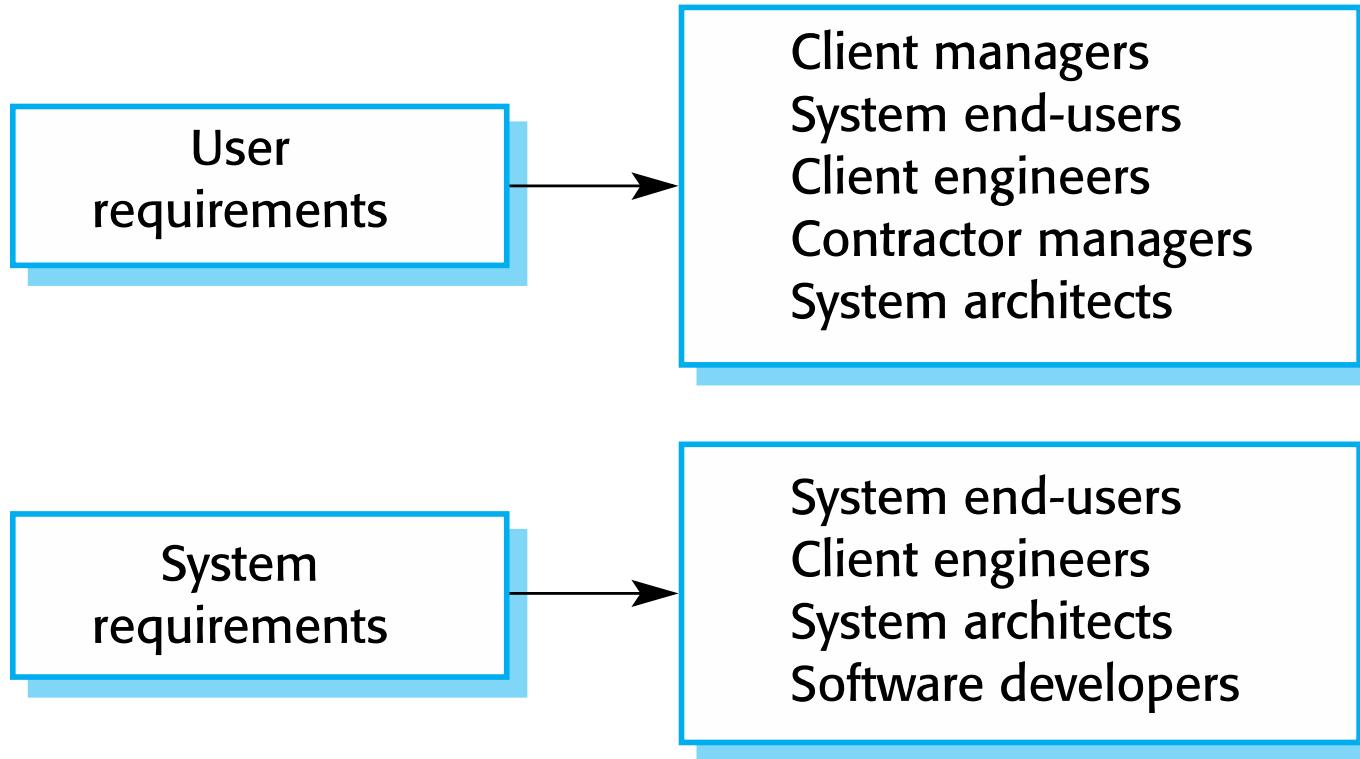
User requirements definition

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.

Readers of Different Types of Requirements Specification



System Stakeholders

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

Stakeholders in the Mentcare System (1 of 2)

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

Stakeholders in the Mentcare System (2 of 2)

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

Agile Methods and Requirements

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

Functional requirements 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.
 - May state what the system should not do.
- **Non-functional requirements**
 - **Constraints on the services or functions offered by the system** such as timing constraints, constraints on the development process, standards, etc.
 - Often apply to the system as a whole rather than individual features or services.
- **Domain requirements**
 - Constraints on the system from the domain of operation

Functional Requirements

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

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.

Requirements Imprecision

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

Requirements

Completeness and Consistency

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

Non-Functional Requirements

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

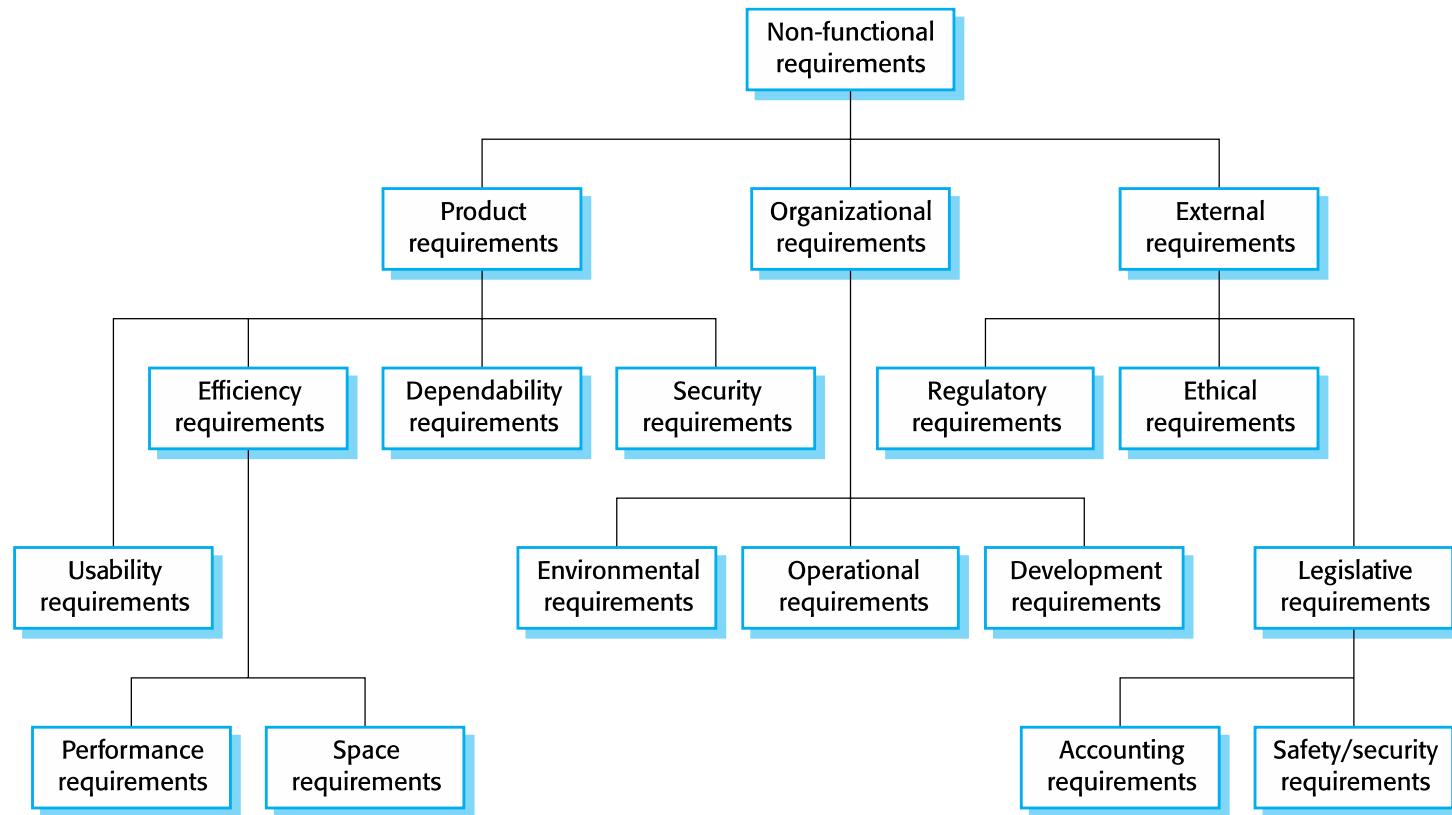
Non-Functional Requirements Implementation

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

Non-Functional Requirements Classification

- **Product requirements**
 - Requirements which specify that the delivered product must behave in a particular way e.g. execution speed, reliability, etc.
- **Organizational requirements**
 - Requirements which are a consequence of organisational policies and procedures e.g. process standards used, implementation requirements, etc.
- **External requirements**
 - Requirements which arise from factors which are external to the system and its development process e.g. interoperability requirements, legislative requirements, etc.

Types of Nonfunctional Requirements



Examples of Nonfunctional Requirements in the Mentcare System

Product requirement

The Mentcare system shall be available to all clinics during normal working hours (Mon–Sat, 0830 to 1730). Downtime within normal working hours shall not exceed five seconds in any one day.

Organizational requirement

Users of the Mentcare system shall authenticate themselves using their health authority identity card.

External requirement

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

Goals and Requirements

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

Usability Requirements: Examples

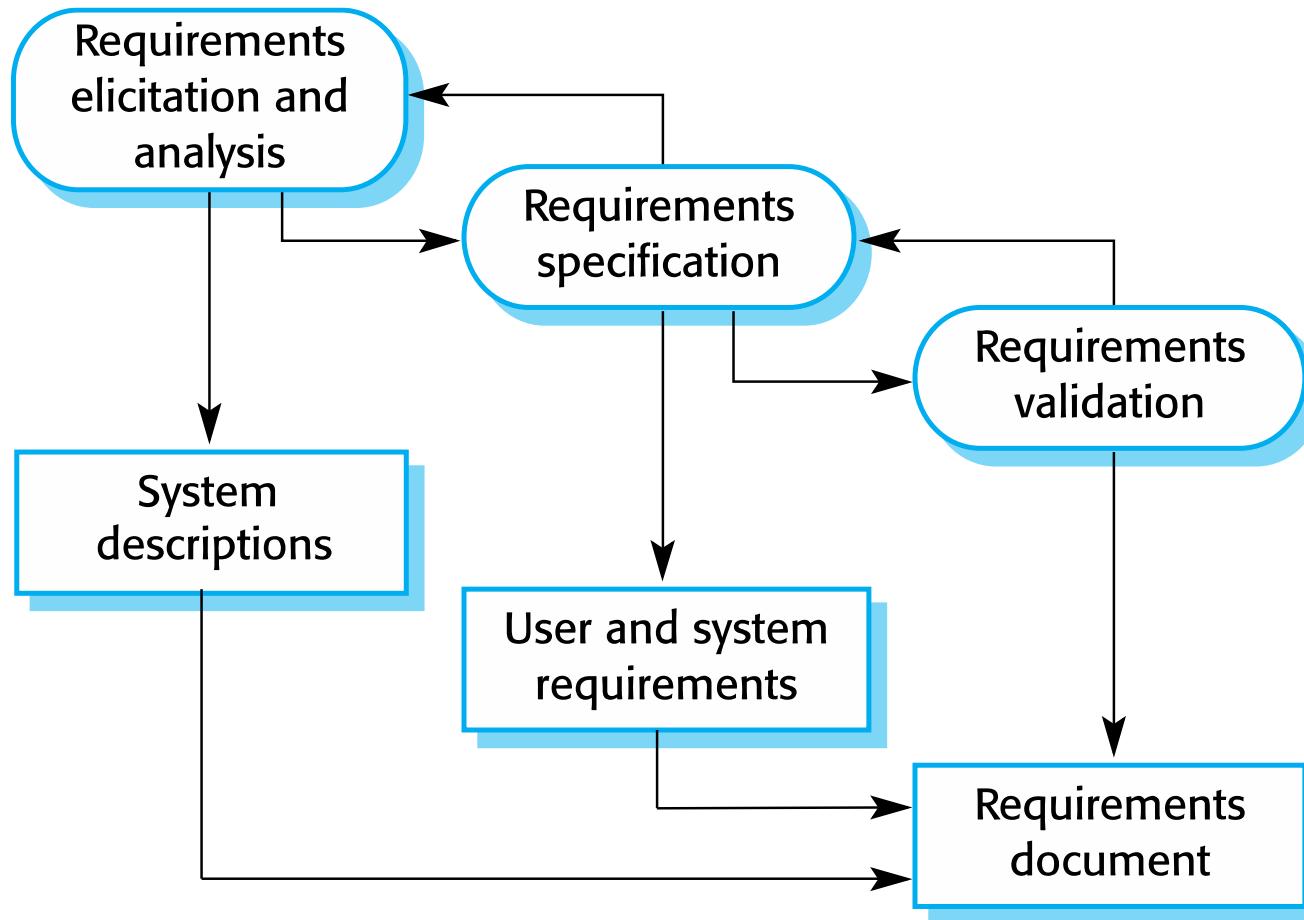
- The system should be easy to use by medical staff and should be organized in such a way that user errors are minimized. **(Goal)**
- Medical staff shall be able to use all the system functions after four hours of training. After this training, the average number of errors made by experienced users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

Metrics for Specifying Nonfunctional Requirements

System 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 Engineering Processes

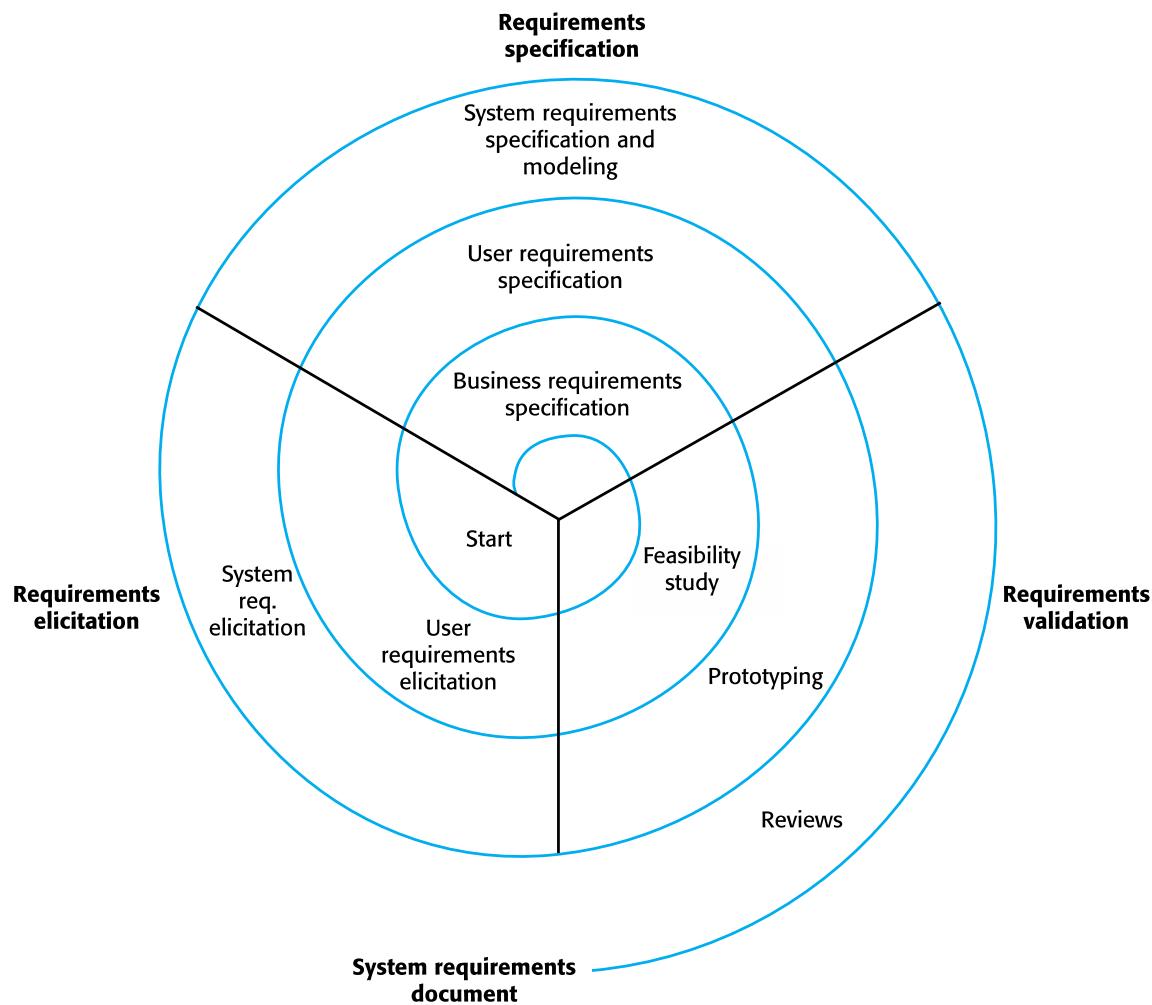
The Requirements Engineering Process



Requirements Engineering Processes

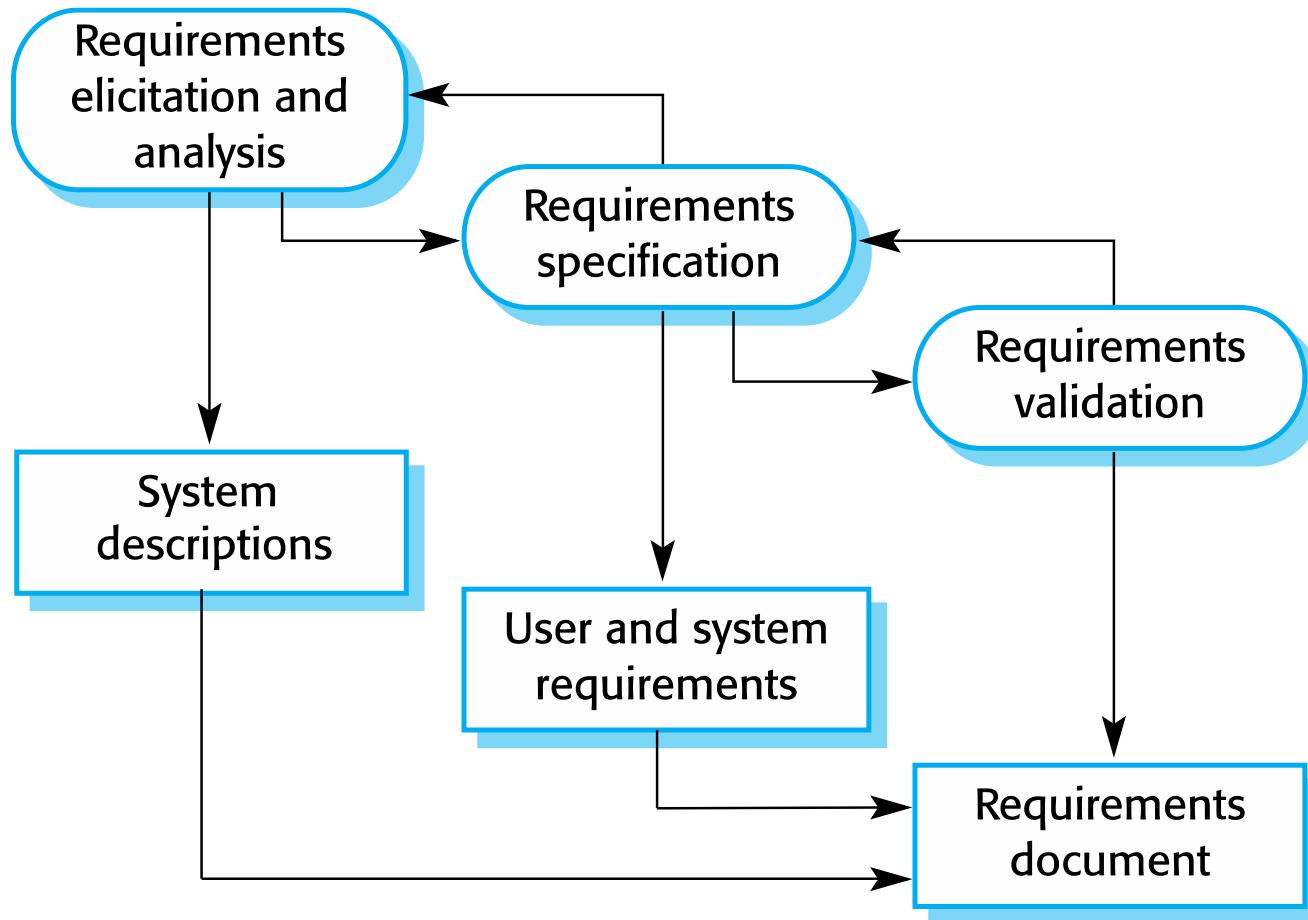
- The processes used for Requirements Engineering vary widely depending on the application domain, the people involved and the organisation developing the requirements.
- However, there are a number of **generic activities** common to all Requirements Engineering processes:
 - Requirements Elicitation
 - Requirements Analysis
 - Requirements Validation
 - Requirements Management
- In practice, **Requirements Engineering is an iterative process** in which these activities are interleaved.

A Spiral View of the Requirements Engineering Process



Requirements Elicitation

The Requirements Engineering Process



Requirements Elicitation and Analysis

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

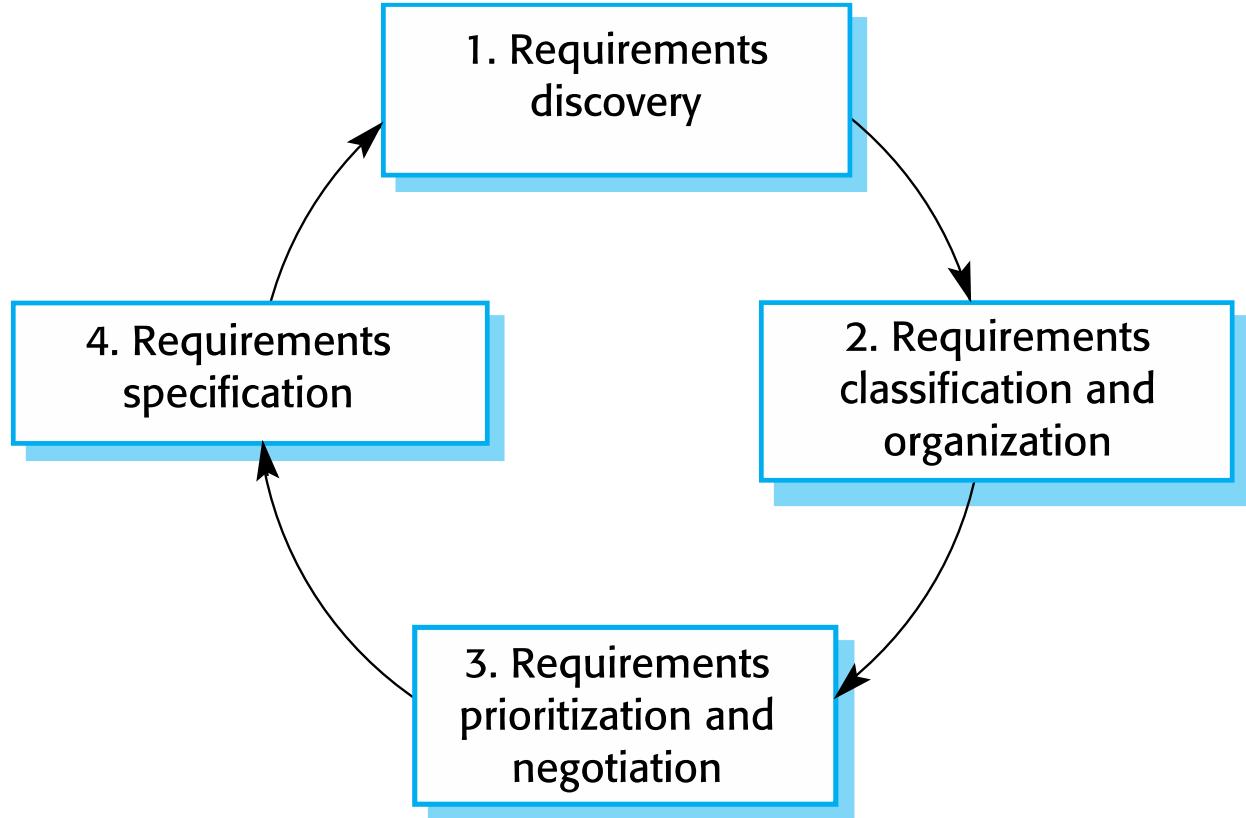
Requirements Elicitation

- Software engineers work with a range of system stakeholders to find out about the application domain, the services that the system should provide, the required system performance, hardware constraints, other systems, etc.
- Stages of **Requirements Elicitation** 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.
- Organisational and political factors may influence the system requirements.
- The requirements change during the analysis process. New stakeholders may emerge and the business environment may change.

The Requirements Elicitation and Analysis Process



Requirements Elicitation Process Activities

- **Requirements discovery**
 - Interacting with stakeholders to discover their requirements. Domain requirements are also discovered at this stage.
- **Requirements classification and organisation**
 - Groups related requirements and organises them into coherent clusters.
- **Prioritisation and negotiation**
 - Prioritising requirements and resolving requirements conflicts.
- **Requirements specification**
 - Requirements are documented and input into the next round of the spiral.

Requirements Discovery

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

Interviewing

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

Interviews in Practice

- Normally a mix of closed and open-ended interviewing.
- Interviews are good for getting an overall understanding of what stakeholders do and how they might interact with the system.
- Interviewers need to be open-minded without pre-conceived ideas of what the system should do
- You need to prompt the user to talk about the system by suggesting requirements rather than simply asking them what they want.

Problems with Interviews

- Application specialists may use language to describe their work that isn't easy for the requirements engineer to understand.
- Interviews are not good for understanding domain requirements
 - Requirements engineers cannot understand specific domain terminology;
 - Some domain knowledge is so familiar that people find it hard to articulate or think that it isn't worth articulating.

Ethnography

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

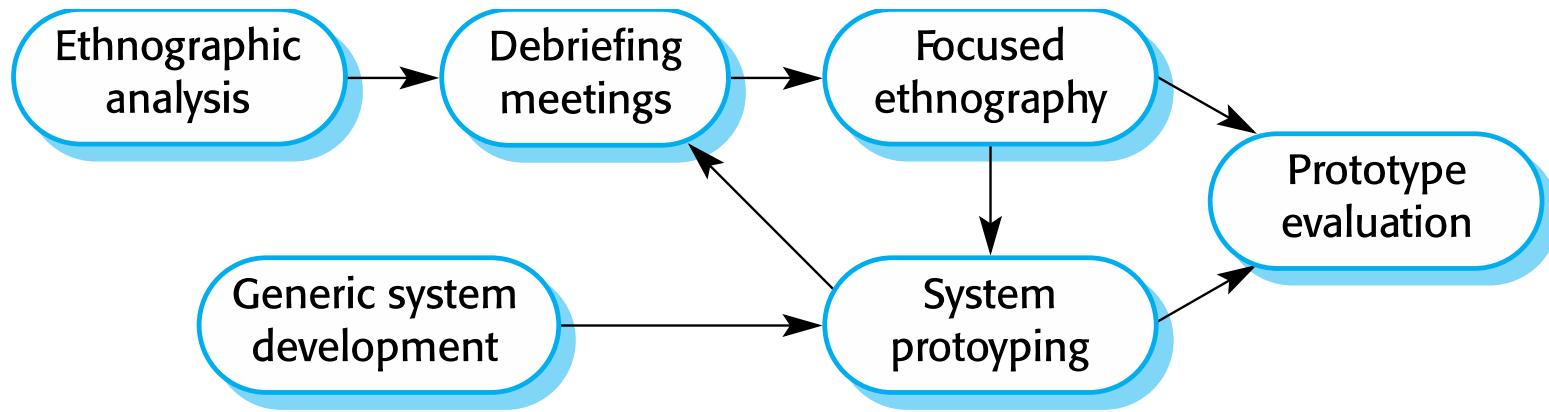
Scope of Ethnography

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

Focused Ethnography

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

Ethnography and Prototyping for Requirements Analysis



Stories and Scenarios

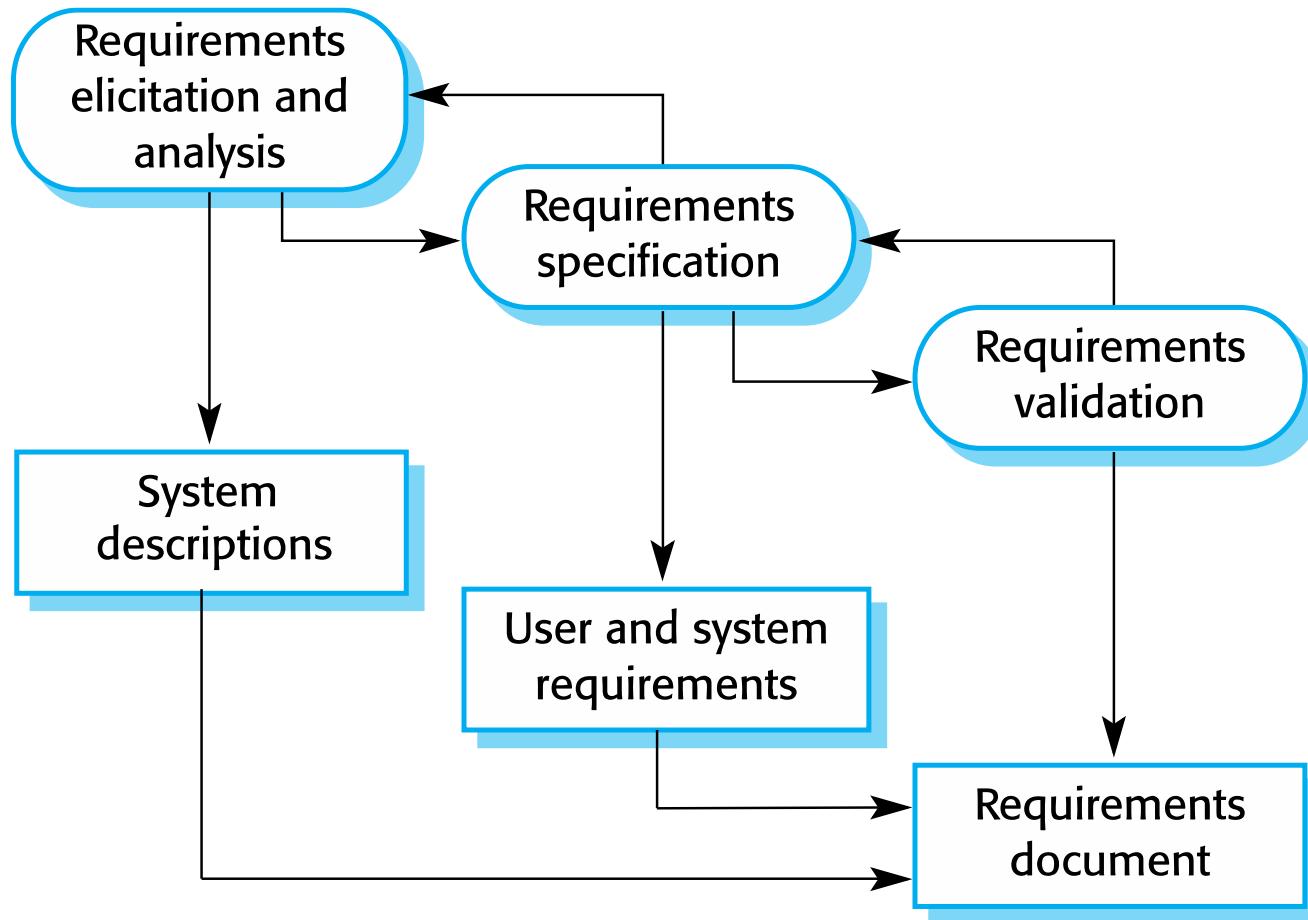
- Scenarios and user stories are real-life examples of how a system can be used.
- Stories and scenarios are a description of how a system may be used for a particular task.
- Because they are based on a practical situation, stakeholders can relate to them and can comment on their situation with respect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

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

The Requirements Engineering Process



Requirements Specification

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

Ways of Writing a System Requirements Specification

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

Requirements and Design

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

Natural Language Specification

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

Guidelines for Writing Requirements

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

Problems with Natural Language

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

Example Requirements for the Insulin Pump Software System

- 3.2 The system shall measure the blood sugar and deliver insulin, if required, every 10 minutes. (Changes in blood sugar are relatively slow so more frequent measurement is unnecessary; less frequent measurement could lead to unnecessarily high sugar levels.)

- 3.6 The system shall run a self-test routine every minute with the conditions to be tested and the associated actions defined in Table 1. (A self-test routine can discover hardware and software problems and alert the user to the fact the normal operation may be impossible.)

Structured Specifications

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

Form-Based Specifications

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

A Structured Specification of a Requirement for an Insulin Pump (1 of 2)

Insulin Pump/Control Software/SRS/3.3.2

Function Compute insulin dose: safe sugar level

Description

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

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

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

Outputs CompDose—the dose in insulin to be delivered.

Destination Main control loop.

A Structured Specification of a Requirement for an Insulin Pump (2 of 2)

Action

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

Requirements

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

Pre-condition

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

Post-condition r_0 is replaced by r_1 then r_1 is replaced by r_2 .

Side effects None.

Tabular Specification

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

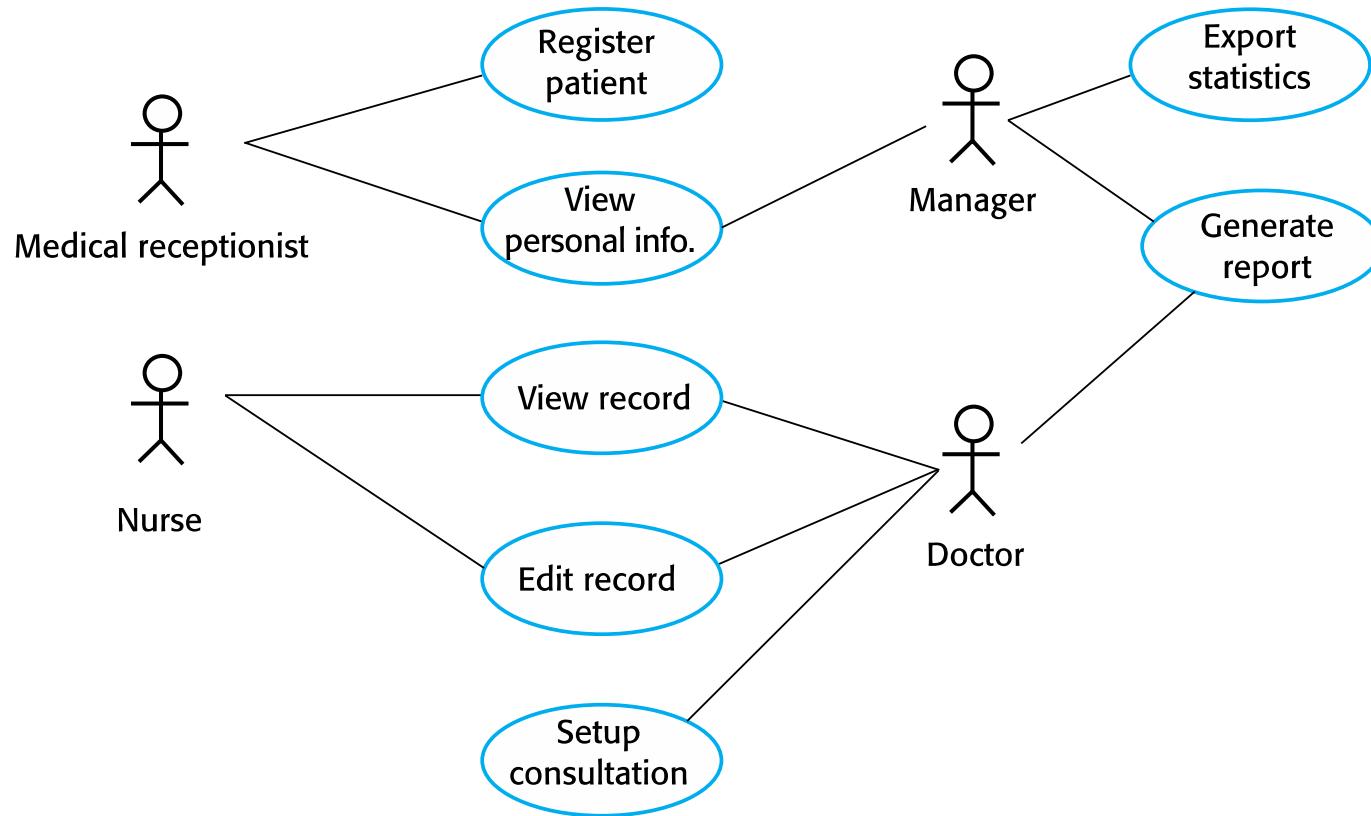
Tabular Specification of Computation for an Insulin Pump

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

Use Cases

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

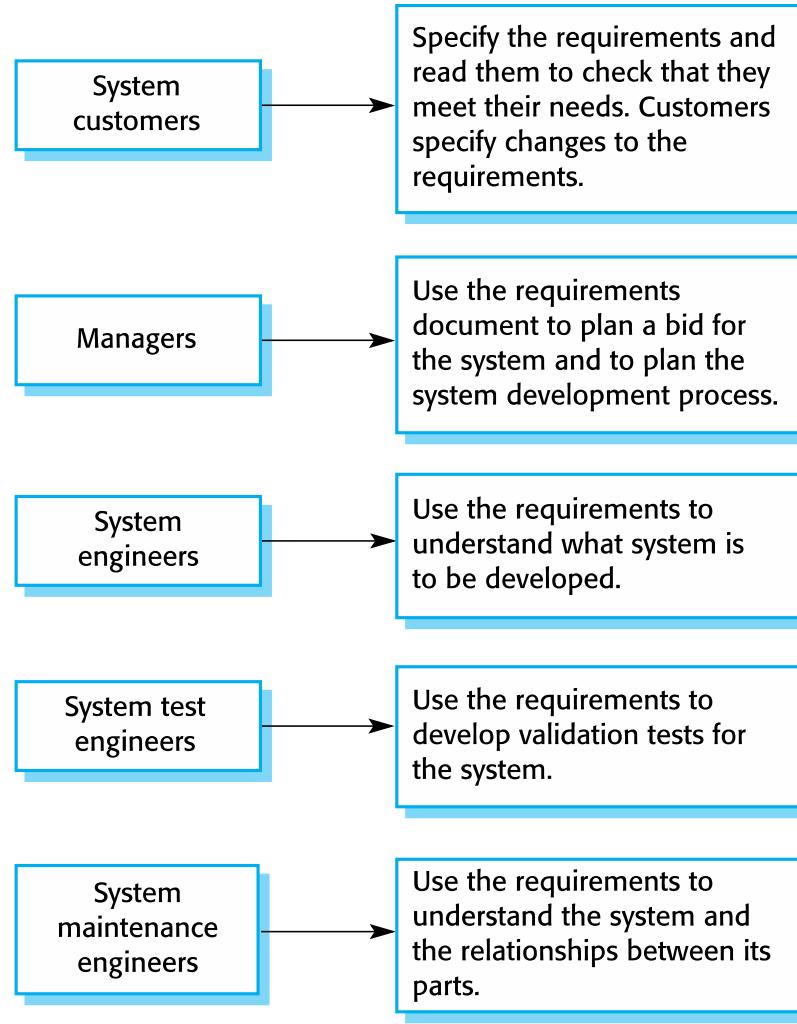
Use Cases for the Mentcare System



The Software Requirements Document

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

Users of a Requirements Document



Requirements Document Variability

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

The Structure of a Requirements Document (1 of 2)

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 (2 of 2)

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 Validation Techniques

- Requirements reviews
 - Systematic manual analysis of the requirements.
- Prototyping
 - Using an executable model of the system to check requirements.
- Test-case generation
 - Developing tests for requirements to check testability.

Requirements Reviews

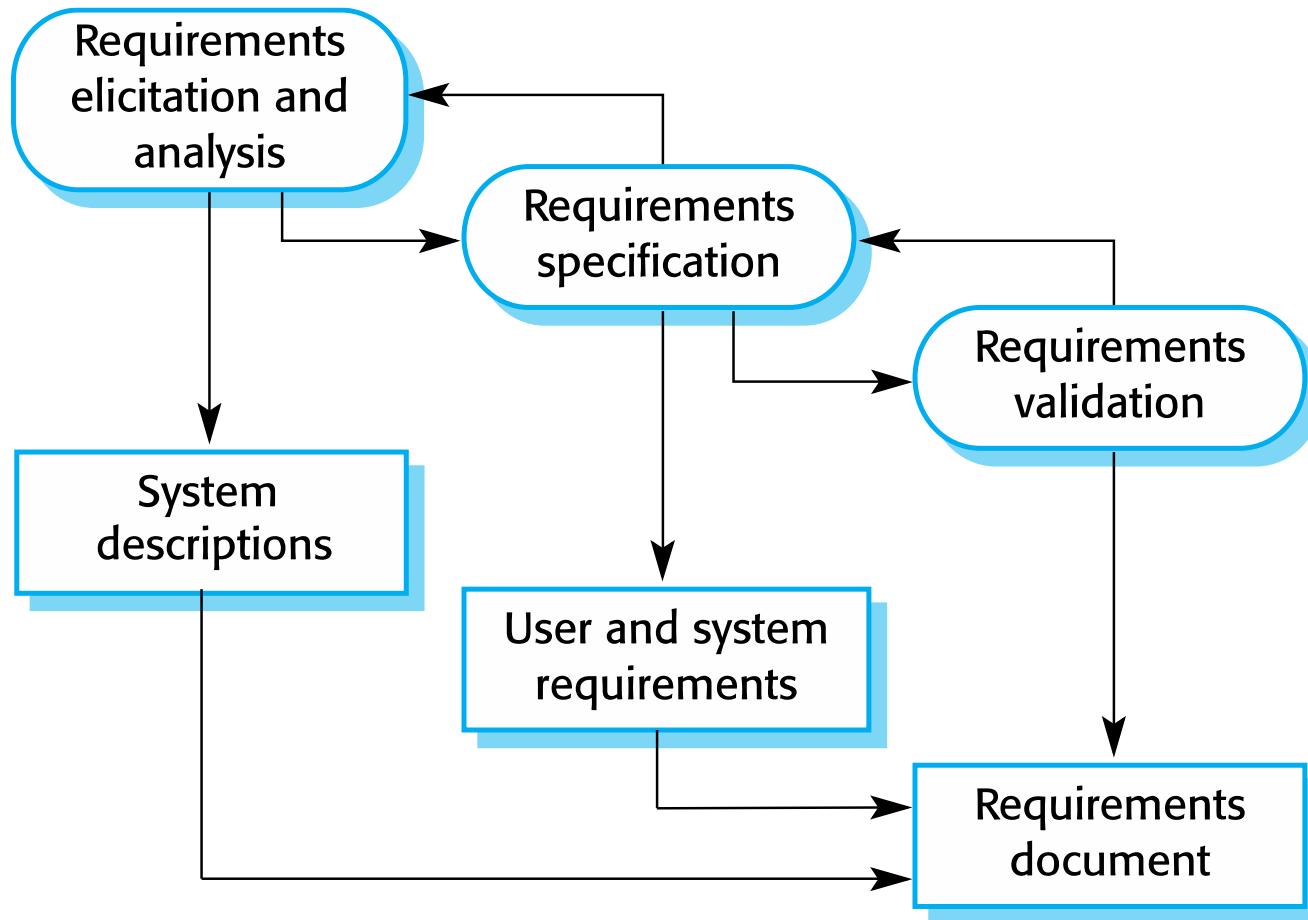
- Regular reviews should be held while the requirements definition is being formulated.
- Both client and contractor staff should be involved in reviews.
- Reviews may be formal (with completed documents) or informal. Good communication between developers, customers and users can resolve problems at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

Review Checks

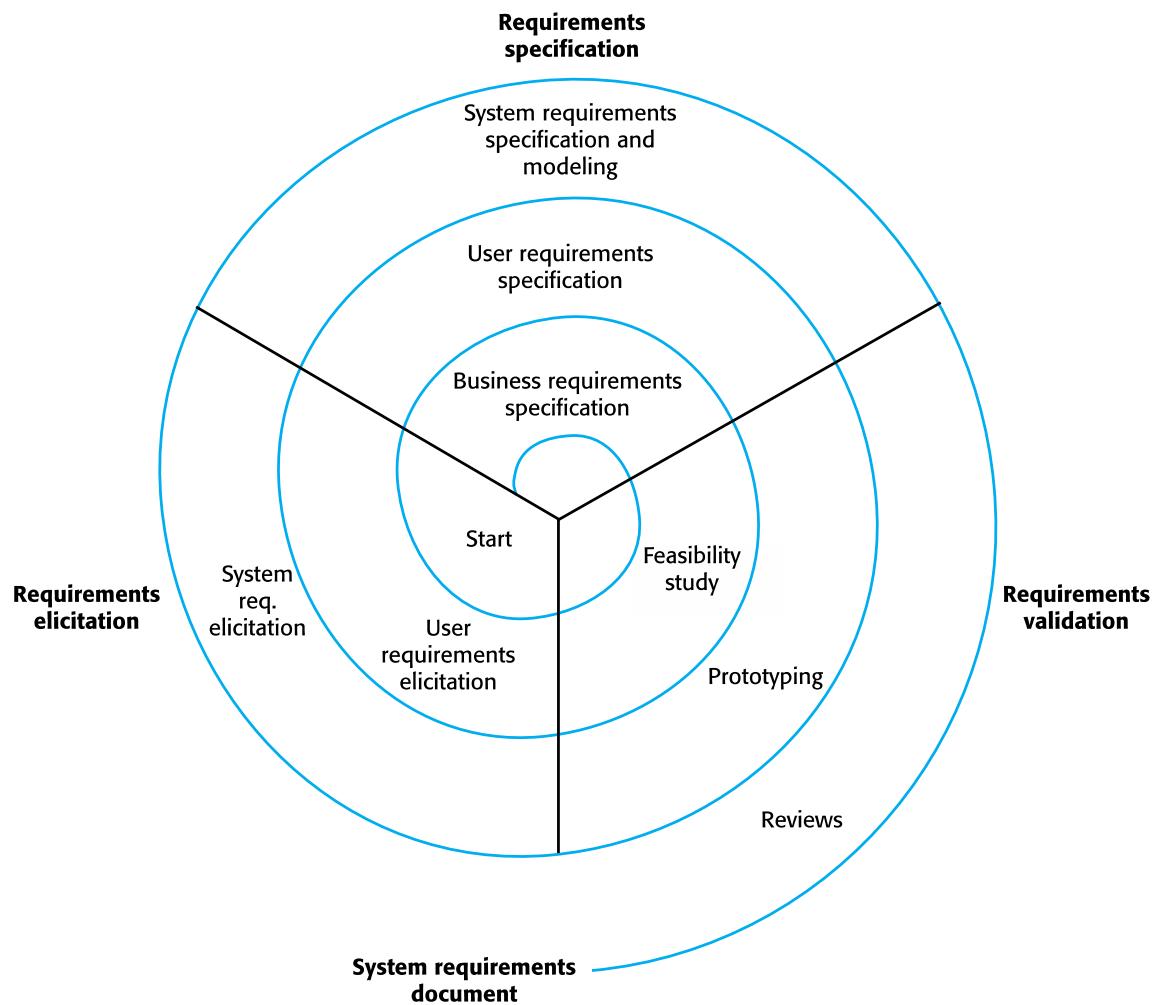
- **Verifiability**
 - Is the requirement realistically testable?
- **Comprehensibility**
 - Is the requirement properly understood?
- **Traceability**
 - Is the origin of the requirement clearly stated?
- **Adaptability**
 - Can the requirement be changed without a large impact on other requirements?

Requirements Change

The Requirements Engineering Process



A Spiral View of the Requirements Engineering Process



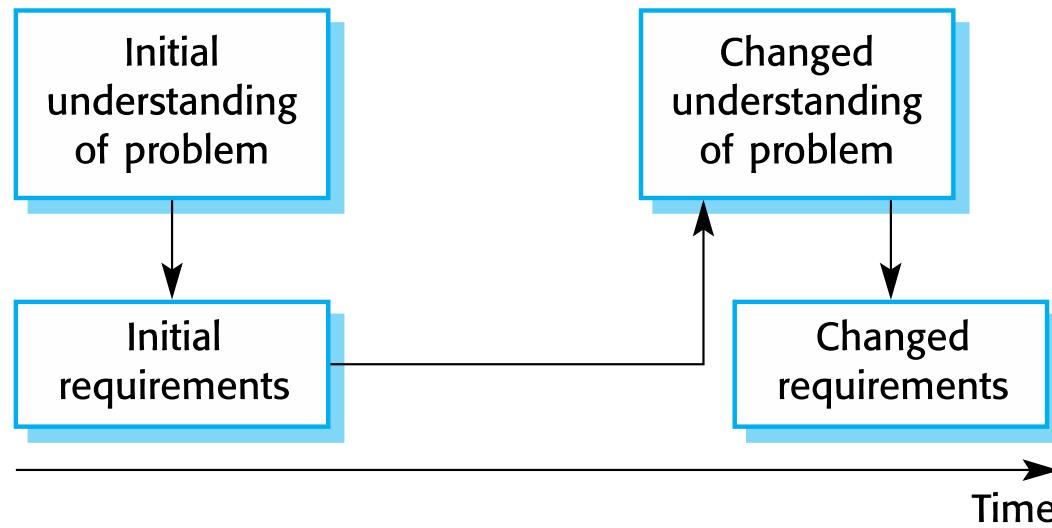
Changing Requirements (1 of 2)

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

Changing Requirements (2 of 2)

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

Requirements Evolution



Requirements Management

- 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.

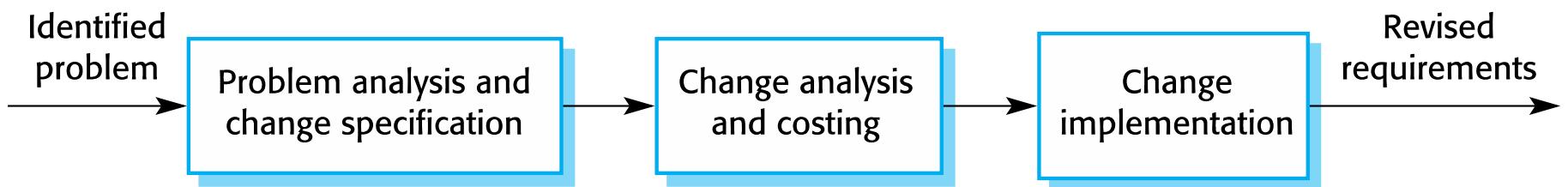
Requirements Management Planning

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

Requirements Change Management (1 of 2)

- Deciding if a requirements change should be accepted
 - **Problem analysis and change specification**
 - During this stage, the problem or the change proposal is analyzed to check that it is valid. This analysis is fed back to the change requestor who may respond with a more specific requirements change proposal, or decide to withdraw the request.
 - **Change analysis and costing**
 - The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements. Once this analysis is completed, a decision is made whether or not to proceed with the requirements change.
 - **Change implementation**
 - The requirements document and, where necessary, the system design and implementation, are modified. Ideally, the document should be organized so that changes can be easily implemented.

Requirements Change Management (2 of 2)



Requirements Engineering: Key Points

Requirements Engineering: Key Points (1 of 4)

- 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 (2 of 4)

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

Key Points (3 of 4)

- **Requirements specification** is the process of formally documenting the user and system requirements and creating a software requirements document.
- The **software requirements specification document** is an agreed statement of the system requirements. It should be organized so that both system customers and software developers can use it.

Key Points (4 of 4)

- 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.



Thank you.



Software Quality Management

Lecture # 4: Requirements Engineering

Requirements Engineering

- User requirements
- System requirements
- Functional requirements
- 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.

What is a Requirement?

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

Requirements Abstraction (Alan Davis)

“If a company wishes to let a contract for a large software development project, it must define its needs in a sufficiently abstract way that a solution is not pre-defined.

The requirements must be written so that several contractors can bid for the contract, offering, perhaps, different ways of meeting the client organization’s needs.

Once a contract has been awarded, the contractor must write a system definition for the client in more detail so that the client understands and can validate what the software will do.

Both of these documents may be called the requirements document for the system.”

Types of Requirement

- **User requirements**
 - Statements in natural language plus diagrams of the services the system provides and its operational constraints. Written for customers.
- **System requirements**
 - A structured document setting out detailed descriptions of the system's functions, services and operational constraints.
 - Defines what should be implemented so may be part of a contract between client and contractor.

User and System Requirements

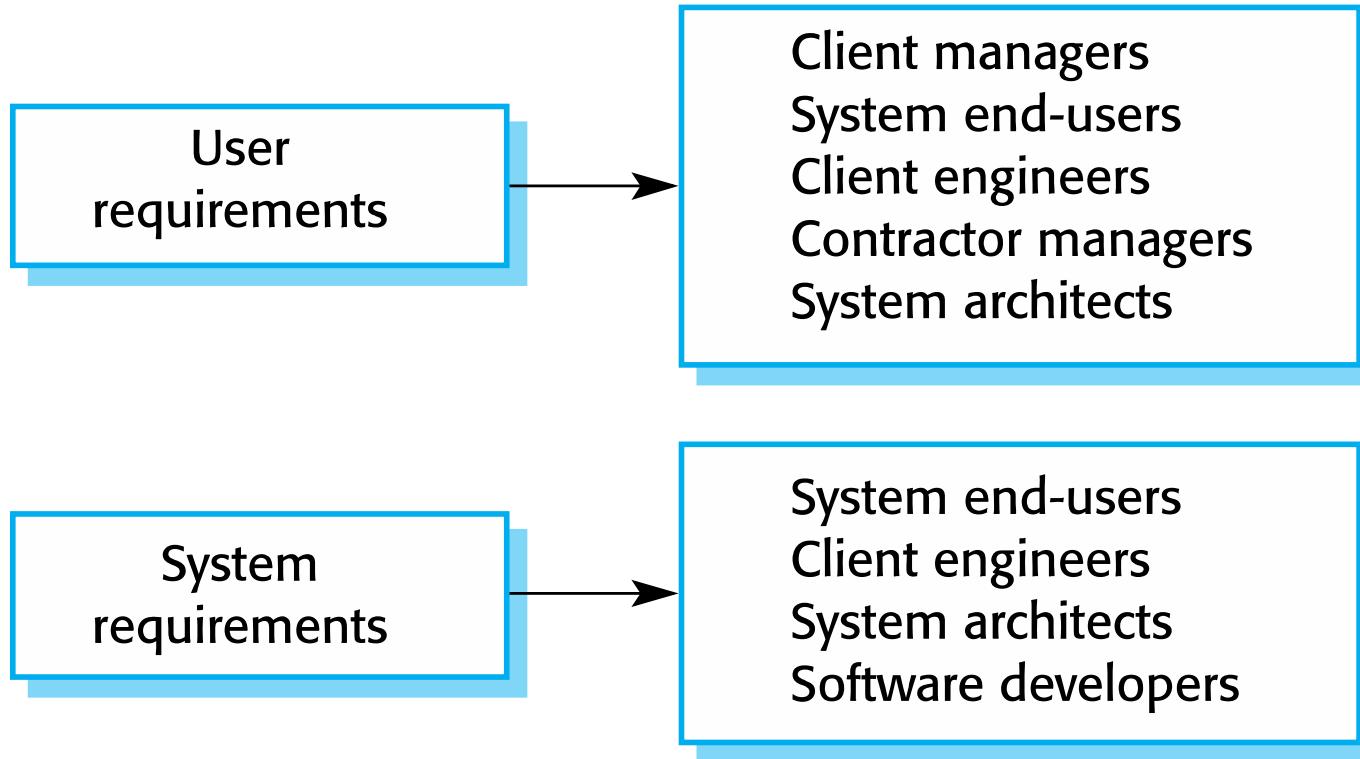
User requirements definition

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.

Readers of Different Types of Requirements Specification



System Stakeholders

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

Stakeholders in the Mentcare System (1 of 2)

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

Stakeholders in the Mentcare System (2 of 2)

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

Agile Methods and Requirements

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

Functional requirements 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.
 - May state what the system should not do.
- **Non-functional requirements**
 - **Constraints on the services or functions offered by the system** such as timing constraints, constraints on the development process, standards, etc.
 - Often apply to the system as a whole rather than individual features or services.
- **Domain requirements**
 - Constraints on the system from the domain of operation

Functional Requirements

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

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.

Requirements Imprecision

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

Requirements

Completeness and Consistency

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

Non-Functional Requirements

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

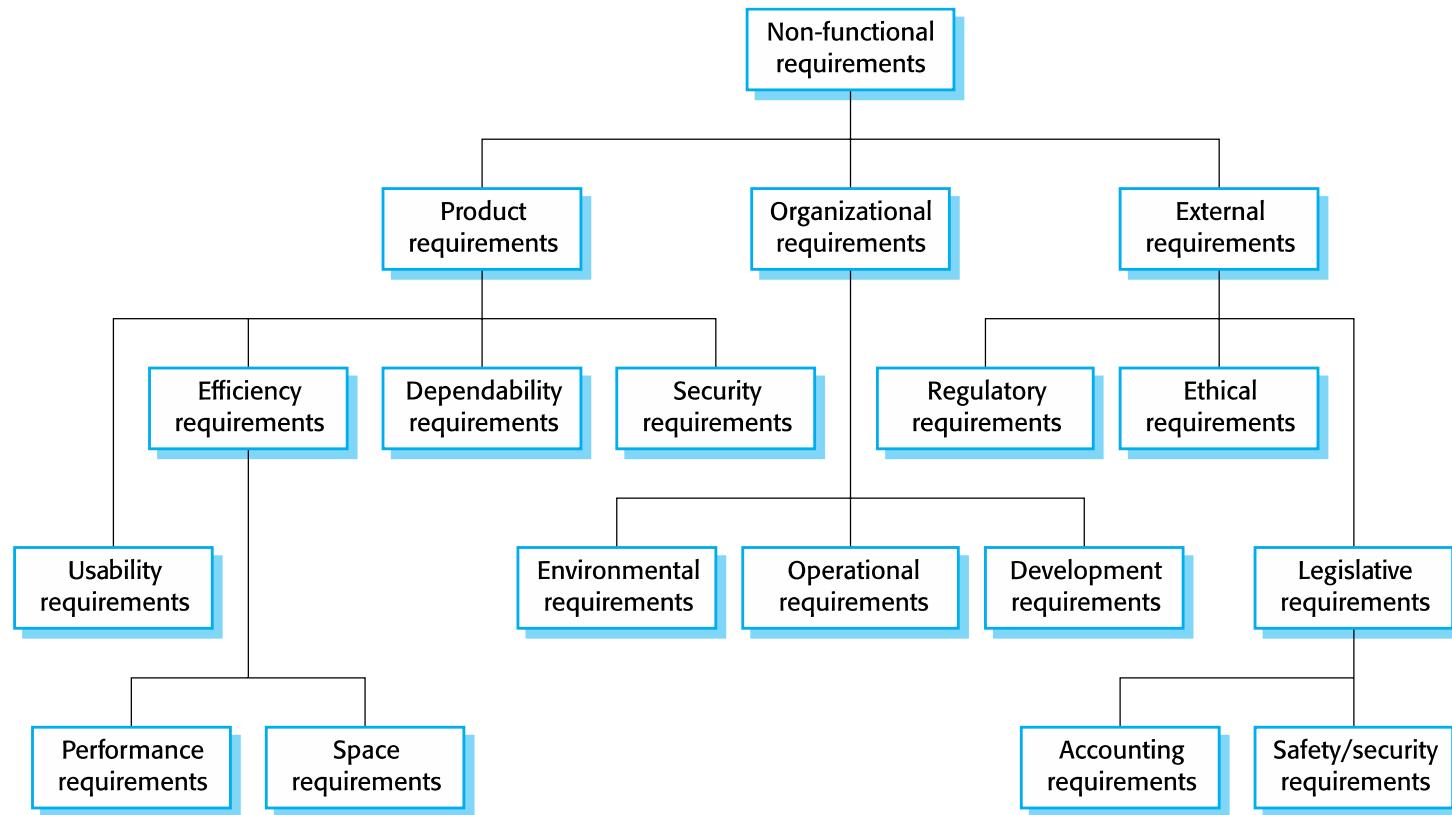
Non-Functional Requirements Implementation

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

Non-Functional Requirements Classification

- **Product requirements**
 - Requirements which specify that the delivered product must behave in a particular way e.g. execution speed, reliability, etc.
- **Organizational requirements**
 - Requirements which are a consequence of organisational policies and procedures e.g. process standards used, implementation requirements, etc.
- **External requirements**
 - Requirements which arise from factors which are external to the system and its development process e.g. interoperability requirements, legislative requirements, etc.

Types of Nonfunctional Requirements



Examples of Nonfunctional Requirements in the Mentcare System

Product requirement

The Mentcare system shall be available to all clinics during normal working hours (Mon–Sat, 0830 to 1730). Downtime within normal working hours shall not exceed five seconds in any one day.

Organizational requirement

Users of the Mentcare system shall authenticate themselves using their health authority identity card.

External requirement

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

Goals and Requirements

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

Usability Requirements: Examples

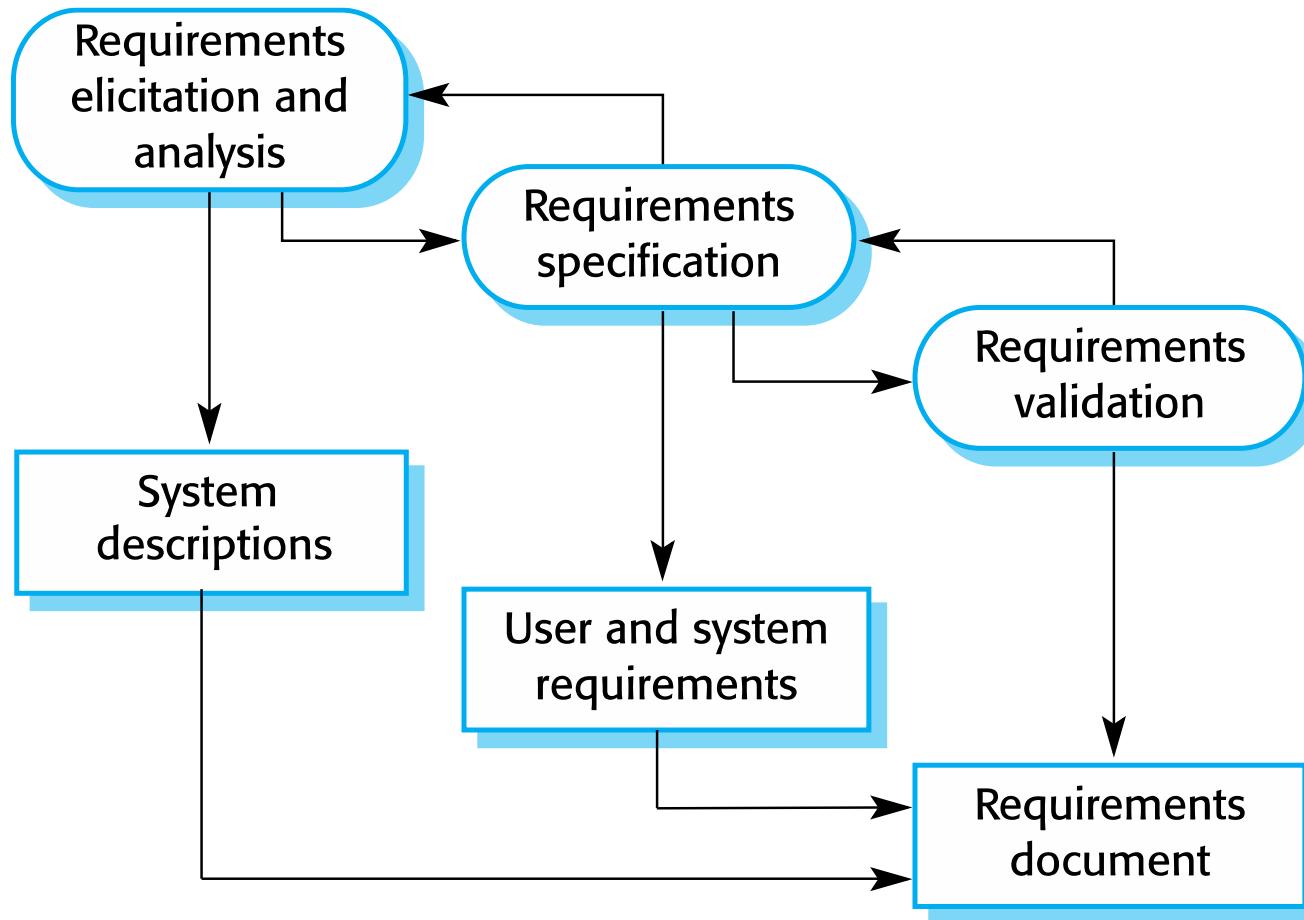
- The system should be easy to use by medical staff and should be organized in such a way that user errors are minimized. **(Goal)**
- Medical staff shall be able to use all the system functions after four hours of training. After this training, the average number of errors made by experienced users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

Metrics for Specifying Nonfunctional Requirements

System 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 Engineering Processes

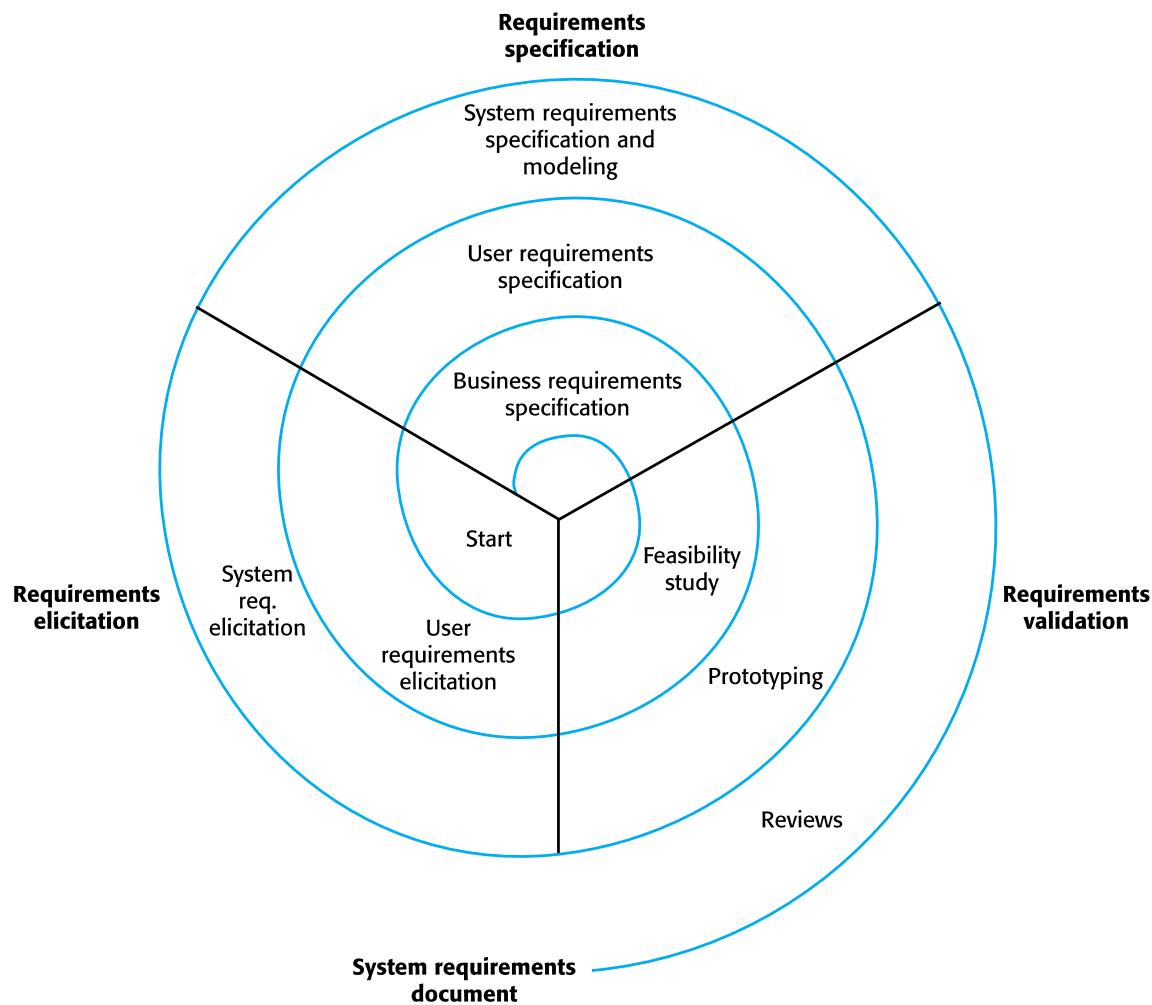
The Requirements Engineering Process



Requirements Engineering Processes

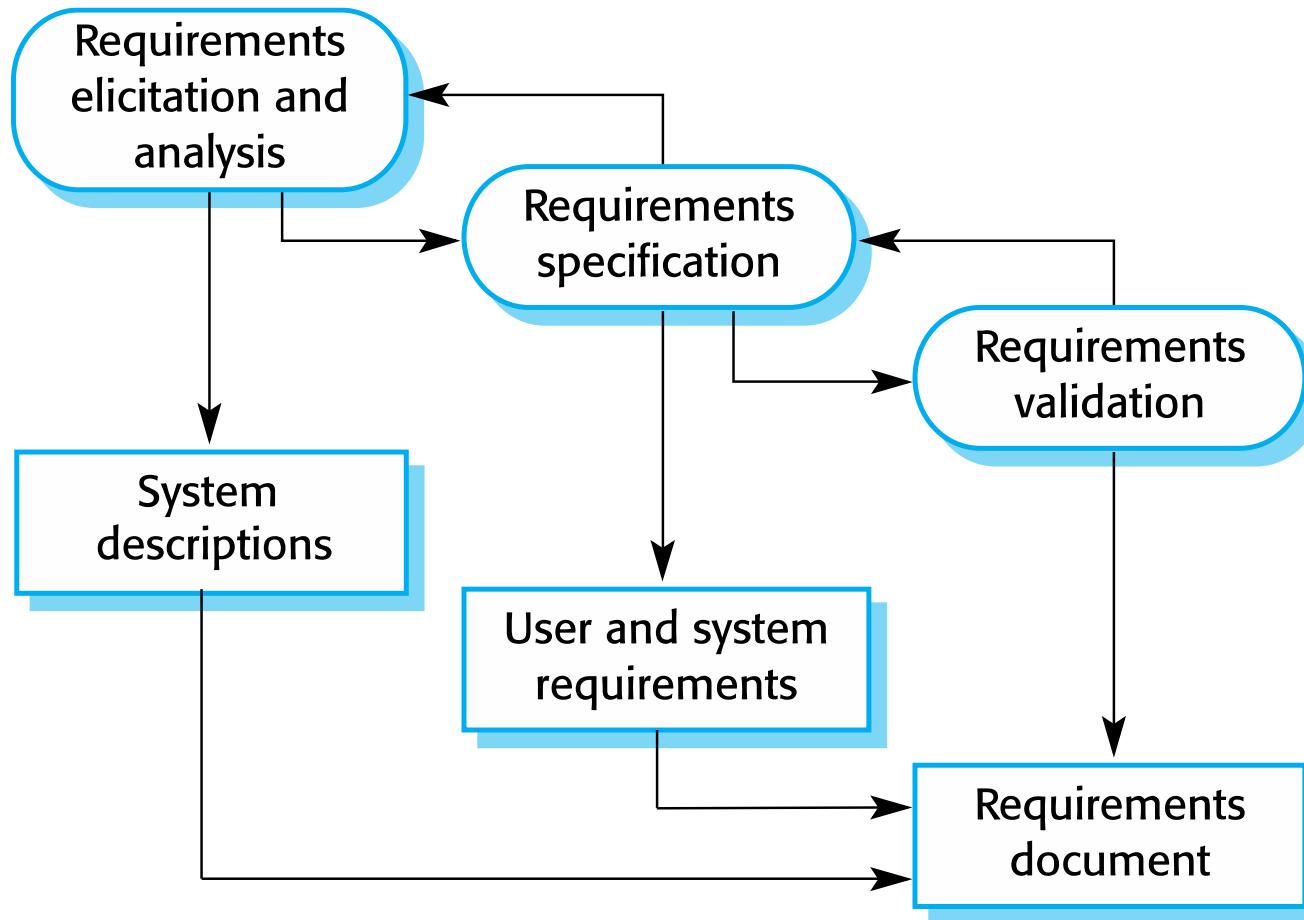
- The processes used for Requirements Engineering vary widely depending on the application domain, the people involved and the organisation developing the requirements.
- However, there are a number of **generic activities** common to all Requirements Engineering processes:
 - Requirements Elicitation
 - Requirements Analysis
 - Requirements Validation
 - Requirements Management
- In practice, **Requirements Engineering is an iterative process** in which these activities are interleaved.

A Spiral View of the Requirements Engineering Process



Requirements Elicitation

The Requirements Engineering Process



Requirements Elicitation and Analysis

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

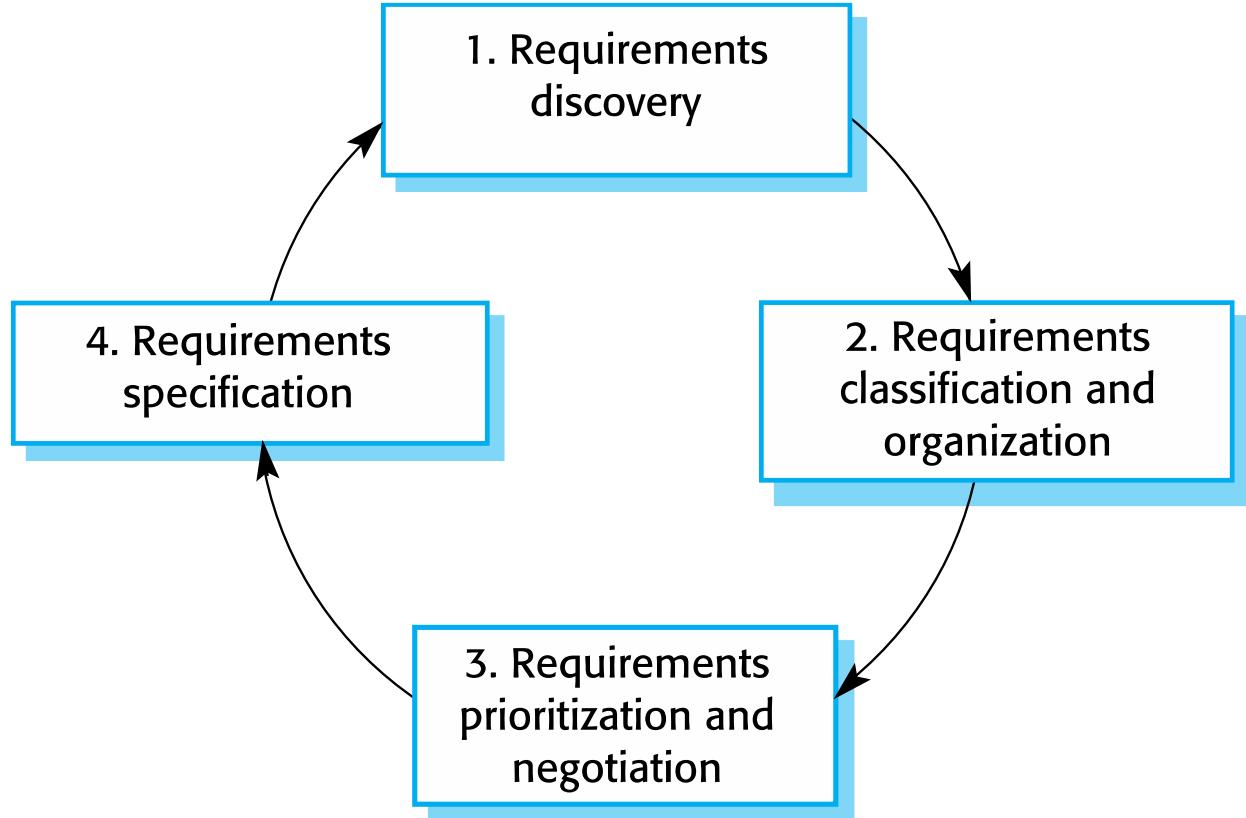
Requirements Elicitation

- Software engineers work with a range of system stakeholders to find out about the application domain, the services that the system should provide, the required system performance, hardware constraints, other systems, etc.
- Stages of **Requirements Elicitation** 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.
- Organisational and political factors may influence the system requirements.
- The requirements change during the analysis process. New stakeholders may emerge and the business environment may change.

The Requirements Elicitation and Analysis Process



Requirements Elicitation Process Activities

- **Requirements discovery**
 - Interacting with stakeholders to discover their requirements. Domain requirements are also discovered at this stage.
- **Requirements classification and organisation**
 - Groups related requirements and organises them into coherent clusters.
- **Prioritisation and negotiation**
 - Prioritising requirements and resolving requirements conflicts.
- **Requirements specification**
 - Requirements are documented and input into the next round of the spiral.

Requirements Discovery

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

Interviewing

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

Interviews in Practice

- Normally a mix of closed and open-ended interviewing.
- Interviews are good for getting an overall understanding of what stakeholders do and how they might interact with the system.
- Interviewers need to be open-minded without pre-conceived ideas of what the system should do
- You need to prompt the user to talk about the system by suggesting requirements rather than simply asking them what they want.

Problems with Interviews

- Application specialists may use language to describe their work that isn't easy for the requirements engineer to understand.
- Interviews are not good for understanding domain requirements
 - Requirements engineers cannot understand specific domain terminology;
 - Some domain knowledge is so familiar that people find it hard to articulate or think that it isn't worth articulating.

Ethnography

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

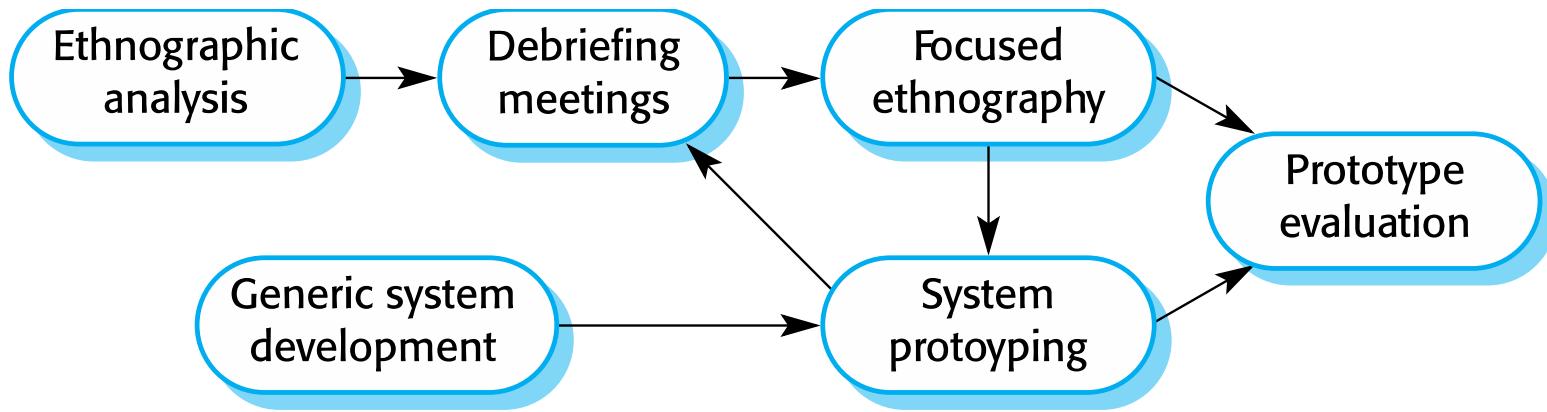
Scope of Ethnography

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

Focused Ethnography

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

Ethnography and Prototyping for Requirements Analysis



Stories and Scenarios

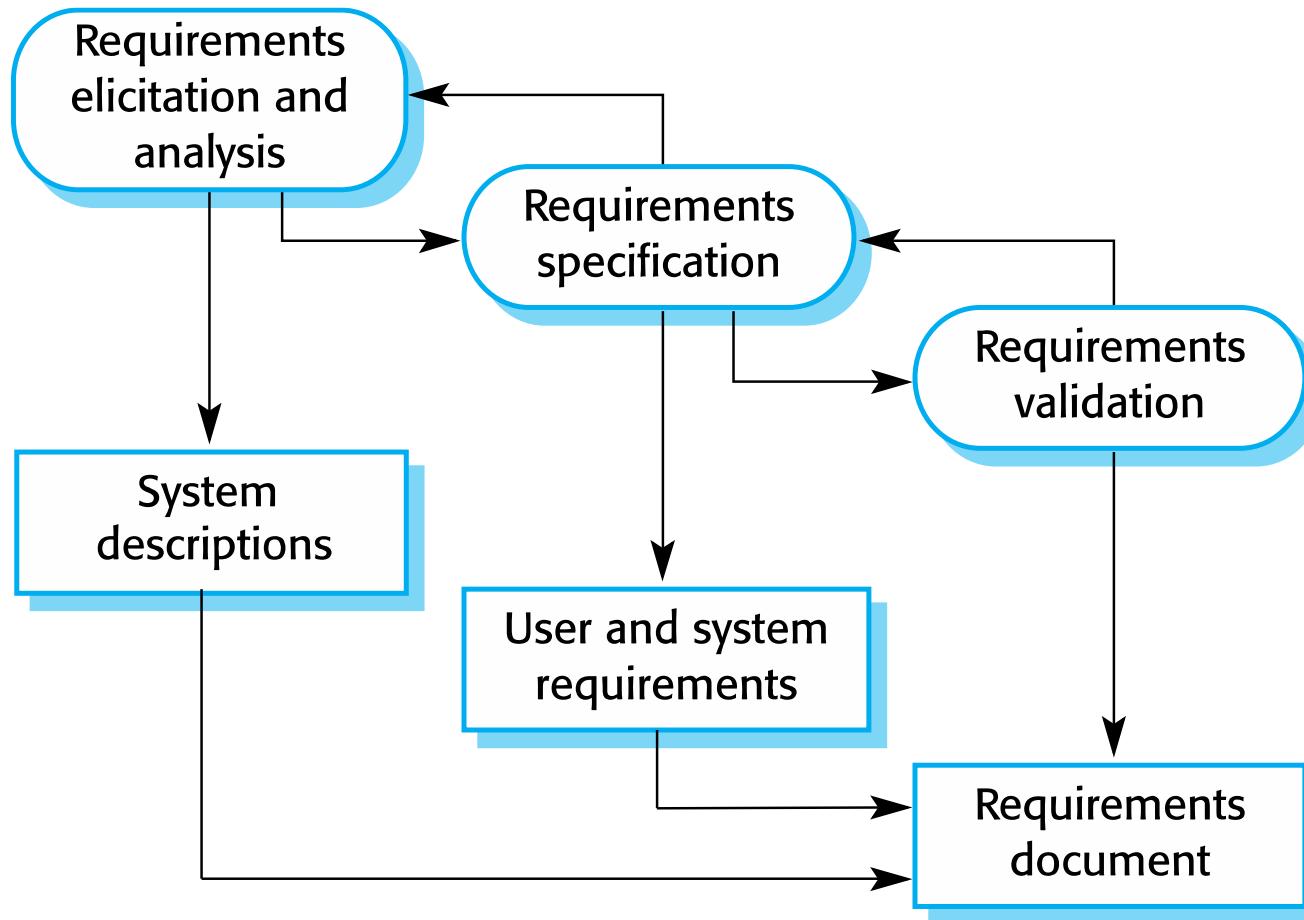
- Scenarios and user stories are real-life examples of how a system can be used.
- Stories and scenarios are a description of how a system may be used for a particular task.
- Because they are based on a practical situation, stakeholders can relate to them and can comment on their situation with respect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

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

The Requirements Engineering Process



Requirements Specification

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

Ways of Writing a System Requirements Specification

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

Requirements and Design

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

Natural Language Specification

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

Guidelines for Writing Requirements

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

Problems with Natural Language

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

Example Requirements for the Insulin Pump Software System

- 3.2 The system shall measure the blood sugar and deliver insulin, if required, every 10 minutes. (Changes in blood sugar are relatively slow so more frequent measurement is unnecessary; less frequent measurement could lead to unnecessarily high sugar levels.)

- 3.6 The system shall run a self-test routine every minute with the conditions to be tested and the associated actions defined in Table 1. (A self-test routine can discover hardware and software problems and alert the user to the fact the normal operation may be impossible.)

Structured Specifications

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

Form-Based Specifications

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

A Structured Specification of a Requirement for an Insulin Pump (1 of 2)

Insulin Pump/Control Software/SRS/3.3.2

Function Compute insulin dose: safe sugar level

Description

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

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

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

Outputs CompDose—the dose in insulin to be delivered.

Destination Main control loop.

A Structured Specification of a Requirement for an Insulin Pump (2 of 2)

Action

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

Requirements

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

Pre-condition

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

Post-condition r_0 is replaced by r_1 then r_1 is replaced by r_2 .

Side effects None.

Tabular Specification

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

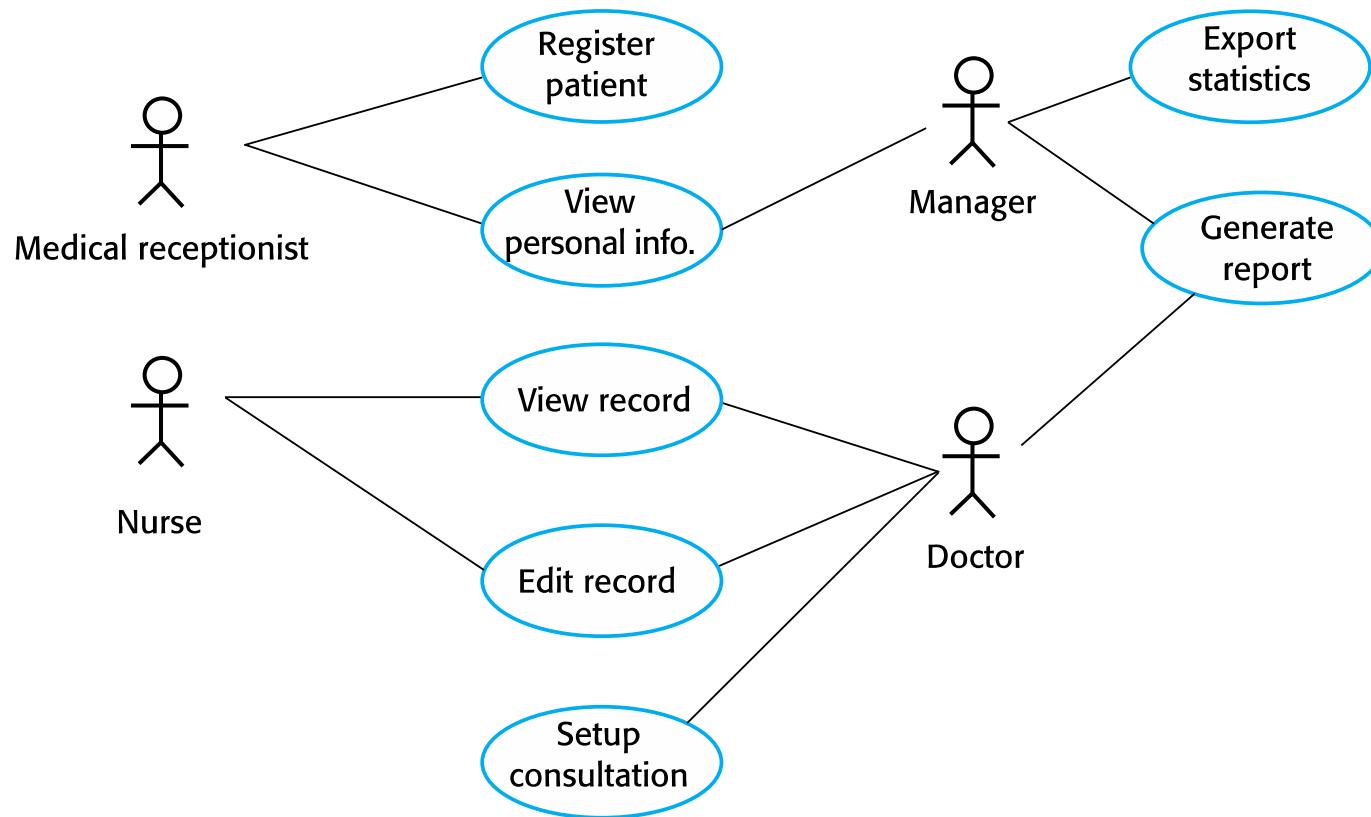
Tabular Specification of Computation for an Insulin Pump

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

Use Cases

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

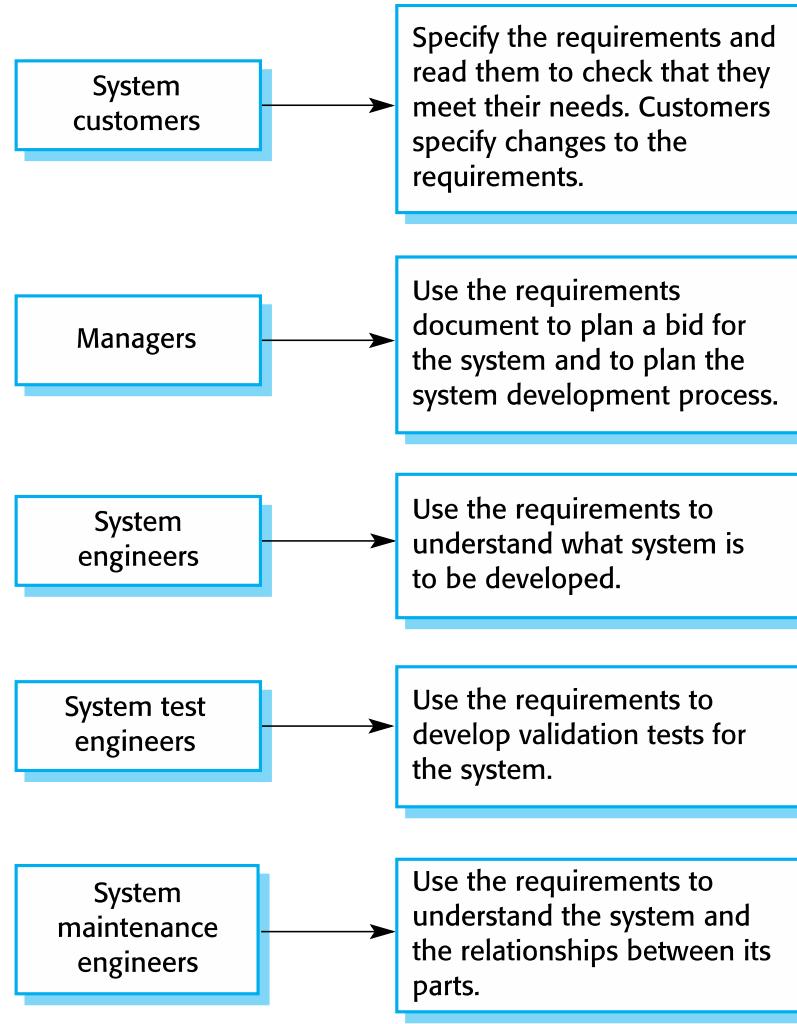
Use Cases for the Mentcare System



The Software Requirements Document

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

Users of a Requirements Document



Requirements Document Variability

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

The Structure of a Requirements Document (1 of 2)

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 (2 of 2)

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 Validation Techniques

- Requirements reviews
 - Systematic manual analysis of the requirements.
- Prototyping
 - Using an executable model of the system to check requirements.
- Test-case generation
 - Developing tests for requirements to check testability.

Requirements Reviews

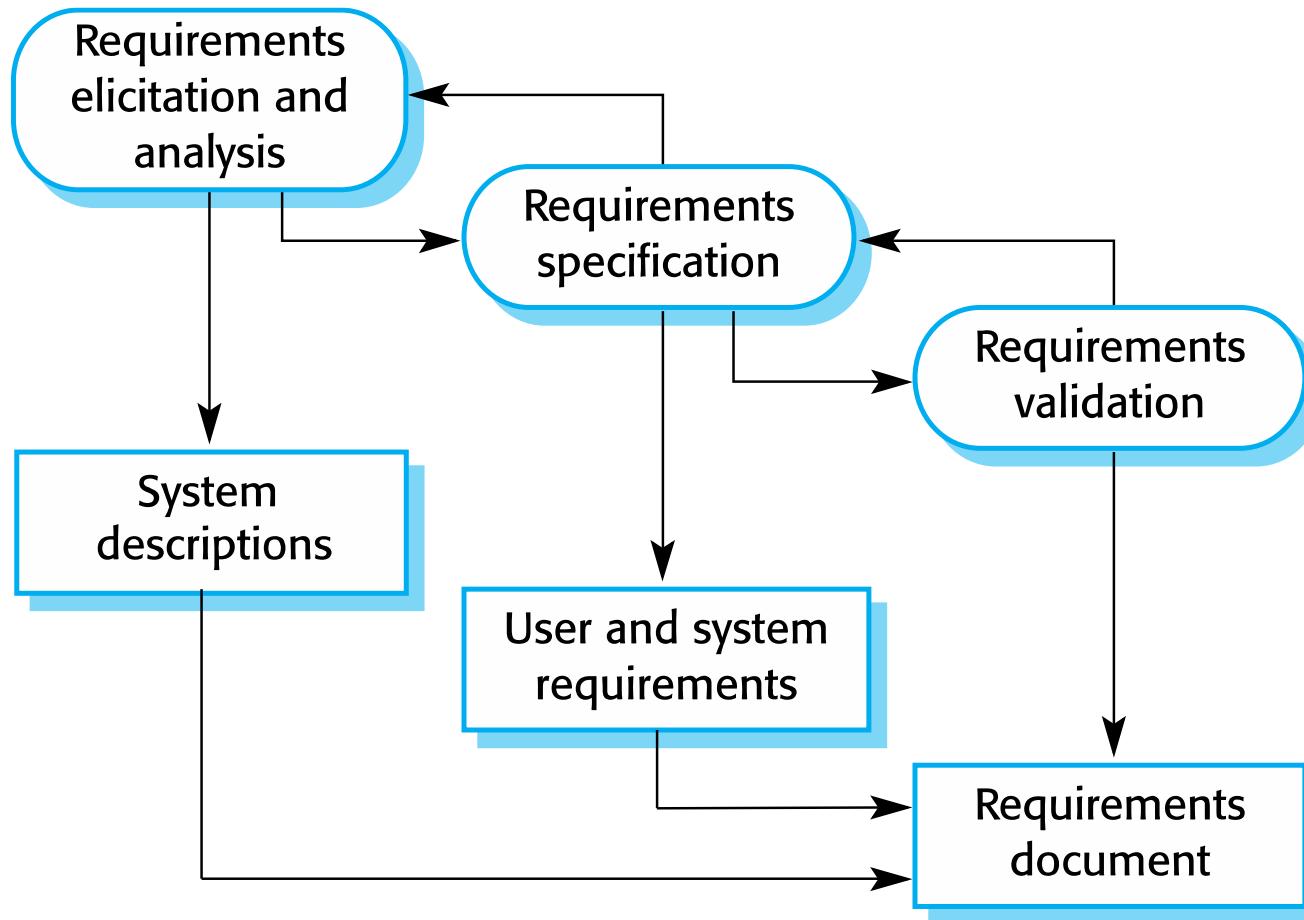
- **Regular reviews** should be held while the requirements definition is being formulated.
- Both client and contractor staff should be involved in reviews.
- **Reviews may be formal** (with completed documents) or **informal**. Good communication between developers, customers and users can resolve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

Review Checks

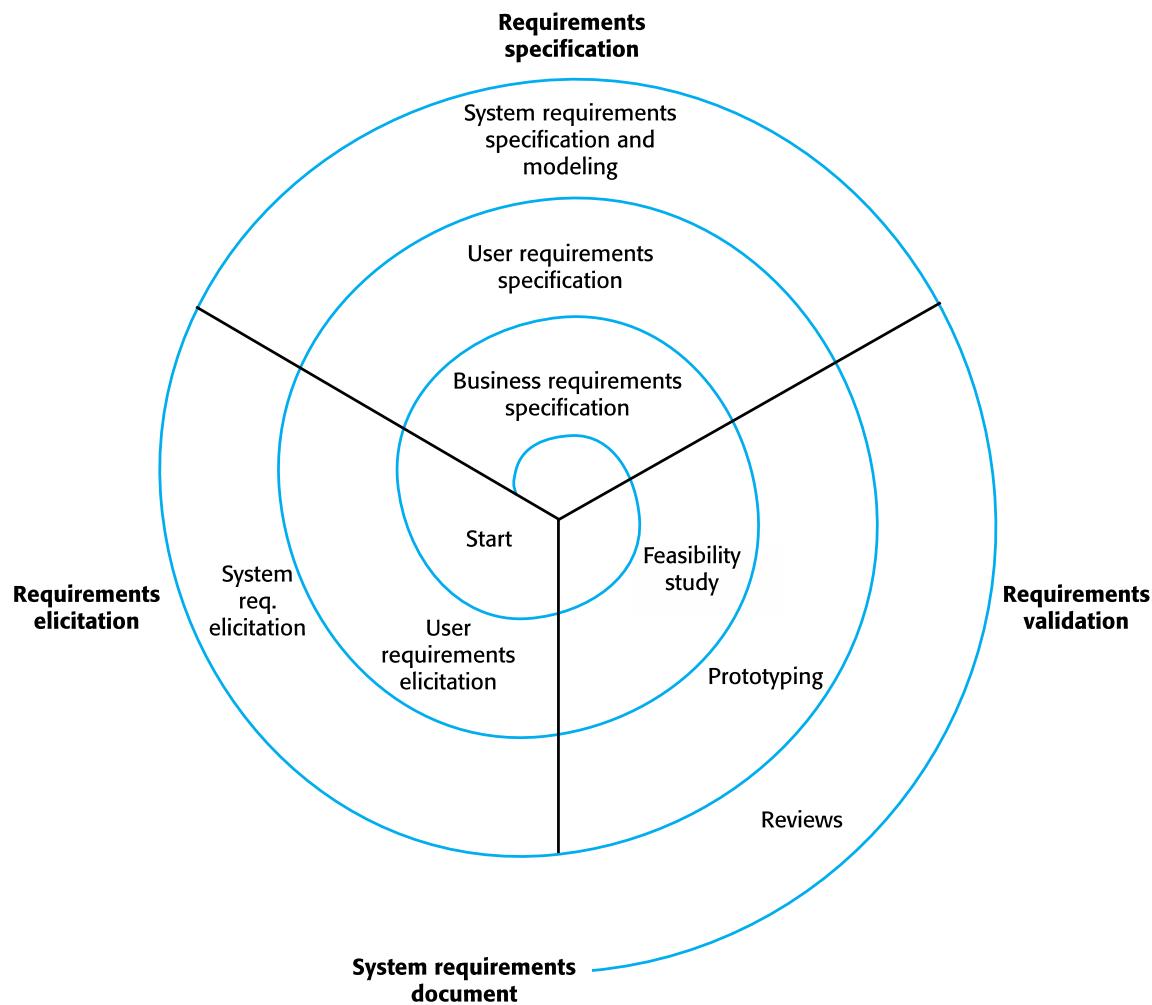
- **Verifiability**
 - Is the requirement realistically testable?
- **Comprehensibility**
 - Is the requirement properly understood?
- **Traceability**
 - Is the origin of the requirement clearly stated?
- **Adaptability**
 - Can the requirement be changed without a large impact on other requirements?

Requirements Change

The Requirements Engineering Process



A Spiral View of the Requirements Engineering Process



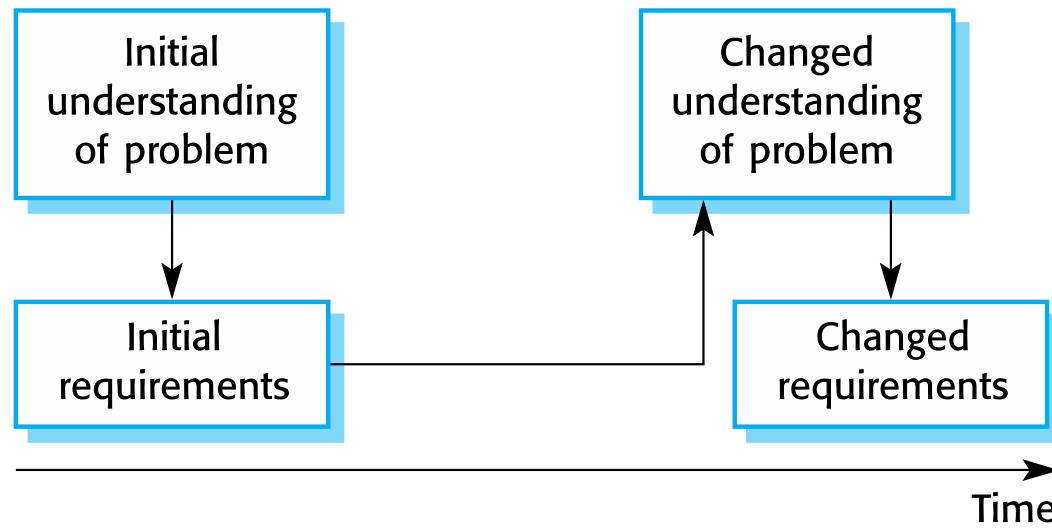
Changing Requirements (1 of 2)

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

Changing Requirements (2 of 2)

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

Requirements Evolution



Requirements Management

- 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.

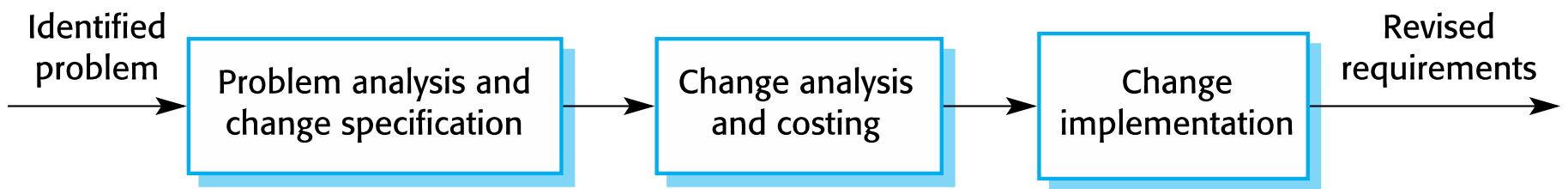
Requirements Management Planning

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

Requirements Change Management (1 of 2)

- Deciding if a requirements change should be accepted
 - **Problem analysis and change specification**
 - During this stage, the problem or the change proposal is analyzed to check that it is valid. This analysis is fed back to the change requestor who may respond with a more specific requirements change proposal, or decide to withdraw the request.
 - **Change analysis and costing**
 - The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements. Once this analysis is completed, a decision is made whether or not to proceed with the requirements change.
 - **Change implementation**
 - The requirements document and, where necessary, the system design and implementation, are modified. Ideally, the document should be organized so that changes can be easily implemented.

Requirements Change Management (2 of 2)



Requirements Engineering: Key Points

Requirements Engineering: Key Points (1 of 4)

- 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 (2 of 4)

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

Key Points (3 of 4)

- **Requirements specification** is the process of formally documenting the user and system requirements and creating a software requirements document.
- The **software requirements specification document** is an agreed statement of the system requirements. It should be organized so that both system customers and software developers can use it.

Key Points (4 of 4)

- 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.



Thank you.



Software Quality Management

Lecture # 5: Software Testing

Software Testing: Topics covered

- Development testing
 - Unit testing
 - Component testing
 - System testing
- Test-driven development
 - Regression testing
- Release testing
- User testing
 - Alpha testing
 - Beta testing
 - Acceptance testing

Program testing

- ✧ **Program 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.
- ✧ Testing can reveal the presence of errors, NOT their absence.
- ✧ Testing is part of a more general **Verification and Validation process**, which also includes **static validation** techniques.

Program testing goals

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

Validation Testing and Defect testing

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

Testing process goals

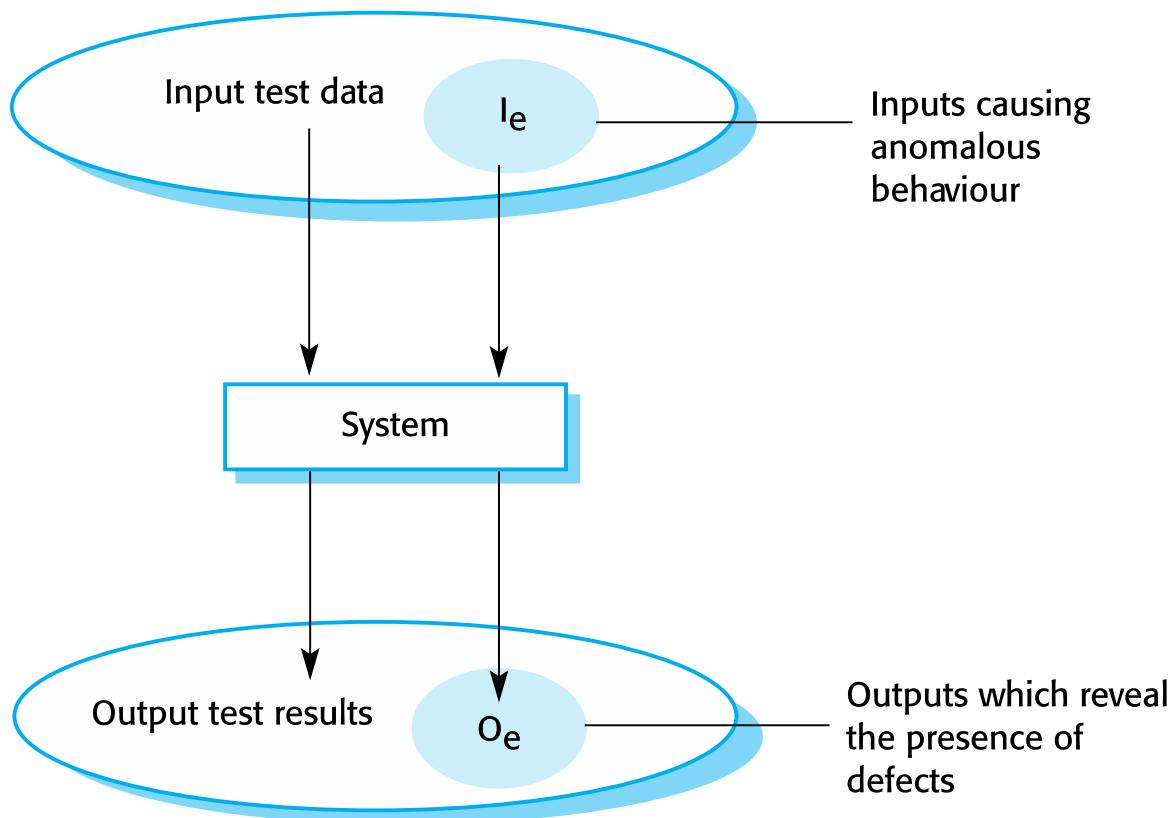
✧ Validation testing

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

✧ Defect testing

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

An input-output model of program testing



Verification vs Validation

✧ **Verification:**

"Are we building the product right?"

✧ The software should conform to its specification.

✧ **Validation:**

"Are we building the right product?"

✧ The software should do what the user really requires.

Verification & Validation confidence

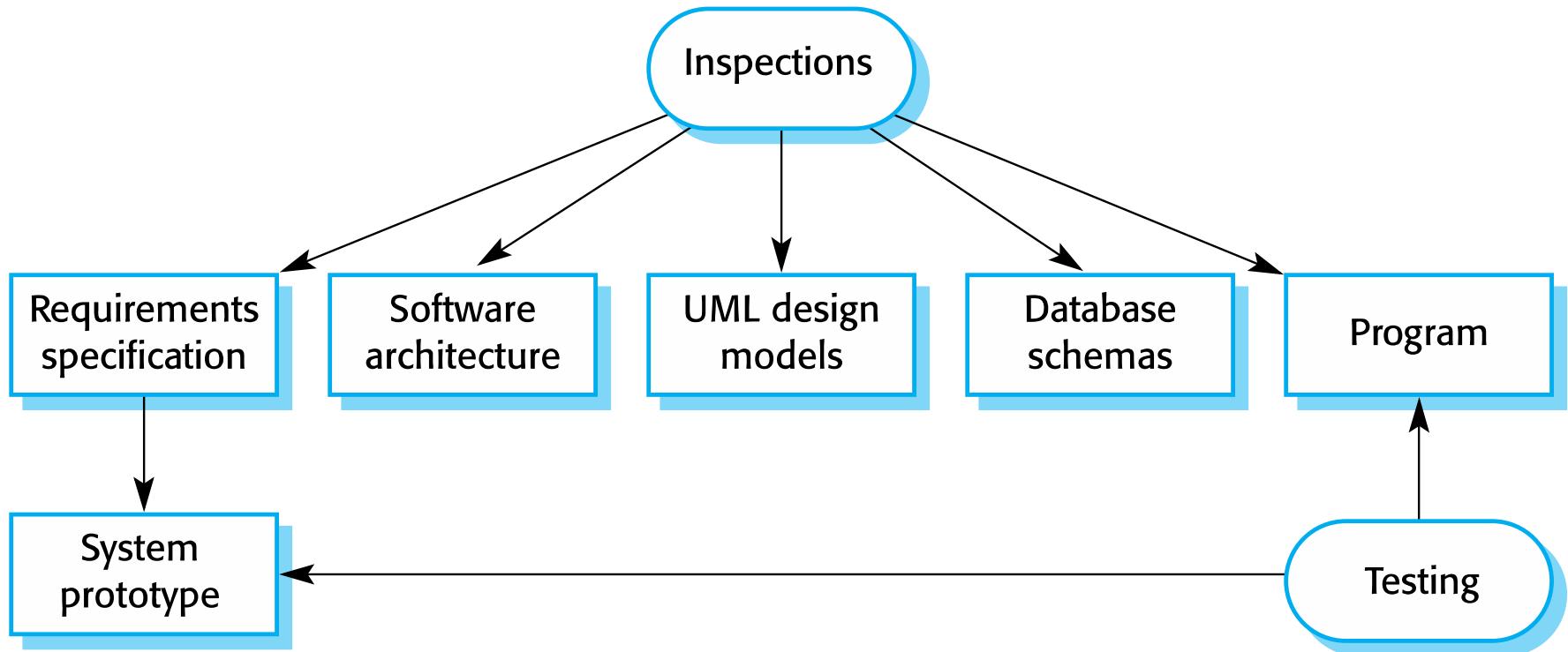
- ✧ Aim of **Verification & Validation (V&V)** is to establish confidence that the system is ‘fit for purpose’.
- ✧ Depends on system’s purpose, user expectations and marketing environment
 - **Software purpose**
 - The level of confidence depends on how critical the software is to an organisation.
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

Inspections and Testing

- ❖ **Software Inspections:** Concerned with analysis of the static system representation to discover problems (*static verification*)
 - May be supplement by tool-based document and code analysis.

- ❖ **Software Testing:** Concerned with exercising and observing product behaviour (*dynamic verification*)
 - The system is executed with test data and its operational behaviour is observed.

Inspections and Testing



Software inspections

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

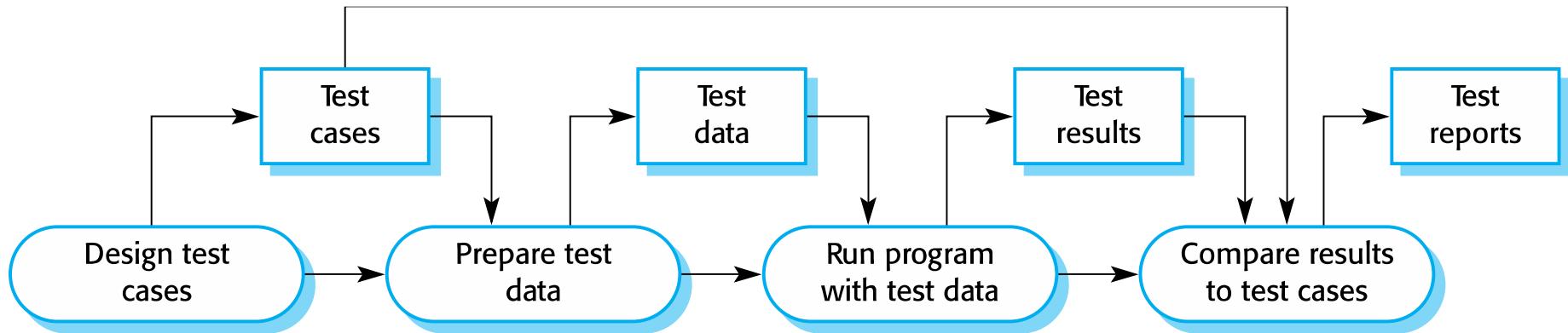
Advantages of inspections

- ✧ 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.
- ✧ In addition to 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 Program Testing

- ✧ Inspections and Program 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, reliability, usability, etc.

A model of the Software Testing process



Stages of Program Testing

1. **Development testing**, where the system is tested during development to discover bugs and defects.
2. **Release testing**, where a separate testing team test a complete version of the system before it is released to users.
3. **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.

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 cases:
 - 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.

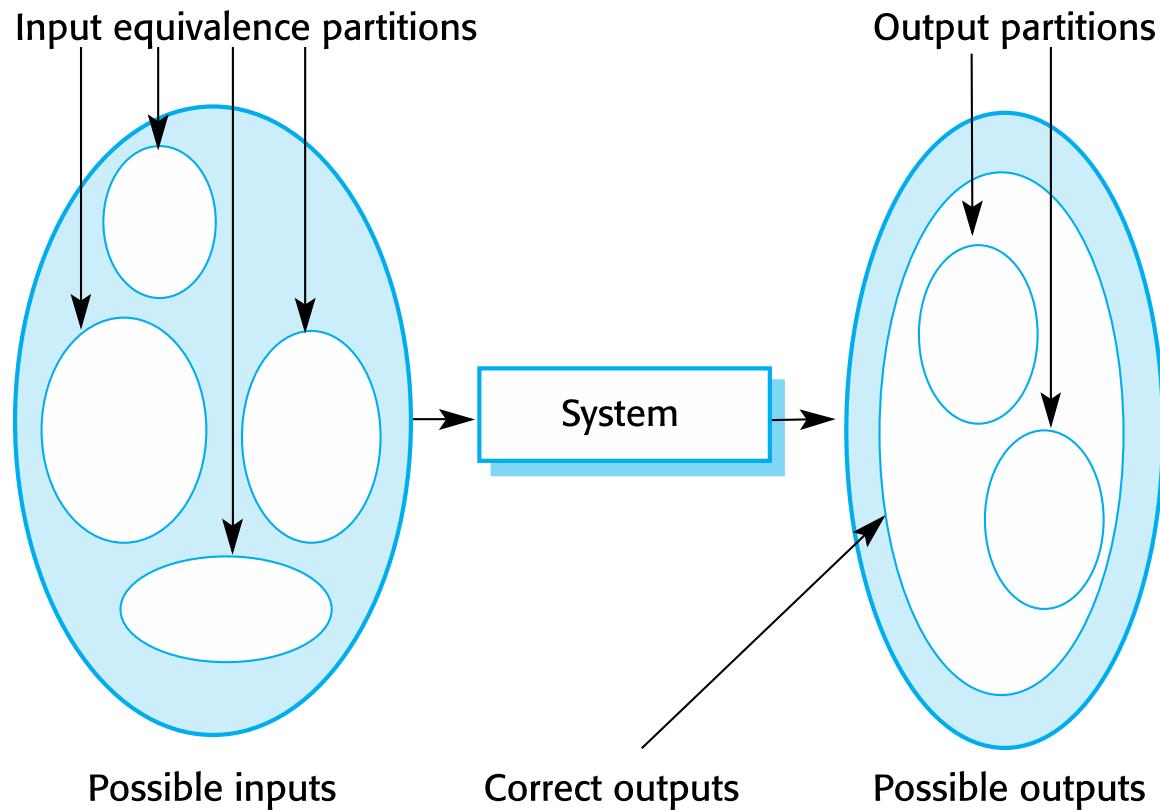
Testing strategies

- ✧ **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.

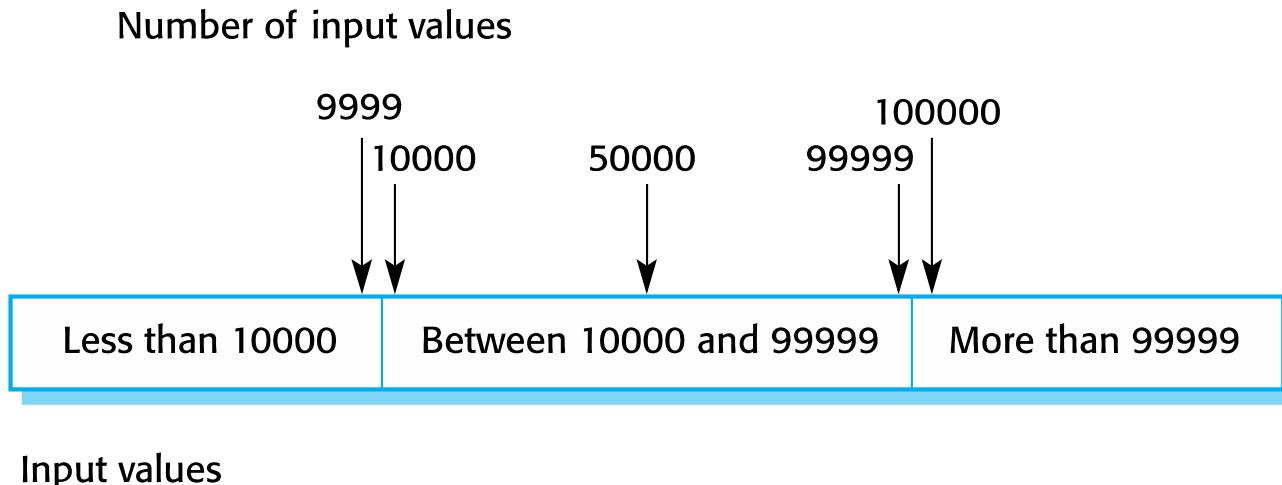
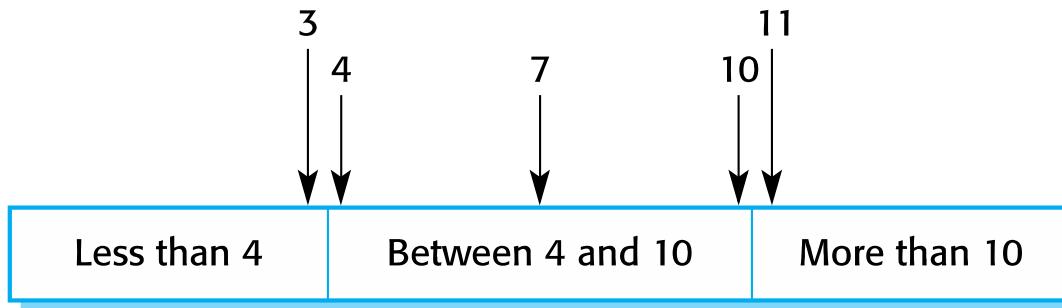
Partition testing

- ✧ 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.
- ✧ 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 guidelines

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

System testing

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

System Testing and Component Testing

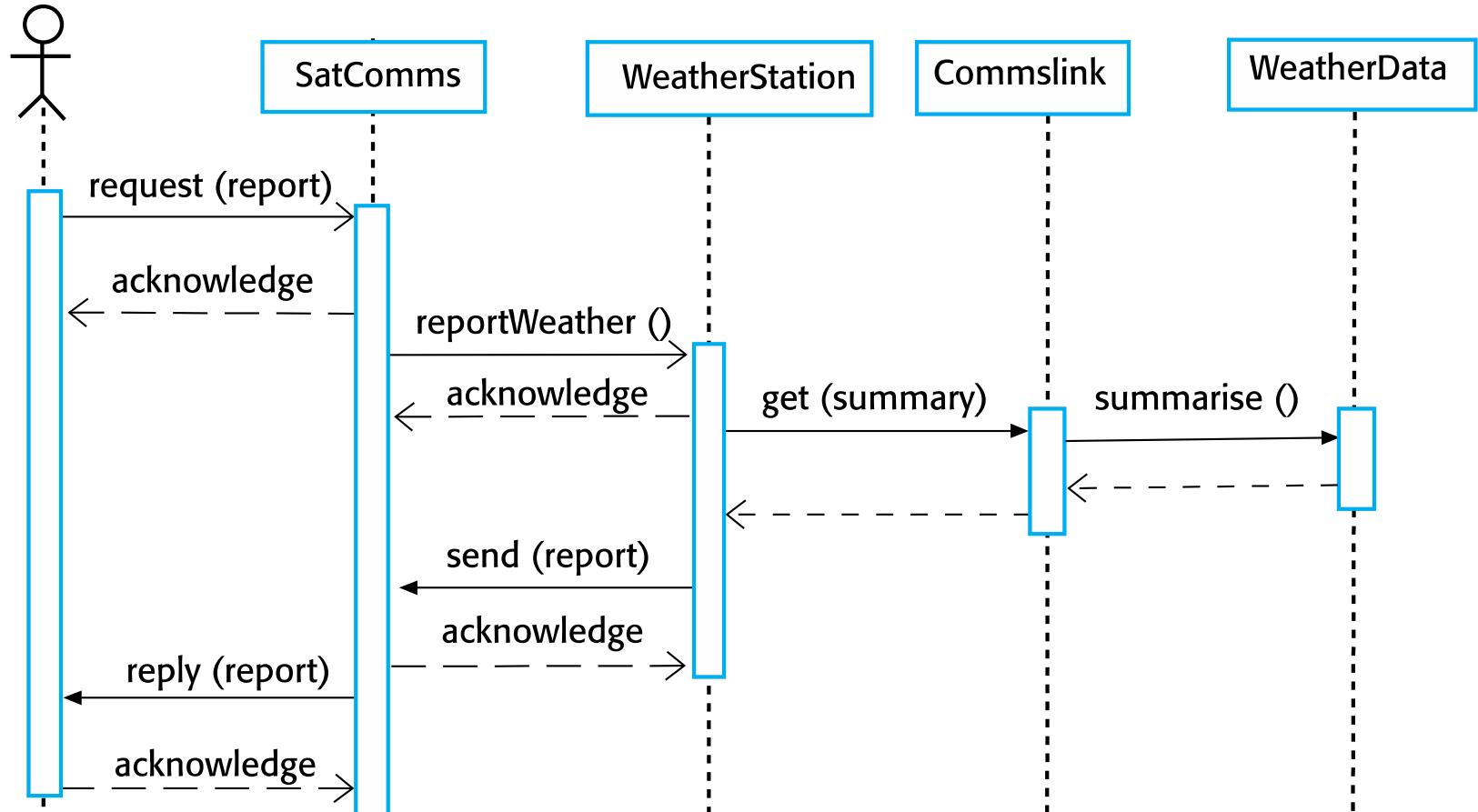
- ✧ During system testing, reusable components that have been separately developed and off-the-shelf systems may be integrated with newly developed components. The complete system is then tested.
- ✧ Components developed by different team members or sub-teams may be integrated at this stage. System testing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

Use-case testing

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

Collect weather data sequence chart

information system



Test cases derived from the sequence diagram

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

Testing policies

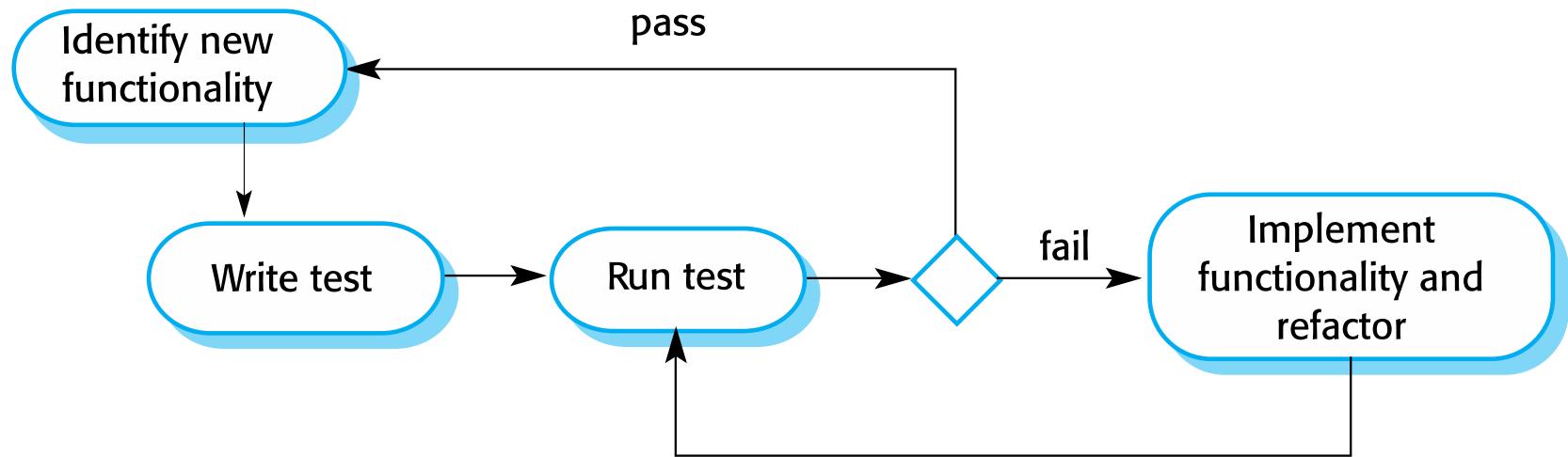
- ✧ **Exhaustive system testing is impossible** so testing policies which define the required system test coverage may be developed.
- ✧ Examples of testing policies:
 - All system functions that are accessed through menus should be tested.
 - Combinations of functions (e.g. text formatting) that are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

Test-driven development

Test-driven development

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

Test-driven development



TDD process activities

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

Benefits of Test-Driven Development

✧ Code coverage

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

✧ Regression testing

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

✧ Simplified debugging

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

✧ System documentation

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

Regression testing

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

Release testing

Release testing

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

Release testing and system testing

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

Requirements-based testing

- ✧ Requirements-based testing involves examining each requirement and developing a test or tests for it.
- ✧ Example: 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.

Performance testing

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

User testing

User testing

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

Types of user testing

✧ **Alpha testing**

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

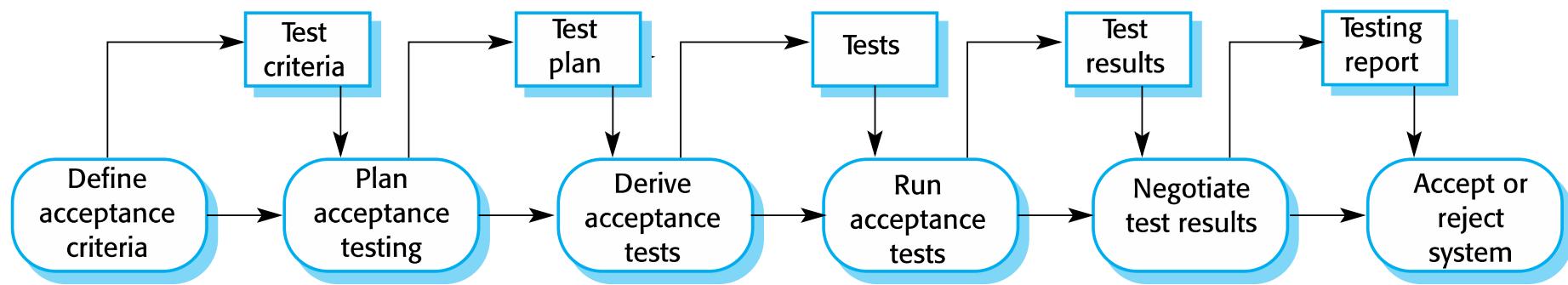
✧ **Beta testing**

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

✧ **Acceptance testing**

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

The Acceptance Testing process



Stages in the Acceptance Testing process

1. Define acceptance criteria
2. Plan acceptance testing
3. Derive acceptance tests
4. Run acceptance tests
5. Negotiate test results
6. Reject/accept system

Agile methods and Acceptance Testing

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

Software Testing: 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 (2 / 2)

- ✧ 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-driven 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.

Thank you.



Software Quality Management

Lecture # 5: Software Testing

Software Testing: Topics covered

- Development testing
 - Unit testing
 - Component testing
 - System testing
- Test-driven development
 - Regression testing
- Release testing
- User testing
 - Alpha testing
 - Beta testing
 - Acceptance testing

Program testing

- ✧ **Program 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.
- ✧ Testing can reveal the presence of errors, NOT their absence.
- ✧ Testing is part of a more general **Verification and Validation process**, which also includes **static validation** techniques.

Program testing goals

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

Validation Testing and Defect testing

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

Testing process goals

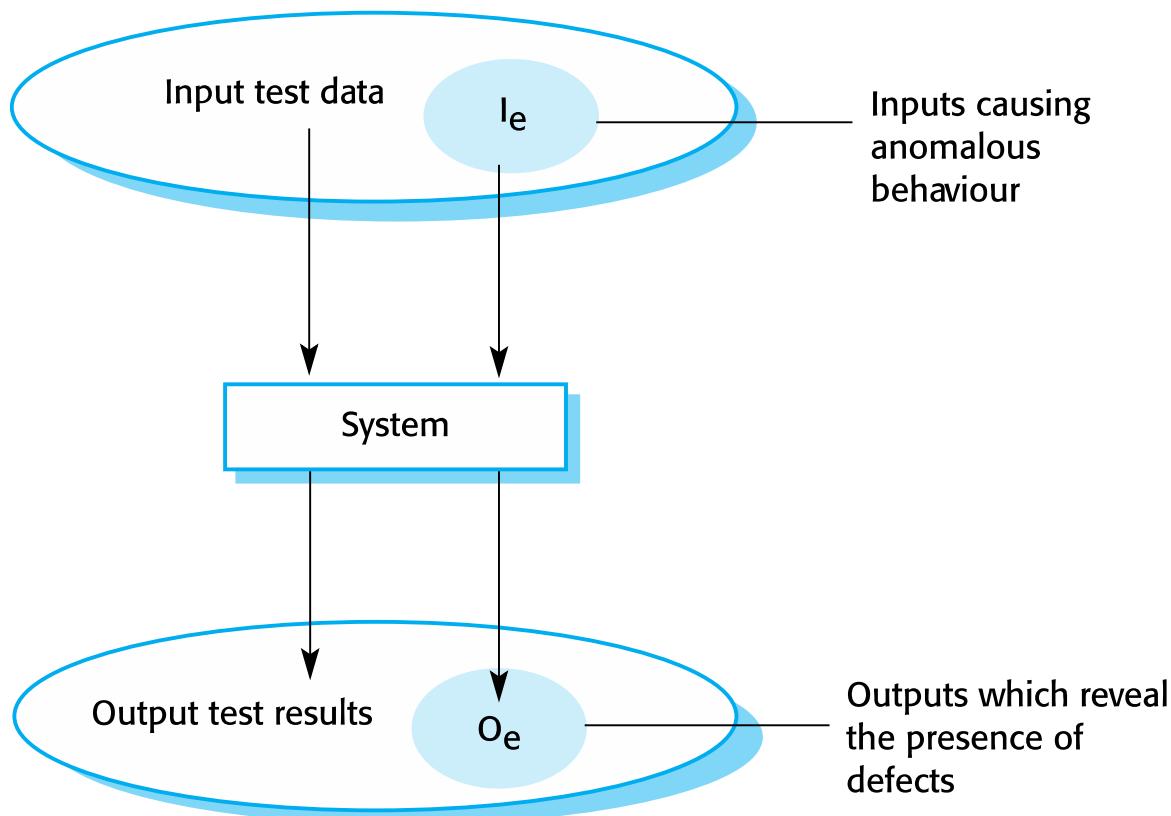
✧ Validation testing

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

✧ Defect testing

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

An input-output model of program testing



Verification vs Validation

✧ **Verification:**

"Are we building the product right?"

✧ The software should conform to its specification.

✧ **Validation:**

"Are we building the right product?"

✧ The software should do what the user really requires.

Verification & Validation confidence

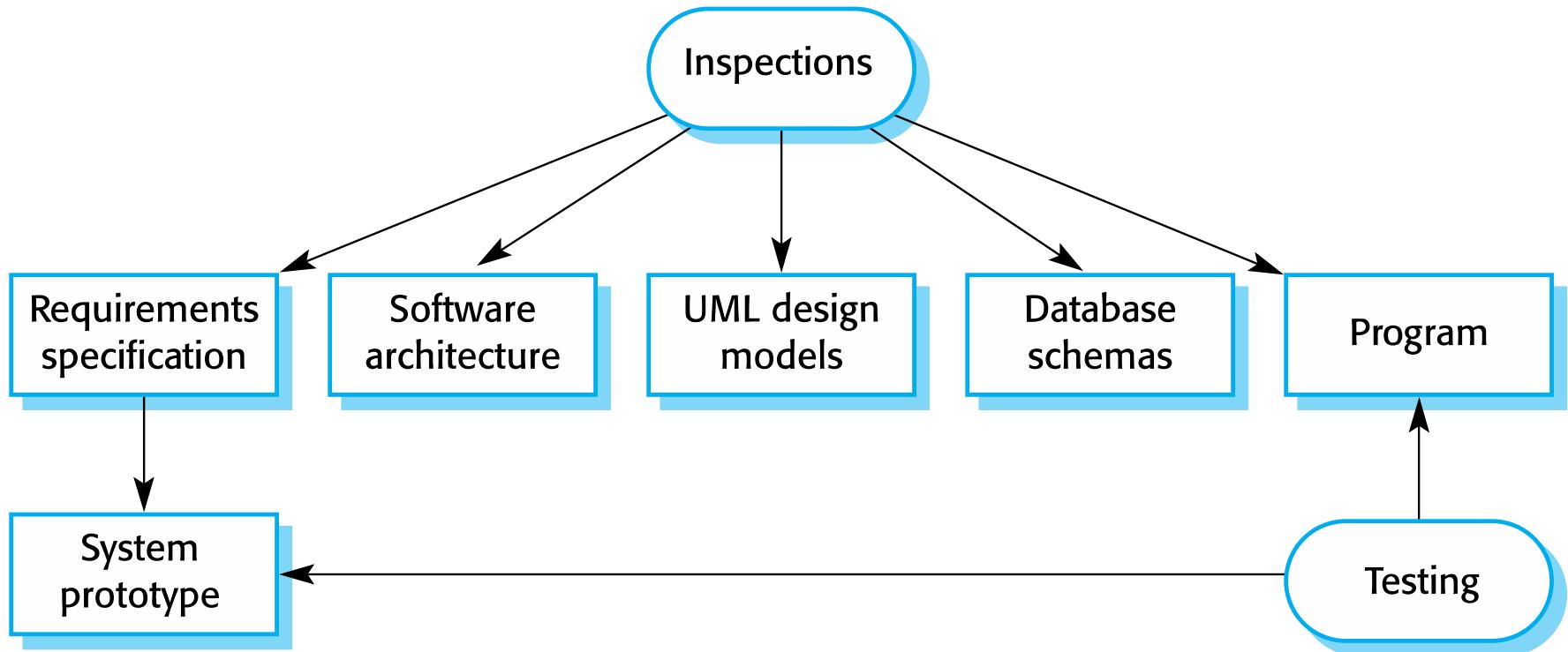
- ✧ Aim of **Verification & Validation (V&V)** is to establish confidence that the system is ‘fit for purpose’.
- ✧ Depends on system’s purpose, user expectations and marketing environment
 - **Software purpose**
 - The level of confidence depends on how critical the software is to an organisation.
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

Inspections and Testing

- ❖ **Software Inspections:** Concerned with analysis of the static system representation to discover problems (*static verification*)
 - May be supplement by tool-based document and code analysis.

- ❖ **Software Testing:** Concerned with exercising and observing product behaviour (*dynamic verification*)
 - The system is executed with test data and its operational behaviour is observed.

Inspections and Testing



Software inspections

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

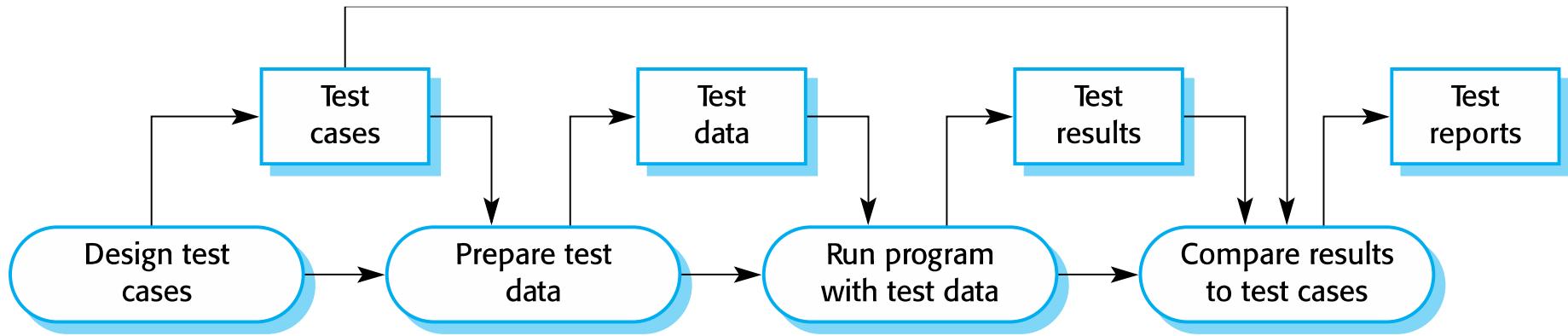
Advantages of inspections

- ✧ 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.
- ✧ In addition to 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 Program Testing

- ✧ Inspections and Program 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, reliability, usability, etc.

A model of the Software Testing process



Stages of Program Testing

1. **Development testing**, where the system is tested during development to discover bugs and defects.
2. **Release testing**, where a separate testing team test a complete version of the system before it is released to users.
3. **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.

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 cases:
 - 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.

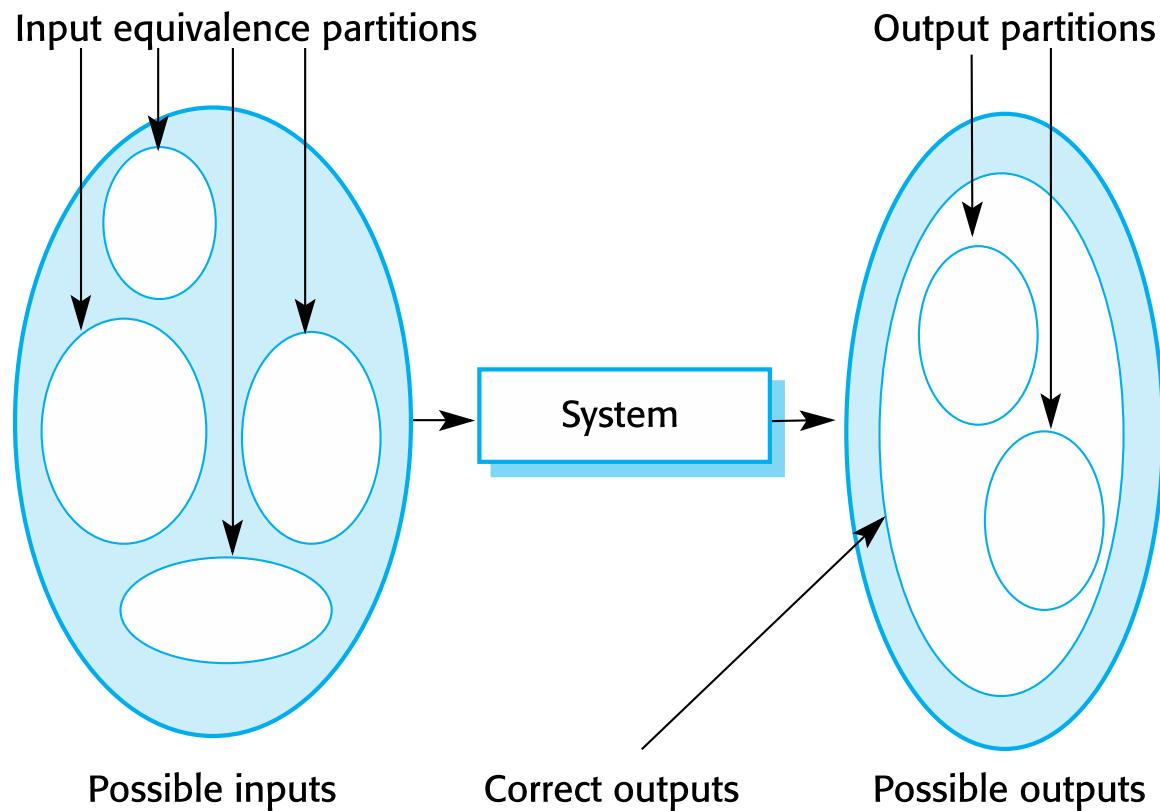
Testing strategies

- ✧ **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.

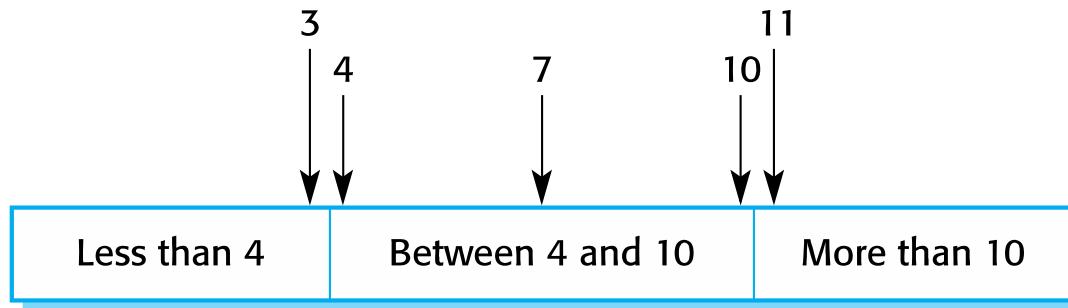
Partition testing

- ✧ 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.

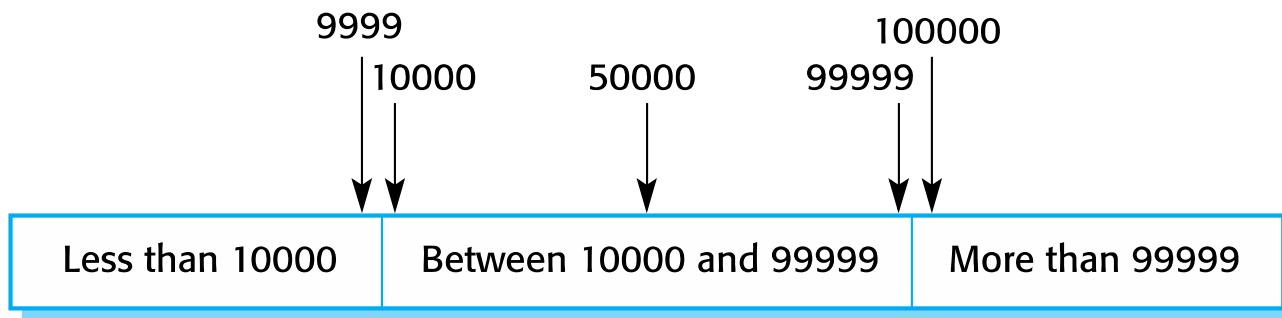
Equivalence partitioning



Equivalence partitions



Number of input values



Input values

Testing guidelines (sequences)

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

General testing guidelines

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

Component testing

- ✧ Software components are often composite components that are made up of several interacting objects.
- ✧ 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 guidelines

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

System testing

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

System Testing and Component Testing

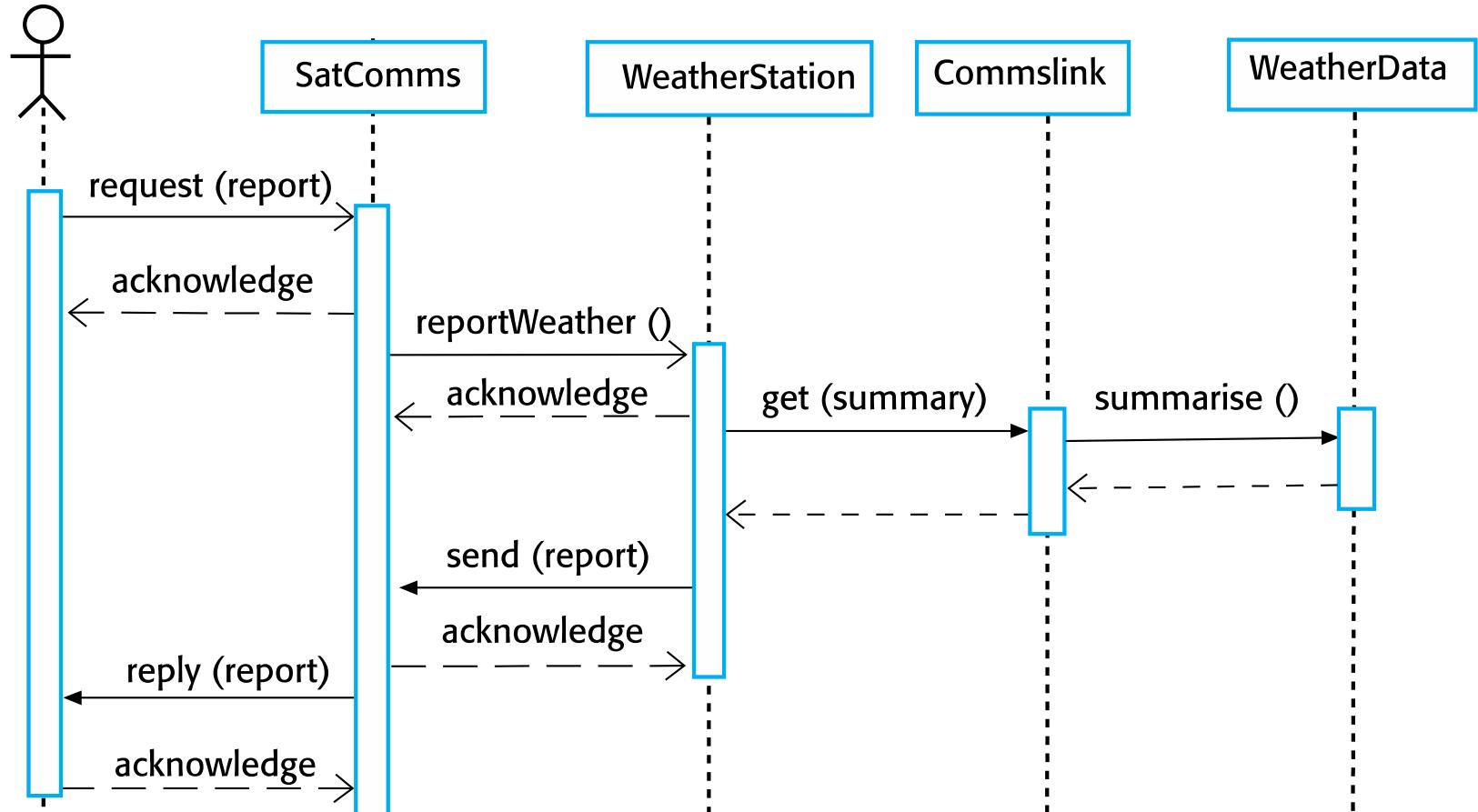
- ✧ During system testing, reusable components that have been separately developed and off-the-shelf systems may be integrated with newly developed components. The complete system is then tested.
- ✧ Components developed by different team members or sub-teams may be integrated at this stage. System testing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

Use-case testing

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

Collect weather data sequence chart

information system



Test cases derived from the sequence diagram

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

Testing policies

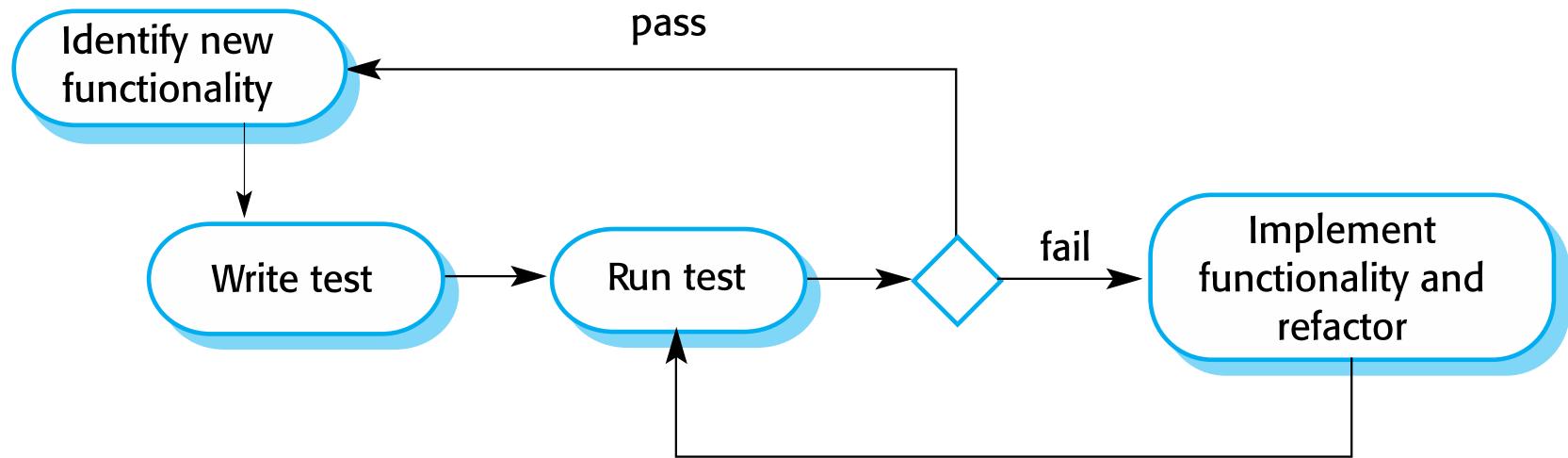
- ✧ **Exhaustive system testing is impossible** so testing policies which define the required system test coverage may be developed.
- ✧ Examples of testing policies:
 - All system functions that are accessed through menus should be tested.
 - Combinations of functions (e.g. text formatting) that are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

Test-driven development

Test-driven development

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

Test-driven development



TDD process activities

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

Benefits of Test-Driven Development

✧ Code coverage

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

✧ Regression testing

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

✧ Simplified debugging

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

✧ System documentation

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

Regression testing

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

Release testing

Release testing

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

Release testing and system testing

✧ Release testing is a form of system testing.

✧ Important differences:

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

Requirements-based testing

- ✧ Requirements-based testing involves examining each requirement and developing a test or tests for it.
- ✧ Example: 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.

Performance testing

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

User testing

User testing

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

Types of user testing

✧ **Alpha testing**

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

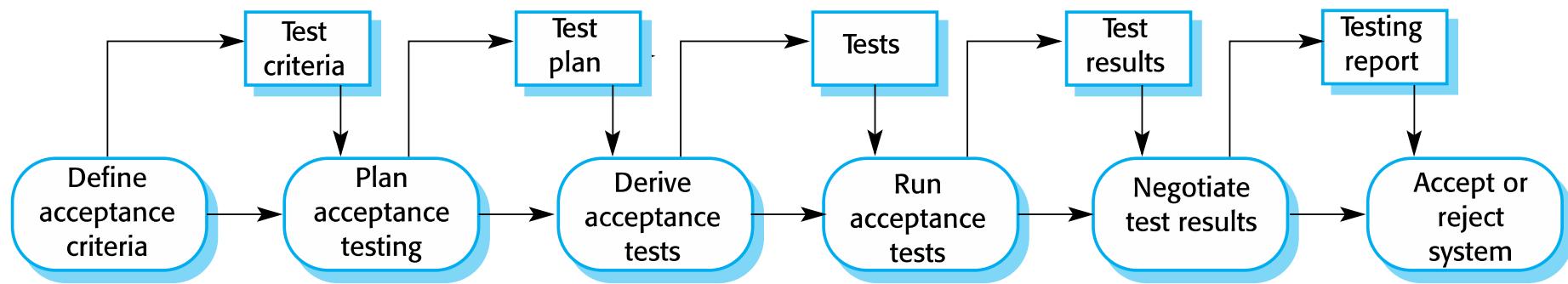
✧ **Beta testing**

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

✧ **Acceptance testing**

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

The Acceptance Testing process



Stages in the Acceptance Testing process

1. Define acceptance criteria
2. Plan acceptance testing
3. Derive acceptance tests
4. Run acceptance tests
5. Negotiate test results
6. Reject/accept system

Agile methods and Acceptance Testing

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

Software Testing: 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 (2 / 2)

- ✧ 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-driven 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.

Thank you.



Software Quality Management

Lecture # 6: Quality Management

Quality Management: Topics covered

1. Software Quality
2. Software Standards
3. Reviews and Inspections
4. Quality Management and Agile Development
5. Software Measurement

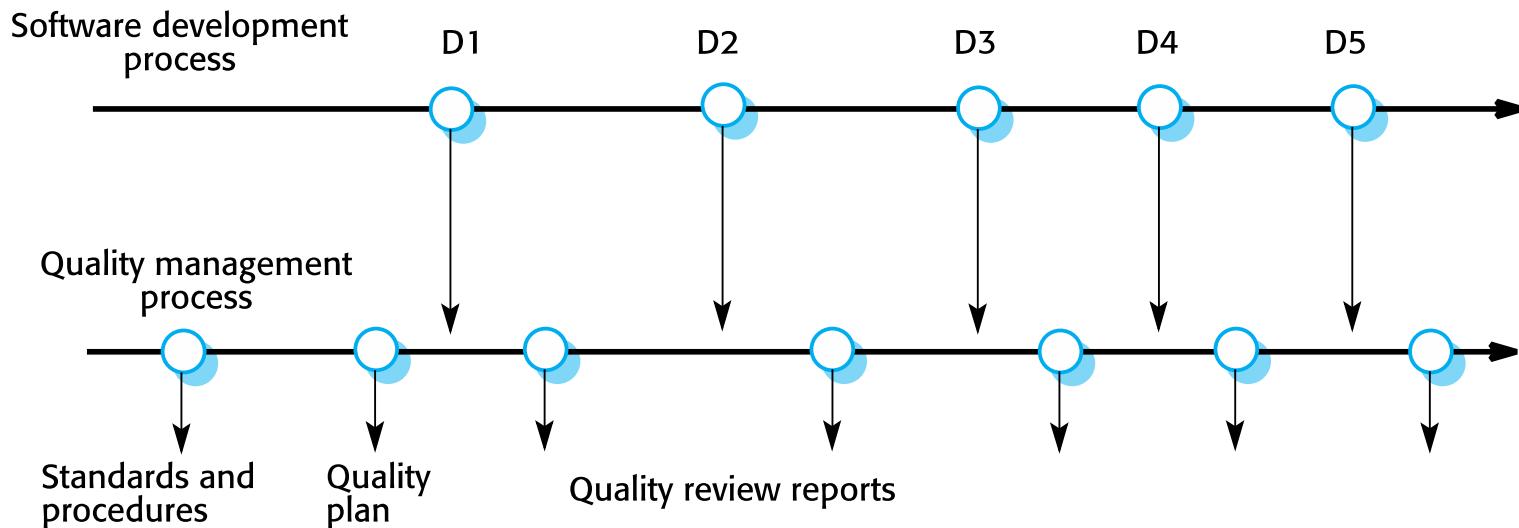
Software Quality Management

- ✧ Concerned with ensuring that the required level of quality is achieved in a software product.
- ✧ Three principal concerns:
 1. At the organizational level, quality management is concerned with establishing a framework of organizational processes and standards that will lead to high-quality software.
 2. At the project level, quality management involves the application of specific quality processes and checking that these planned processes have been followed.
 3. At the project level, quality management is also concerned with establishing a quality plan for a project. The quality plan should set out the quality goals for the project and define what processes and standards are to be used.

Quality Management activities

- ✧ Quality Management provides an independent check on the software development process.
- ✧ The Quality Management process checks the project deliverables to ensure that they are consistent with organizational standards and goals
- ✧ The Quality team should be independent from the development team so that they can take an objective view of the software. This allows them to report on software quality without being influenced by software development issues.

Quality Management and Software Development



Quality Planning

- ✧ A **Quality Plan** sets out the desired product qualities and how these are assessed and defines the most significant quality attributes.
- ✧ The **Quality Plan** should define the quality assessment process.
- ✧ It should set out which organizational standards should be applied and, where necessary, define new standards to be used.

Quality plans

✧ **Quality Plan** structure

- Product introduction;
- Product plans;
- Process descriptions;
- Quality goals;
- Risks and risk management.

✧ **Quality Plans** should be short, succinct documents

- If they are too long, no-one will read them.

Scope of Quality Management

- ✧ Quality Management is particularly important for large, complex systems. The quality documentation is a record of progress and supports continuity of development as the development team changes.
- ✧ For smaller systems, Quality Management needs less documentation and should focus on establishing a quality culture.
- ✧ Techniques have to evolve when agile development is used.

Software Quality

Software Quality

- ✧ Quality, simplistically, means that a product should meet its specification.
- ✧ This is problematical for software systems:
 - There is a tension between customer quality requirements (efficiency, reliability, etc.) and developer quality requirements (maintainability, reusability, etc.);
 - Some quality requirements are difficult to specify in an unambiguous way;
 - Software specifications are usually incomplete and often inconsistent.
- ✧ The **focus** may be ‘fitness for purpose’ rather than specification conformance.

Software fitness for purpose

1. Has the software been properly tested?
2. Is the software sufficiently dependable to be put into use?
3. Is the performance of the software acceptable for normal use?
4. Is the software usable?
5. Is the software well-structured and understandable?
6. Have programming and documentation standards been followed in the development process?

Non-functional characteristics

- ✧ The subjective quality of a software system is largely based on its non-functional characteristics.
- ✧ This reflects practical user experience – if the software's functionality is not what is expected, then users will often just work around this and find other ways to do what they want to do.
- ✧ However, if the software is unreliable or too slow, then it is practically impossible for them to achieve their goals.

Software Quality attributes

Safety	Understandability	Portability
Security	Testability	Usability
Reliability	Adaptability	Reusability
Resilience	Modularity	Efficiency
Robustness	Complexity	Learnability

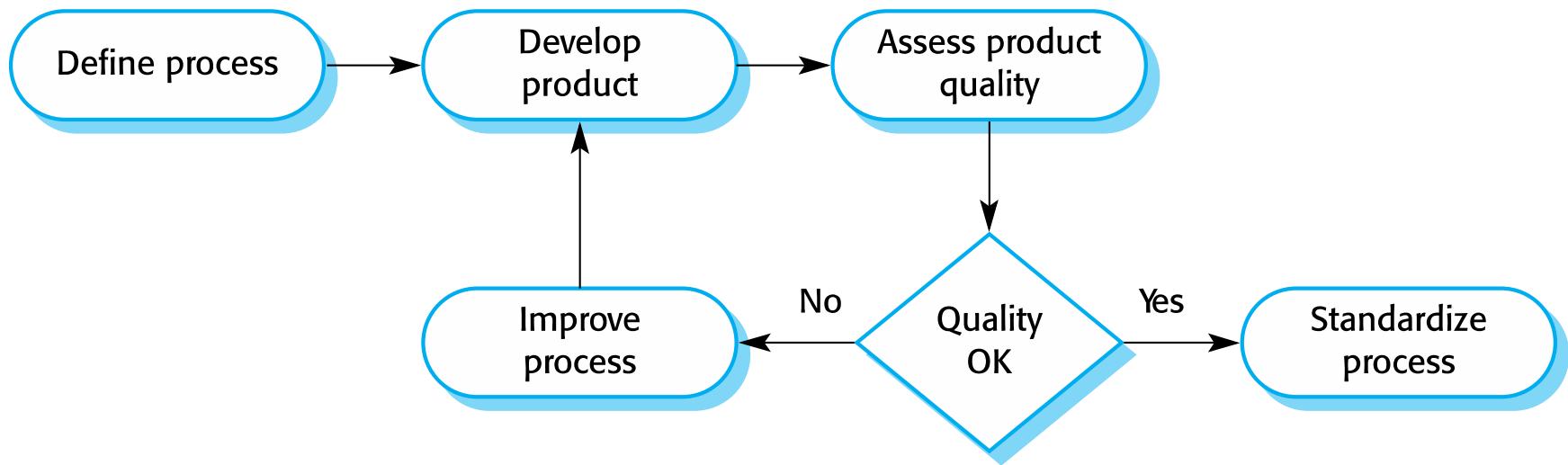
Quality conflicts

- ✧ It is not possible for any system to be optimized for all of these attributes – for example, improving robustness may lead to loss of performance.
- ✧ The **Quality Plan** should therefore define the **most important quality attributes** for the software that is being developed.
- ✧ The plan should also include a **definition of the quality assessment process**, an agreed way of assessing whether some quality, such as maintainability or robustness, is present in the product.

Process Quality and Product Quality

- ✧ The quality of a developed product is influenced by the quality of the production process.
- ✧ This is important in software development as some product quality attributes are hard to assess.
- ✧ However, there is a very complex and poorly understood relationship between software processes and product quality.
 - The application of individual skills and experience is particularly important in software development;
 - External factors such as the novelty of an application or the need for an accelerated development schedule may impair product quality.

Process-based quality



Quality Culture

- ✧ Quality managers should aim to develop a '**Quality Culture**' where everyone responsible for software development is committed to achieving a high level of product quality.
- ✧ They should encourage teams to take responsibility for the quality of their work and to develop new approaches to quality improvement.
- ✧ They should support people who are interested in the intangible aspects of quality and encourage professional behavior in all team members.

Software Standards

Software Standards

- ✧ **Standards** define the required attributes of a product or process. They play an important role in quality management.
- ✧ Standards may be international, national, organizational, or project standards.

Importance of Standards

- ✧ Encapsulation of best practices - avoids repetition of past mistakes.
- ✧ They are a framework for defining what quality means in a particular setting i.e. that organization's view of quality.
- ✧ They provide continuity - new staff can understand the organisation by understanding the standards that are used.

Product Standards and Process Standards

✧ Product Standards

- **Apply to the software product being developed.** They include document standards, such as the structure of requirements documents, documentation standards, such as a standard comment header for an object class definition, and coding standards, which define how a programming language should be used.

✧ Process standards

- **These define the processes that should be followed during software development.** Process standards may include definitions of specification, design and validation processes, process support tools and a description of the documents that should be written during these processes.

Product Standards and Process Standards

Product Standards	Process Standards
Design review form	Design review conduct
Requirements document structure	Submission of new code for system building
Method header format	Version release process
Java programming style	Project plan approval process
Project plan format	Change control process
Change request form	Test recording process

Problems with Standards

- ✧ They may not be seen as relevant and up-to-date by software engineers.
- ✧ They often involve too much bureaucratic form filling.
- ✧ If they are unsupported by software tools, tedious form filling work is often involved to maintain the documentation associated with the standards.

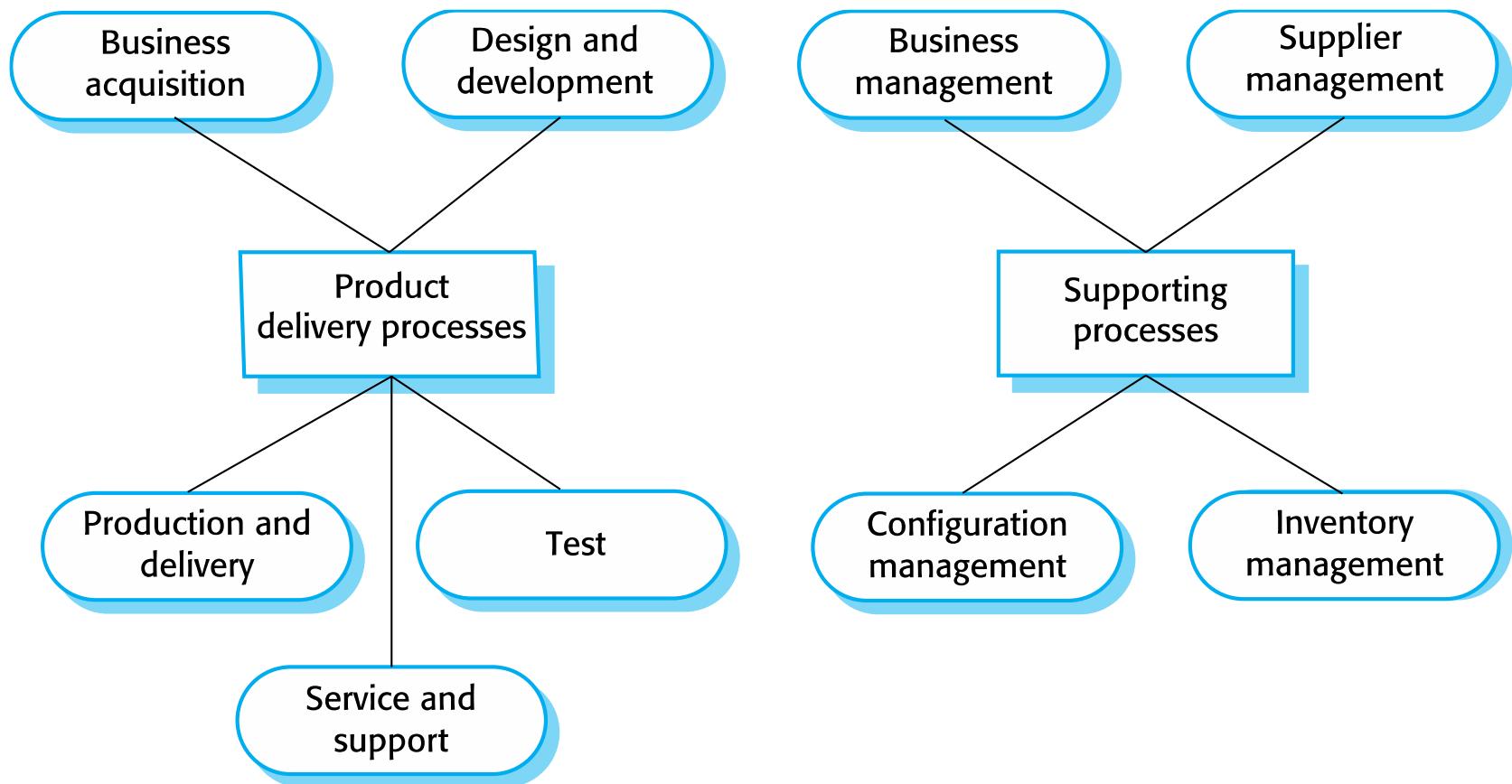
Standards development

- ✧ **Involve practitioners in development.** Engineers should understand the rationale underlying a standard.
- ✧ **Review standards and their usage regularly.**
Standards can quickly become outdated and this reduces their credibility amongst practitioners.
- ✧ **Detailed standards should have specialized tool support.** Excessive clerical work is the most significant complaint against standards.
 - Web-based forms are not good enough.

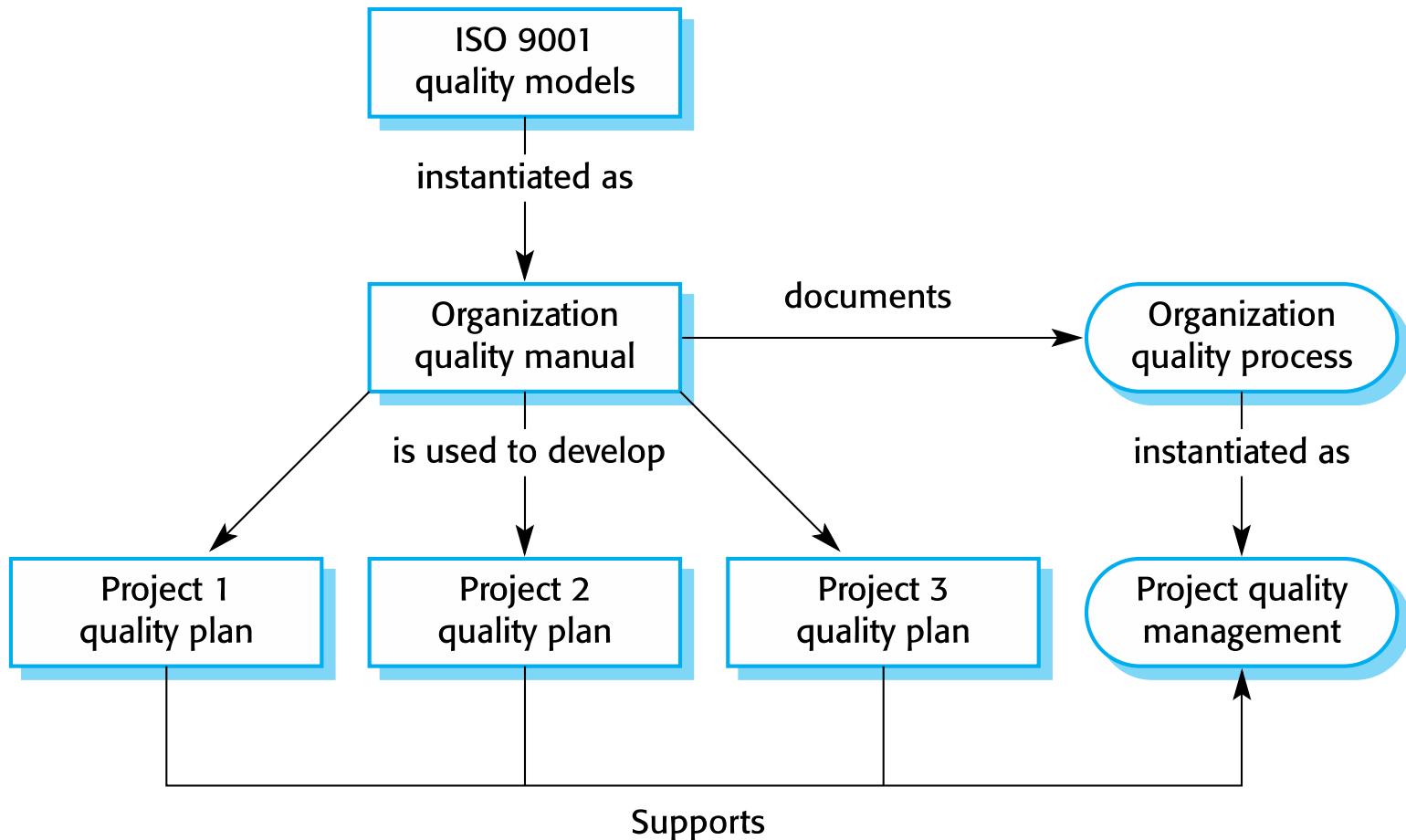
ISO 9001 Standards Framework

- ✧ An international set of standards that can be used as a basis for developing quality management systems.
- ✧ ISO 9001, the most general of these standards, applies to organizations that design, develop and maintain products, including software.
- ✧ The ISO 9001 standards is a framework for developing software standards.
 - It sets out general quality principles, describes quality processes in general and lays out the organizational standards and procedures that should be defined. These should be documented in an organizational quality manual.

ISO 9001 core processes



ISO 9001 and Quality Management



ISO 9001 Certification

- ✧ **Quality standards and procedures** should be documented in an **Organizational Quality Manual**.
- ✧ An external body may certify that an organisation's quality manual conforms to ISO 9000 standards.
- ✧ Some customers require suppliers to be ISO 9000 certified although the need for flexibility here is increasingly recognised.

Software Quality and ISO 9001

- ✧ The ISO 9001 certification is inadequate because it defines quality to be the conformance to standards.
- ✧ It takes no account of quality as experienced by users of the software. For example, a company could define test coverage standards specifying that all methods in objects must be called at least once.
- ✧ Unfortunately, this standard can be met by incomplete software testing that does not include tests with different method parameters. So long as the defined testing procedures are followed and test records maintained, the company could be ISO 9001 certified.

Reviews and Inspections

Reviews and Inspections

- ✧ A group examines part or all of a process or system and its documentation to find potential problems.
- ✧ Software or documents may be 'signed off' at a review which signifies that progress to the next development stage has been approved by management.
- ✧ There are **different types of reviews** with different objectives
 - **Inspections** for defect removal (**product**);
 - **Reviews** for progress assessment (**product and process**);
 - **Quality reviews** (**product and standards**).

Quality Reviews

- ✧ A group of people carefully examine part or all of a software system and its associated documentation.
- ✧ Code, designs, specifications, test plans, standards, etc. can all be reviewed.
- ✧ Software or documents may be 'signed off' at a review which signifies that progress to the next development stage has been approved by management.

Phases in the Review process

1. Pre-review activities

- Pre-review activities are concerned with review planning and review preparation

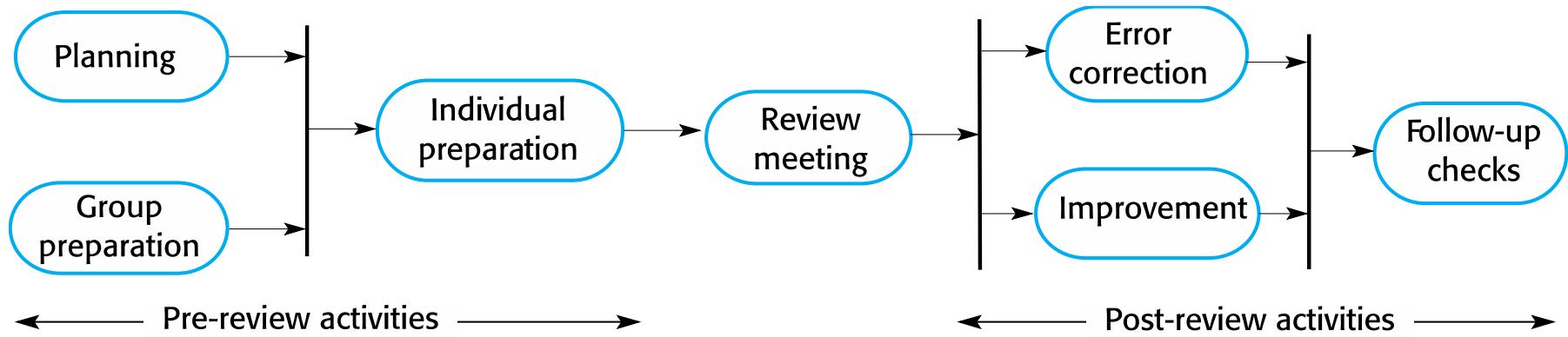
2. The Review meeting

- During the review meeting, an author of the document or program being reviewed should ‘walk through’ the document with the review team.

3. Post-review activities

- These address the problems and issues that have been raised during the review meeting.

The Software Review process



Distributed Reviews

- ✧ The processes suggested for reviews assume that the review team has a face-to-face meeting to discuss the software or documents that they are reviewing.
- ✧ However, project teams are now often distributed, sometimes across countries or continents, so it is impractical for team members to meet face to face.
- ✧ Remote reviewing can be supported using shared documents where each review team member can annotate the document with their comments.

Program Inspections

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

Inspection Checklists

- ✧ Checklist of common errors should be used to drive the inspection.
- ✧ Error checklists are programming language dependent and reflect the characteristic errors that are likely to arise in the language.
- ✧ In general, the 'weaker' the type checking, the larger the checklist.
- ✧ Examples: Initialisation, Constant naming, loop termination, array bounds, etc.

An Inspection Checklist (1 of 2)

Fault class	Inspection check
Data faults	<ul style="list-style-type: none">• Are all program variables initialized before their values are used?• Have all constants been named?• Should the upper bound of arrays be equal to the size of the array or Size -1?• If character strings are used, is a delimiter explicitly assigned?• Is there any possibility of buffer overflow?
Control faults	<ul style="list-style-type: none">• For each conditional statement, is the condition correct?• Is each loop certain to terminate?• Are compound statements correctly bracketed?• In case statements, are all possible cases accounted for?• If a break is required after each case in case statements, has it been included?
Input / Output faults	<ul style="list-style-type: none">• Are all input variables used?• Are all output variables assigned a value before they are output?• Can unexpected inputs cause corruption?

An Inspection Checklist (2 of 2)

Fault class	Inspection check
Interface faults	<ul style="list-style-type: none">• Do all function and method calls have the correct number of parameters?• Do formal and actual parameter types match?• Are the parameters in the right order?• If components access shared memory, do they have the same model of the shared memory structure?
Storage management faults	<ul style="list-style-type: none">• If a linked structure is modified, have all links been correctly reassigned?• If dynamic storage is used, has space been allocated correctly?• Is space explicitly deallocated after it is no longer required?
Exception management faults	<ul style="list-style-type: none">• Have all possible error conditions been taken into account?

Quality Management and Agile Development

Quality Management and Agile Development

- ✧ **Quality management in Agile development** is informal rather than document-based.
- ✧ It relies on establishing a **quality culture**, where all team members feel responsible for software quality and take actions to ensure that quality is maintained.
- ✧ The agile community is fundamentally opposed to what it sees as the bureaucratic overheads of standards-based approaches and quality processes as embodied in ISO 9001.

Shared good practices in Agile

1. Check before check-in

- Programmers are responsible for organizing their own code reviews with other team members before the code is checked in to the build system.

2. Never break the build

- Team members should not check in code that causes the system to fail. Developers have to test their code changes against the whole system and be confident that these work as expected.

3. Fix problems when you see them

- If a programmer discovers problems or obscurities in code developed by someone else, they can fix these directly rather than referring them back to the original developer.

Reviews and Agile methods

- ✧ The **review process** in agile software development **is usually informal**.
- ✧ In **Scrum**, there is a review meeting after each iteration of the software has been completed (a sprint review), where quality issues and problems may be discussed.
- ✧ In **Extreme Programming**, pair programming ensures that code is constantly being examined and reviewed by another team member.

Pair Programming in XP

- ✧ This is an approach where 2 people are responsible for code development and work together to achieve this.
- ✧ Code developed by an individual is therefore constantly being examined and reviewed by another team member.
- ✧ **Pair programming** leads to a deep knowledge of a program, as both programmers have to understand the program in detail to continue development.
- ✧ This depth of knowledge is difficult to achieve in inspection processes and pair programming can find bugs that would not be discovered in formal inspections.

Pair Programming weaknesses

✧ **Mutual misunderstandings**

- Both members of a pair may make the same mistake in understanding the system requirements. Discussions may reinforce these errors.

✧ **Pair reputation**

- Pairs may be reluctant to look for errors because they do not want to slow down the progress of the project.

✧ **Working relationships**

- The pair's ability to discover defects is likely to be compromised by their close working relationship that often leads to reluctance to criticize work partners.

Agile Quality Management and large systems

- ✧ When a **large system** is being developed for an external customer, **agile approaches to quality management with minimal documentation may be impractical.**
 - If the customer is a large company, it may have its own quality management processes and may expect the software development company to report on progress in a way that is compatible with them.
 - Where there are several geographically distributed teams involved in development, perhaps from different companies, then informal communications may be impractical.
 - For long-lifetime systems, the team involved in development will change without documentation, new team members may find it impossible to understand development.

Software Measurement

Software measurement

- ✧ **Software measurement** is concerned with deriving a numeric value for an attribute of a software product or process.
- ✧ This allows for objective comparisons between techniques and processes.
- ✧ Although some companies have introduced measurement programmes, most organisations still don't make systematic use of software measurement.
- ✧ There are few established standards in this area.

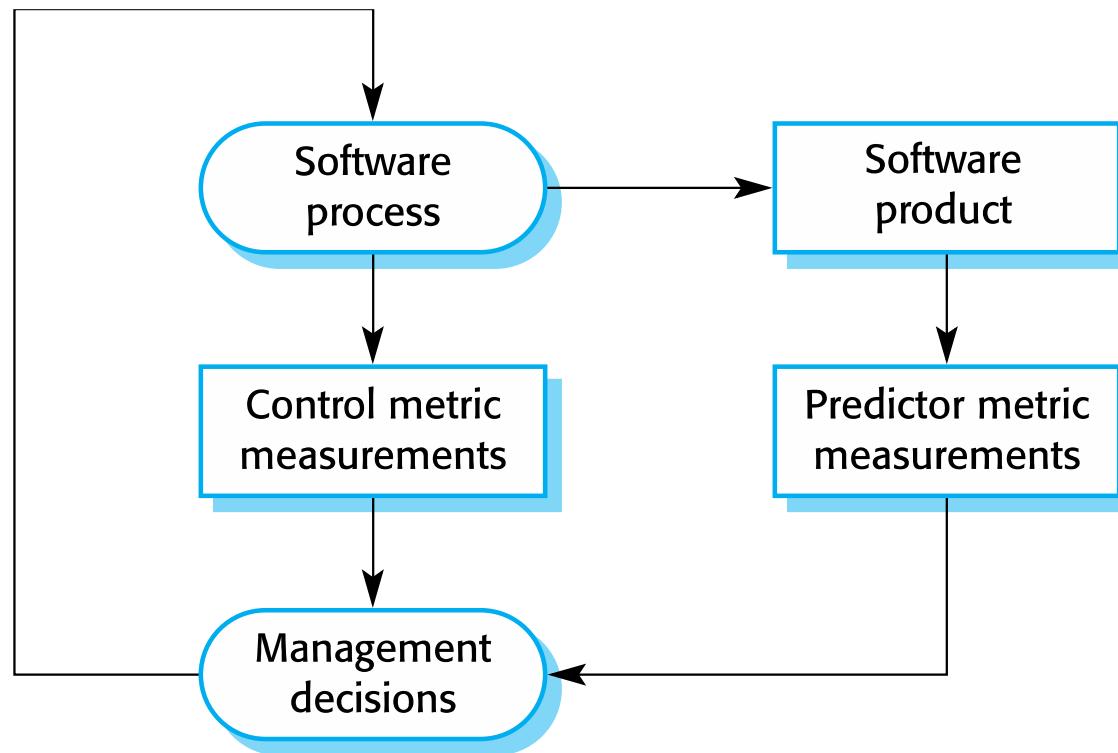
Software metrics

- ✧ Any type of measurement which relates to a software system, process or related documentation
 - Lines of code in a program, the Fog index, number of person-days required to develop a component.
- ✧ Metrics allow the software product and the software process to be quantified.
- ✧ Metrics may be used to predict product attributes or to control the software process.
- ✧ Product metrics can be used for general predictions or to identify anomalous components.

Types of Process metric

- ✧ **The time taken for a particular process to be completed**
 - This can be the total time devoted to the process, calendar time, the time spent on the process by particular engineers, and so on.
- ✧ **The resources required for a particular process**
 - Resources might include total effort in person-days, travel costs or computer resources.
- ✧ **The number of occurrences of a particular event**
 - Examples of events that might be monitored include the number of defects discovered during code inspection, the number of requirements changes requested, the number of bug reports in a delivered system and the average number of lines of code modified in response to a requirements change.

Predictor and Control measurements



Use of Measurements

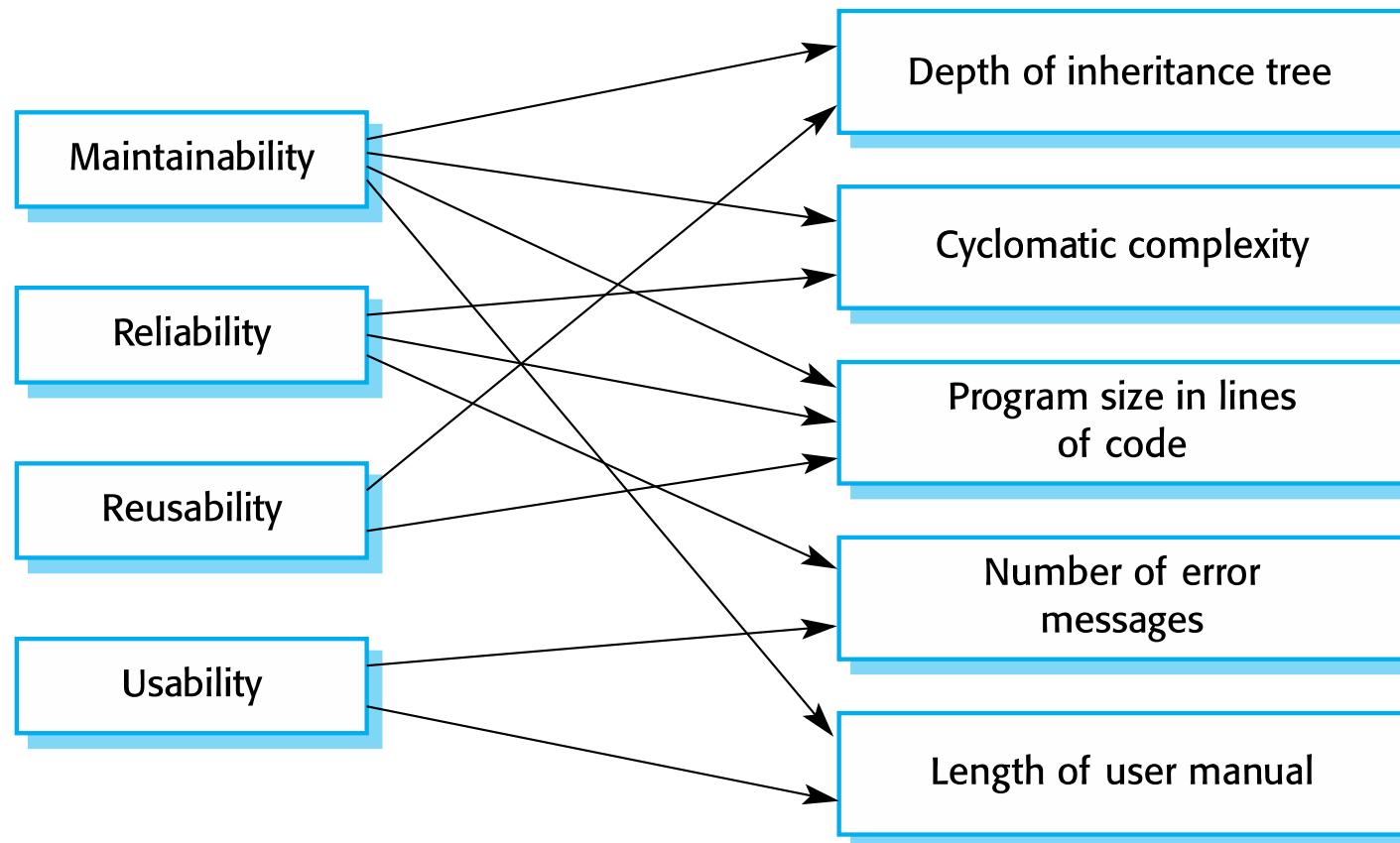
- ✧ To assign a value to system quality attributes
 - By measuring the characteristics of system components, such as their **cyclomatic complexity**, and then aggregating these measurements, you can assess system quality attributes, such as maintainability.
- ✧ To identify the system components whose quality is sub-standard
 - Measurements can identify individual components with characteristics that deviate from the norm.
 - For example, you can measure components to discover those with the highest complexity. These are most likely to contain bugs because the complexity makes them harder to understand.

Metrics assumptions

- ✧ A software property can be measured accurately.
- ✧ The relationship exists between what we can measure and what we want to know. We can only measure internal attributes but are often more interested in external software attributes.
- ✧ This relationship has been formalised and validated.
- ✧ It may be difficult to relate what can be measured to desirable external quality attributes.

Internal and External quality attributes

External quality attributes Internal attributes



Problems with measurement in industry

- ✧ It is impossible to quantify the return on investment of introducing an organizational metrics program.
- ✧ There are no standards for software metrics or standardized processes for measurement and analysis.
- ✧ In many companies, software processes are not standardized and are poorly defined and controlled.
- ✧ Most work on software measurement has focused on code-based metrics and plan-driven development processes. However, more and more software is now developed by configuring ERP systems or COTS.
- ✧ Introducing measurement adds additional overhead to processes.

Empirical software engineering

- ✧ Software measurement and metrics are the basis of empirical software engineering.
- ✧ This is a research area in which experiments on software systems and the collection of data about real projects has been used to form and validate hypotheses about software engineering methods and techniques.
- ✧ Research on empirical software engineering, this has not had a significant impact on software engineering practice.
- ✧ It is difficult to relate generic research to a project that is different from the research study.

Product metrics

- ✧ A quality metric should be a predictor of product quality.
- ✧ **Classes of product metrics**
 - **Dynamic metrics** which are collected by measurements made of a program in execution;
 - **Static metrics** which are collected by measurements made of the system representations;
 - **Dynamic metrics** help assess efficiency and reliability
 - **Static metrics** help assess complexity, understandability and maintainability.

Dynamic metrics and Static metrics

- ✧ **Dynamic metrics** are closely related to software quality attributes
 - It is relatively easy to measure the response time of a system (performance attribute) or the number of failures (reliability attribute).
- ✧ **Static metrics** have an indirect relationship with quality attributes
 - You need to try and derive a relationship between these metrics and properties such as complexity, understandability and maintainability.

Static software product metrics

Software metric	Description
Fan-in/Fan-out	<p>Fan-in is a measure of the number of functions or methods that call another function or method (say X). Fan-out is the number of functions that are called by function X. A high value for fan-in means that X is tightly coupled to the rest of the design and changes to X will have extensive knock-on effects. A high value for fan-out suggests that the overall complexity of X may be high because of the complexity of the control logic needed to coordinate the called components.</p>
Length of code	<p>This is a measure of the size of a program. Generally, the larger the size of the code of a component, the more complex and error-prone that component is likely to be. Length of code has been shown to be one of the most reliable metrics for predicting error-proneness in components.</p>

Static software product metrics

Software metric	Description
Cyclomatic complexity	This is a measure of the control complexity of a program. This control complexity may be related to program understandability.
Length of identifiers	This is a measure of the average length of identifiers (names for variables, classes, methods, etc.) in a program. The longer the identifiers, the more likely they are to be meaningful and hence the more understandable the program.
Depth of conditional nesting	This is a measure of the depth of nesting of if-statements in a program. Deeply nested if-statements are hard to understand and potentially error-prone.
Fog index	This is a measure of the average length of words and sentences in documents . The higher the value of a document's Fog index, the more difficult the document is to understand.

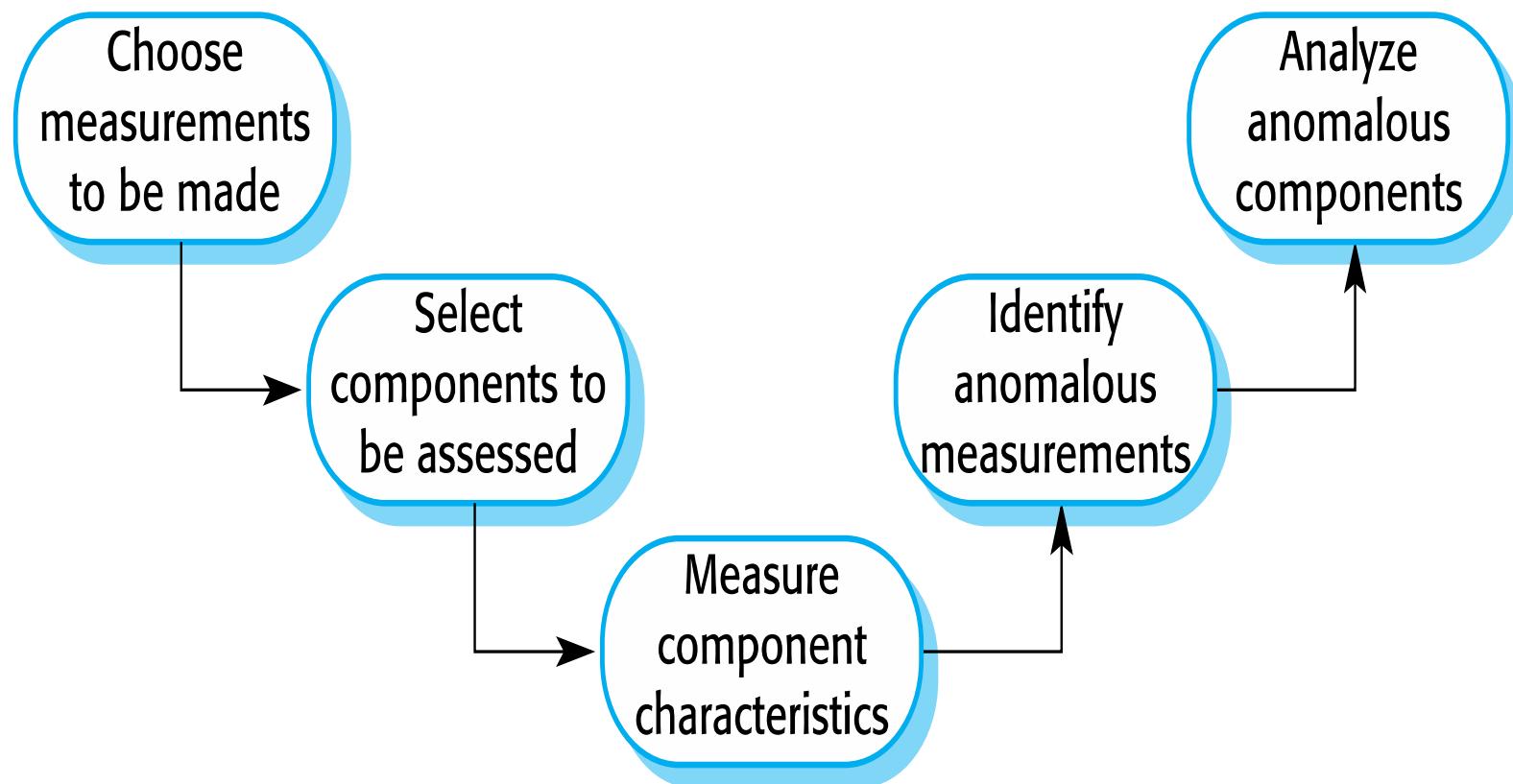
The CK object-oriented metrics suite

Object-oriented metric	Description
Weighted methods per class (WMC)	This is the number of methods in each class, weighted by the complexity of each method. Therefore, a simple method may have a complexity of 1, and a large and complex method a much higher value. The larger the value for this metric, the more complex the object class. Complex objects are more likely to be difficult to understand. They may not be logically cohesive, so cannot be reused effectively as superclasses in an inheritance tree.
Depth of inheritance tree (DIT)	This represents the number of discrete levels in the inheritance tree where subclasses inherit attributes and operations (methods) from superclasses. The deeper the inheritance tree, the more complex the design. Many object classes may have to be understood to understand the object classes at the leaves of the tree.
Number of children (NOC)	This is a measure of the number of immediate subclasses in a class. It measures the breadth of a class hierarchy, whereas DIT measures its depth. A high value for NOC may indicate greater reuse. It may mean that more effort should be made in validating base classes because of the number of subclasses that depend on them.

Software component analysis

- ✧ System component can be analyzed separately using a range of metrics.
- ✧ The values of these metrics may then compared for different components and, perhaps, with historical measurement data collected on previous projects.
- ✧ **Anomalous measurements**, which deviate significantly from the norm, may imply that there are problems with the quality of these components.

The process of product measurement



Measurement ambiguity

- ✧ When you collect quantitative data about software and software processes, you have to analyze that data to understand its meaning.
- ✧ It is easy to misinterpret data and to make inferences that are incorrect.
- ✧ You cannot simply look at the data on its own. You must also consider the context where the data is collected.

Measurement surprises

- ✧ Reducing the number of faults in a program leads to an increased number of help desk calls
 - The program is now thought of as more reliable and so has a wider more diverse market. The percentage of users who call the help desk may have decreased but the total may increase;
 - A more reliable system is used in a different way from a system where users work around the faults. This leads to more help desk calls.

Software context

- ✧ Processes and products that are being measured **are not insulated from their environment.**
- ✧ The business environment is constantly changing and it is impossible to avoid changes to work practice just because they may make comparisons of data invalid.
- ✧ Data about human activities cannot always be taken at face value. The reasons why a measured value changes are often ambiguous. These reasons must be investigated in detail before drawing conclusions from any measurements that have been made.

Software analytics

- ✧ **Software analytics** is analytics on software data for managers and software engineers with the aim of empowering software development individuals and teams to gain and share insight from their data to make better decisions.

Software analytics enablers

- ✧ The automated collection of user data by software product companies when their product is used.
 - If the software fails, information about the failure and the state of the system can be sent over the Internet from the user's computer to servers run by the product developer.
- ✧ The use of open source software available on platforms such as Sourceforge and GitHub and open source repositories of software engineering data.
 - The source code of open source software is available for automated analysis and this can sometimes be linked with data in the open source repository.

Analytics tool use

- ✧ Tools should be easy to use as managers are unlikely to have experience with analysis.
- ✧ Tools should run quickly and produce concise outputs rather than large volumes of information.
- ✧ Tools should make many measurements using as many parameters as possible. It is impossible to predict in advance what insights might emerge.
- ✧ Tools should be interactive and allow managers and developers to explore the analyses.

Status of software analytics

- ✧ Software analytics is still immature and it is too early to say what effect it will have.
- ✧ Not only are there general problems of ‘big data’ processing, our knowledge depends on collected data from large companies.
 - This is primarily from software products and it is unclear if the tools and techniques that are appropriate for products can also be used with custom software.
- ✧ Small companies are unlikely to invest in the data collection systems that are required for automated analysis so may not be able to use software analytics.

Quality Management: Key points (1 / 3)

- ✧ **Software quality management** is concerned with ensuring that software has a low number of defects and that it reaches the required standards of maintainability, reliability, portability etc.
- ✧ **Software standards** are important for quality assurance as they represent an identification of ‘best practice’. When developing software, standards provide a solid foundation for building good quality software.
- ✧ **Reviews of the software process deliverables** involve a team of people who check that quality standards are being followed. Reviews are the most widely used technique for assessing quality.

Quality Management: Key points (2 / 3)

- ✧ **In a program inspection or peer review**, a small team systematically checks the code. They read the code in detail and look for possible errors and omissions. The problems detected are discussed at a code review meeting.
- ✧ **Agile quality management** relies on establishing a quality culture where the development team works together to improve software quality.
- ✧ **Software measurement** can be used to gather quantitative data about software and the software process.

Quality Management: Key points (3 / 3)

- ✧ You may be able to use the values of the **software metrics** that are collected to make inferences about **product quality and process quality**.
- ✧ **Product quality metrics** are particularly useful for highlighting **anomalous components** that may have quality problems. These components should then be analyzed in more detail.
- ✧ **Software analytics** is the automated analysis of large volumes of software product and process data to **discover relationships that may provide insights** for project managers and developers.

Thank you.



Software Quality Management

Lecture # 6: Quality Management

Quality Management: Topics covered

1. Software Quality
2. Software Standards
3. Reviews and Inspections
4. Quality Management and Agile Development
5. Software Measurement

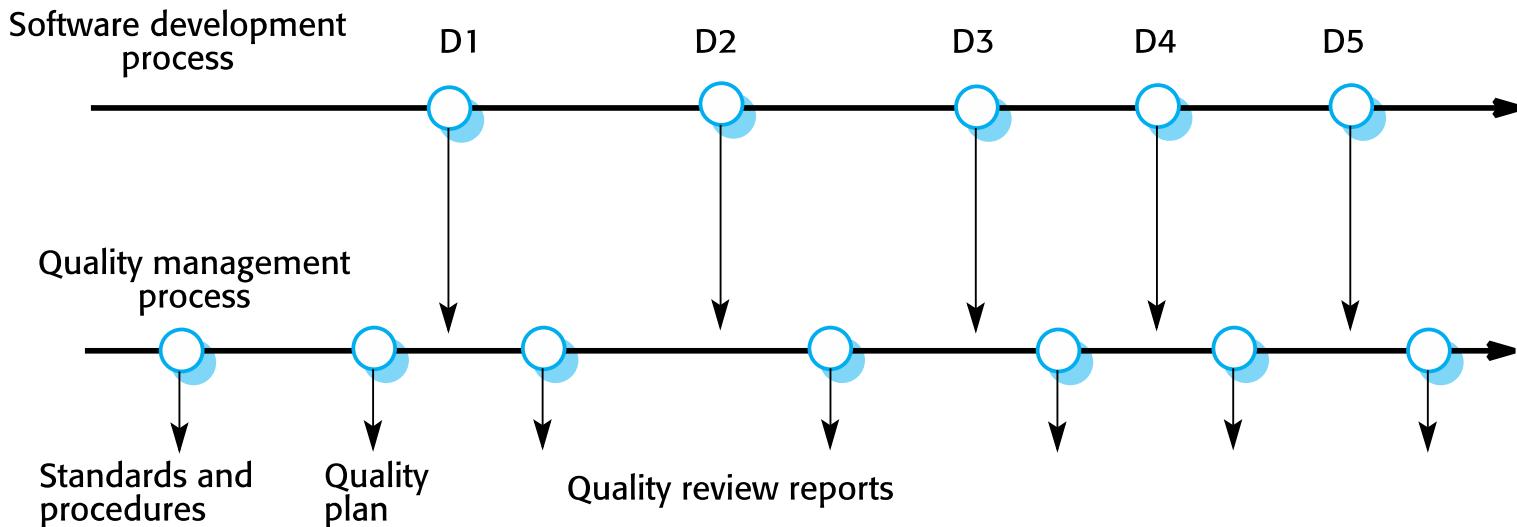
Software Quality Management

- ✧ Concerned with ensuring that the required level of quality is achieved in a software product.
- ✧ Three principal concerns:
 1. At the organizational level, quality management is concerned with establishing a framework of organizational processes and standards that will lead to high-quality software.
 2. At the project level, quality management involves the application of specific quality processes and checking that these planned processes have been followed.
 3. At the project level, quality management is also concerned with establishing a quality plan for a project. The quality plan should set out the quality goals for the project and define what processes and standards are to be used.

Quality Management activities

- ✧ Quality Management provides an independent check on the software development process.
- ✧ The Quality Management process checks the project deliverables to ensure that they are consistent with organizational standards and goals
- ✧ The Quality team should be independent from the development team so that they can take an objective view of the software. This allows them to report on software quality without being influenced by software development issues.

Quality Management and Software Development



Quality Planning

- ✧ A **Quality Plan** sets out the desired product qualities and how these are assessed and defines the most significant quality attributes.
- ✧ The **Quality Plan** should define the quality assessment process.
- ✧ It should set out which organizational standards should be applied and, where necessary, define new standards to be used.

Quality plans

✧ **Quality Plan** structure

- Product introduction;
- Product plans;
- Process descriptions;
- Quality goals;
- Risks and risk management.

✧ **Quality Plans** should be short, succinct documents

- If they are too long, no-one will read them.

Scope of Quality Management

- ✧ Quality Management is particularly important for large, complex systems. The quality documentation is a record of progress and supports continuity of development as the development team changes.
- ✧ For smaller systems, Quality Management needs less documentation and should focus on establishing a quality culture.
- ✧ Techniques have to evolve when agile development is used.

Software Quality

Software Quality

- ✧ Quality, simplistically, means that a product should meet its specification.
- ✧ This is problematical for software systems:
 - There is a tension between customer quality requirements (efficiency, reliability, etc.) and developer quality requirements (maintainability, reusability, etc.);
 - Some quality requirements are difficult to specify in an unambiguous way;
 - Software specifications are usually incomplete and often inconsistent.
- ✧ The **focus** may be ‘fitness for purpose’ rather than specification conformance.

Software fitness for purpose

1. Has the software been properly tested?
2. Is the software sufficiently dependable to be put into use?
3. Is the performance of the software acceptable for normal use?
4. Is the software usable?
5. Is the software well-structured and understandable?
6. Have programming and documentation standards been followed in the development process?

Non-functional characteristics

- ✧ The subjective quality of a software system is largely based on its non-functional characteristics.
- ✧ This reflects practical user experience – if the software's functionality is not what is expected, then users will often just work around this and find other ways to do what they want to do.
- ✧ However, if the software is unreliable or too slow, then it is practically impossible for them to achieve their goals.

Software Quality attributes

Safety	Understandability	Portability
Security	Testability	Usability
Reliability	Adaptability	Reusability
Resilience	Modularity	Efficiency
Robustness	Complexity	Learnability

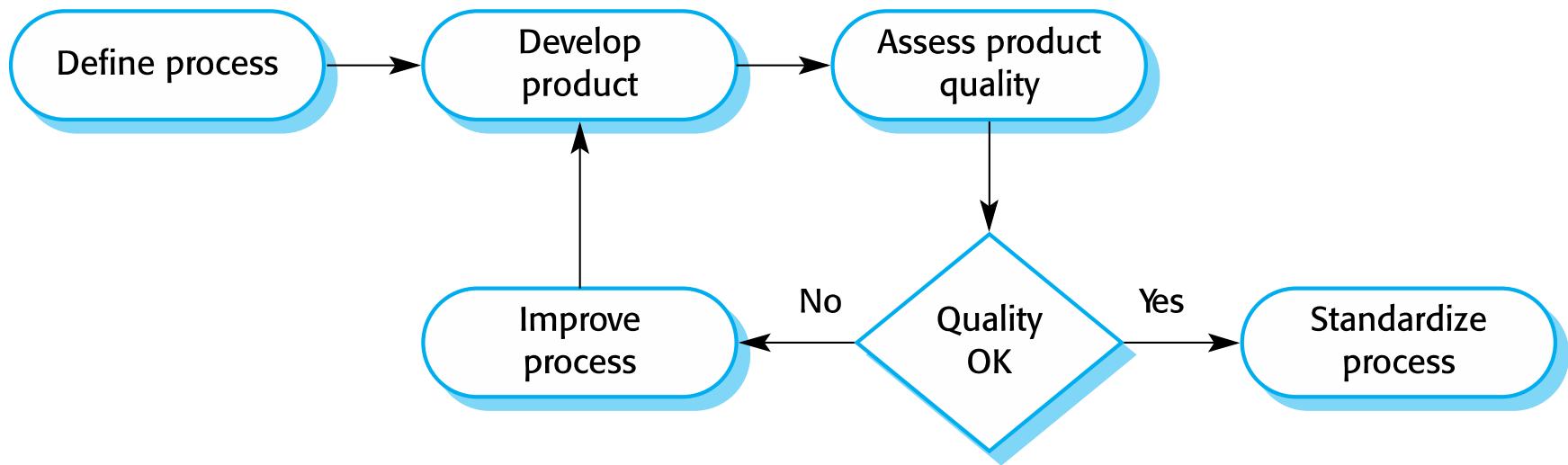
Quality conflicts

- ✧ It is not possible for any system to be optimized for all of these attributes – for example, improving robustness may lead to loss of performance.
- ✧ The **Quality Plan** should therefore define the **most important quality attributes** for the software that is being developed.
- ✧ The plan should also include a **definition of the quality assessment process**, an agreed way of assessing whether some quality, such as maintainability or robustness, is present in the product.

Process Quality and Product Quality

- ✧ The quality of a developed product is influenced by the quality of the production process.
- ✧ This is important in software development as some product quality attributes are hard to assess.
- ✧ However, there is a very complex and poorly understood relationship between software processes and product quality.
 - The application of individual skills and experience is particularly important in software development;
 - External factors such as the novelty of an application or the need for an accelerated development schedule may impair product quality.

Process-based quality



Quality Culture

- ✧ Quality managers should aim to develop a '**Quality Culture**' where everyone responsible for software development is committed to achieving a high level of product quality.
- ✧ They should encourage teams to take responsibility for the quality of their work and to develop new approaches to quality improvement.
- ✧ They should support people who are interested in the intangible aspects of quality and encourage professional behavior in all team members.

Software Standards

Software Standards

- ✧ **Standards** define the required attributes of a product or process. They play an important role in quality management.
- ✧ Standards may be international, national, organizational, or project standards.

Importance of Standards

- ✧ Encapsulation of best practices - avoids repetition of past mistakes.
- ✧ They are a framework for defining what quality means in a particular setting i.e. that organization's view of quality.
- ✧ They provide continuity - new staff can understand the organisation by understanding the standards that are used.

Product Standards and Process Standards

✧ Product Standards

- **Apply to the software product being developed.** They include document standards, such as the structure of requirements documents, documentation standards, such as a standard comment header for an object class definition, and coding standards, which define how a programming language should be used.

✧ Process standards

- **These define the processes that should be followed during software development.** Process standards may include definitions of specification, design and validation processes, process support tools and a description of the documents that should be written during these processes.

Product Standards and Process Standards

Product Standards	Process Standards
Design review form	Design review conduct
Requirements document structure	Submission of new code for system building
Method header format	Version release process
Java programming style	Project plan approval process
Project plan format	Change control process
Change request form	Test recording process

Problems with Standards

- ✧ They may not be seen as relevant and up-to-date by software engineers.
- ✧ They often involve too much bureaucratic form filling.
- ✧ If they are unsupported by software tools, tedious form filling work is often involved to maintain the documentation associated with the standards.

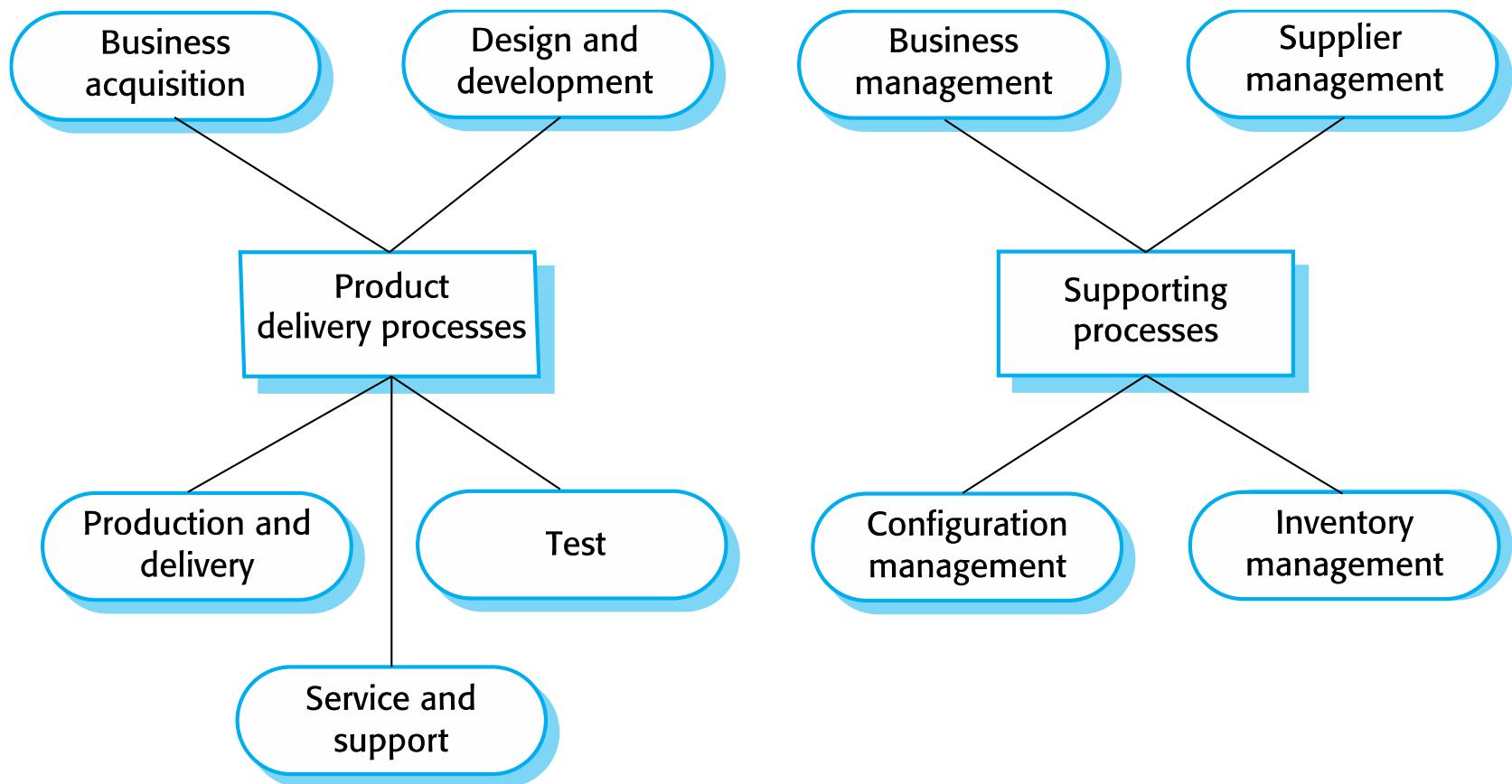
Standards development

- ✧ **Involve practitioners in development.** Engineers should understand the rationale underlying a standard.
- ✧ **Review standards and their usage regularly.**
Standards can quickly become outdated and this reduces their credibility amongst practitioners.
- ✧ **Detailed standards should have specialized tool support.** Excessive clerical work is the most significant complaint against standards.
 - Web-based forms are not good enough.

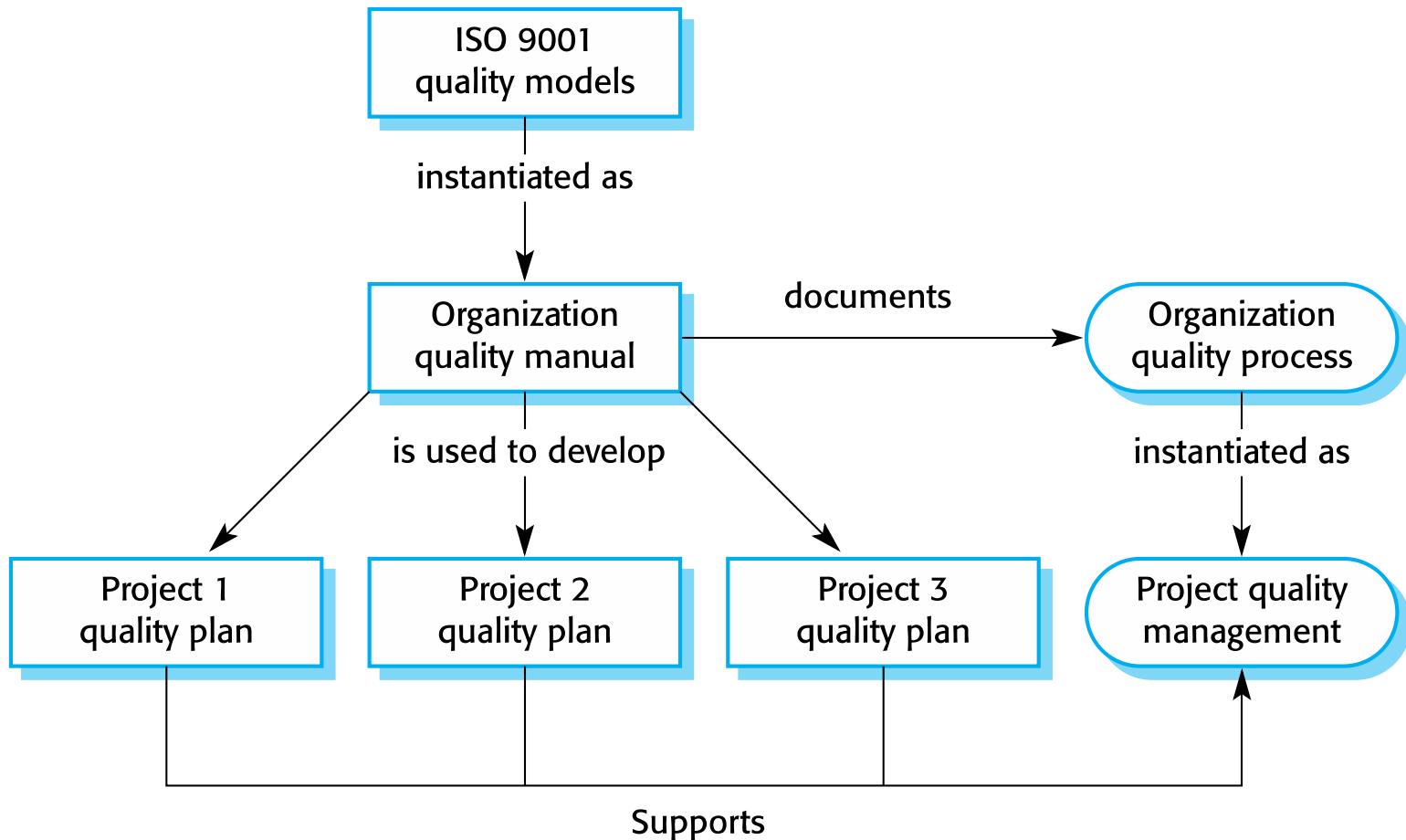
ISO 9001 Standards Framework

- ✧ An international set of standards that can be used as a basis for developing quality management systems.
- ✧ ISO 9001, the most general of these standards, applies to organizations that design, develop and maintain products, including software.
- ✧ The ISO 9001 standards is a framework for developing software standards.
 - It sets out general quality principles, describes quality processes in general and lays out the organizational standards and procedures that should be defined. These should be documented in an organizational quality manual.

ISO 9001 core processes



ISO 9001 and Quality Management



ISO 9001 Certification

- ✧ **Quality standards and procedures** should be documented in an **Organizational Quality Manual**.
- ✧ An external body may certify that an organisation's quality manual conforms to ISO 9000 standards.
- ✧ Some customers require suppliers to be ISO 9000 certified although the need for flexibility here is increasingly recognised.

Software Quality and ISO 9001

- ✧ The ISO 9001 certification is inadequate because it defines quality to be the conformance to standards.
- ✧ It takes no account of quality as experienced by users of the software. For example, a company could define test coverage standards specifying that all methods in objects must be called at least once.
- ✧ Unfortunately, this standard can be met by incomplete software testing that does not include tests with different method parameters. So long as the defined testing procedures are followed and test records maintained, the company could be ISO 9001 certified.

Reviews and Inspections

Reviews and Inspections

- ✧ A group examines part or all of a process or system and its documentation to find potential problems.
- ✧ Software or documents may be 'signed off' at a review which signifies that progress to the next development stage has been approved by management.
- ✧ There are **different types of reviews** with different objectives
 - **Inspections** for defect removal (**product**);
 - **Reviews** for progress assessment (**product and process**);
 - **Quality reviews** (**product and standards**).

Quality Reviews

- ✧ A group of people carefully examine part or all of a software system and its associated documentation.
- ✧ Code, designs, specifications, test plans, standards, etc. can all be reviewed.
- ✧ Software or documents may be 'signed off' at a review which signifies that progress to the next development stage has been approved by management.

Phases in the Review process

1. Pre-review activities

- Pre-review activities are concerned with review planning and review preparation

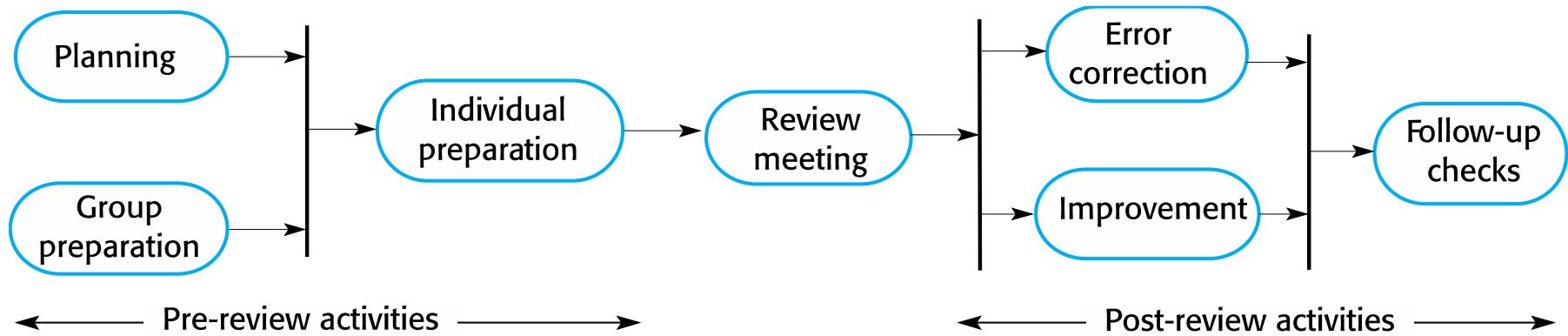
2. The Review meeting

- During the review meeting, an author of the document or program being reviewed should ‘walk through’ the document with the review team.

3. Post-review activities

- These address the problems and issues that have been raised during the review meeting.

The Software Review process



Distributed Reviews

- ✧ The processes suggested for reviews assume that the review team has a face-to-face meeting to discuss the software or documents that they are reviewing.
- ✧ However, project teams are now often distributed, sometimes across countries or continents, so it is impractical for team members to meet face to face.
- ✧ Remote reviewing can be supported using shared documents where each review team member can annotate the document with their comments.

Program Inspections

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

Inspection Checklists

- ✧ Checklist of common errors should be used to drive the inspection.
- ✧ Error checklists are programming language dependent and reflect the characteristic errors that are likely to arise in the language.
- ✧ In general, the 'weaker' the type checking, the larger the checklist.
- ✧ Examples: Initialisation, Constant naming, loop termination, array bounds, etc.

An Inspection Checklist (1 of 2)

Fault class	Inspection check
Data faults	<ul style="list-style-type: none">• Are all program variables initialized before their values are used?• Have all constants been named?• Should the upper bound of arrays be equal to the size of the array or Size -1?• If character strings are used, is a delimiter explicitly assigned?• Is there any possibility of buffer overflow?
Control faults	<ul style="list-style-type: none">• For each conditional statement, is the condition correct?• Is each loop certain to terminate?• Are compound statements correctly bracketed?• In case statements, are all possible cases accounted for?• If a break is required after each case in case statements, has it been included?
Input / Output faults	<ul style="list-style-type: none">• Are all input variables used?• Are all output variables assigned a value before they are output?• Can unexpected inputs cause corruption?

An Inspection Checklist (2 of 2)

Fault class	Inspection check
Interface faults	<ul style="list-style-type: none">• Do all function and method calls have the correct number of parameters?• Do formal and actual parameter types match?• Are the parameters in the right order?• If components access shared memory, do they have the same model of the shared memory structure?
Storage management faults	<ul style="list-style-type: none">• If a linked structure is modified, have all links been correctly reassigned?• If dynamic storage is used, has space been allocated correctly?• Is space explicitly deallocated after it is no longer required?
Exception management faults	<ul style="list-style-type: none">• Have all possible error conditions been taken into account?

Quality Management and Agile Development

Quality Management and Agile Development

- ✧ **Quality management in Agile development** is informal rather than document-based.
- ✧ It relies on establishing a **quality culture**, where all team members feel responsible for software quality and take actions to ensure that quality is maintained.
- ✧ The agile community is fundamentally opposed to what it sees as the bureaucratic overheads of standards-based approaches and quality processes as embodied in ISO 9001.

Shared good practices in Agile

1. Check before check-in

- Programmers are responsible for organizing their own code reviews with other team members before the code is checked in to the build system.

2. Never break the build

- Team members should not check in code that causes the system to fail. Developers have to test their code changes against the whole system and be confident that these work as expected.

3. Fix problems when you see them

- If a programmer discovers problems or obscurities in code developed by someone else, they can fix these directly rather than referring them back to the original developer.

Reviews and Agile methods

- ✧ The **review process** in agile software development **is usually informal**.
- ✧ In **Scrum**, there is a review meeting after each iteration of the software has been completed (a sprint review), where quality issues and problems may be discussed.
- ✧ In **Extreme Programming**, pair programming ensures that code is constantly being examined and reviewed by another team member.

Pair Programming in XP

- ✧ This is an approach where 2 people are responsible for code development and work together to achieve this.
- ✧ Code developed by an individual is therefore constantly being examined and reviewed by another team member.
- ✧ **Pair programming** leads to a deep knowledge of a program, as both programmers have to understand the program in detail to continue development.
- ✧ This depth of knowledge is difficult to achieve in inspection processes and pair programming can find bugs that would not be discovered in formal inspections.

Pair Programming weaknesses

✧ **Mutual misunderstandings**

- Both members of a pair may make the same mistake in understanding the system requirements. Discussions may reinforce these errors.

✧ **Pair reputation**

- Pairs may be reluctant to look for errors because they do not want to slow down the progress of the project.

✧ **Working relationships**

- The pair's ability to discover defects is likely to be compromised by their close working relationship that often leads to reluctance to criticize work partners.

Agile Quality Management and large systems

- ✧ When a **large system** is being developed for an external customer, **agile approaches to quality management with minimal documentation may be impractical.**
 - If the customer is a large company, it may have its own quality management processes and may expect the software development company to report on progress in a way that is compatible with them.
 - Where there are several geographically distributed teams involved in development, perhaps from different companies, then informal communications may be impractical.
 - For long-lifetime systems, the team involved in development will change without documentation, new team members may find it impossible to understand development.

Software Measurement

Software measurement

- ✧ **Software measurement** is concerned with deriving a numeric value for an attribute of a software product or process.
- ✧ This allows for objective comparisons between techniques and processes.
- ✧ Although some companies have introduced measurement programmes, most organisations still don't make systematic use of software measurement.
- ✧ There are few established standards in this area.

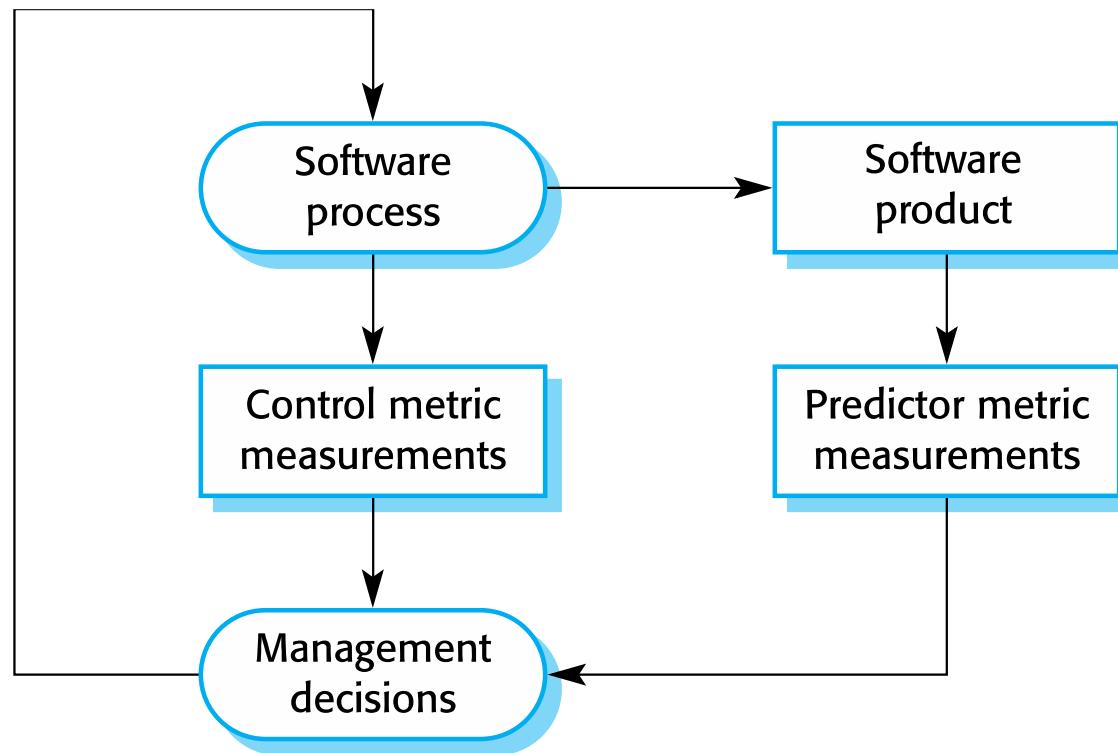
Software metrics

- ✧ Any type of measurement which relates to a software system, process or related documentation
 - Lines of code in a program, the Fog index, number of person-days required to develop a component.
- ✧ Metrics allow the software product and the software process to be quantified.
- ✧ Metrics may be used to predict product attributes or to control the software process.
- ✧ Product metrics can be used for general predictions or to identify anomalous components.

Types of Process metric

- ✧ **The time taken for a particular process to be completed**
 - This can be the total time devoted to the process, calendar time, the time spent on the process by particular engineers, and so on.
- ✧ **The resources required for a particular process**
 - Resources might include total effort in person-days, travel costs or computer resources.
- ✧ **The number of occurrences of a particular event**
 - Examples of events that might be monitored include the number of defects discovered during code inspection, the number of requirements changes requested, the number of bug reports in a delivered system and the average number of lines of code modified in response to a requirements change.

Predictor and Control measurements



Use of Measurements

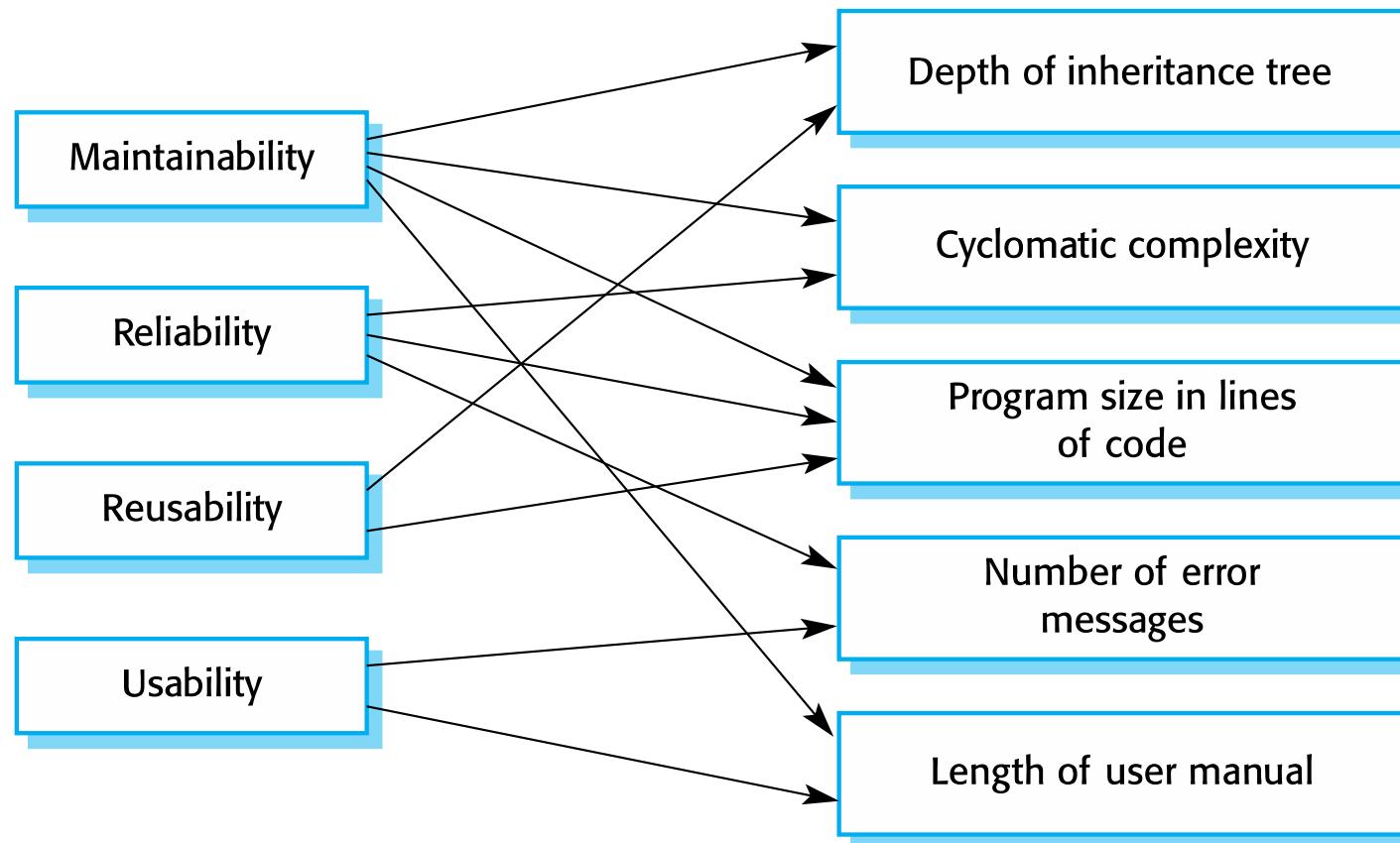
- ✧ To assign a value to system quality attributes
 - By measuring the characteristics of system components, such as their **cyclomatic complexity**, and then aggregating these measurements, you can assess system quality attributes, such as maintainability.
- ✧ To identify the system components whose quality is sub-standard
 - Measurements can identify individual components with characteristics that deviate from the norm.
 - For example, you can measure components to discover those with the highest complexity. These are most likely to contain bugs because the complexity makes them harder to understand.

Metrics assumptions

- ✧ A software property can be measured accurately.
- ✧ The relationship exists between what we can measure and what we want to know. We can only measure internal attributes but are often more interested in external software attributes.
- ✧ This relationship has been formalised and validated.
- ✧ It may be difficult to relate what can be measured to desirable external quality attributes.

Internal and External quality attributes

External quality attributes Internal attributes



Problems with measurement in industry

- ✧ It is impossible to quantify the return on investment of introducing an organizational metrics program.
- ✧ There are no standards for software metrics or standardized processes for measurement and analysis.
- ✧ In many companies, software processes are not standardized and are poorly defined and controlled.
- ✧ Most work on software measurement has focused on code-based metrics and plan-driven development processes. However, more and more software is now developed by configuring ERP systems or COTS.
- ✧ Introducing measurement adds additional overhead to processes.

Empirical software engineering

- ✧ Software measurement and metrics are the basis of empirical software engineering.
- ✧ This is a research area in which experiments on software systems and the collection of data about real projects has been used to form and validate hypotheses about software engineering methods and techniques.
- ✧ Research on empirical software engineering, this has not had a significant impact on software engineering practice.
- ✧ It is difficult to relate generic research to a project that is different from the research study.

Product metrics

- ✧ A quality metric should be a predictor of product quality.
- ✧ **Classes of product metrics**
 - **Dynamic metrics** which are collected by measurements made of a program in execution;
 - **Static metrics** which are collected by measurements made of the system representations;
 - **Dynamic metrics** help assess efficiency and reliability
 - **Static metrics** help assess complexity, understandability and maintainability.

Dynamic metrics and Static metrics

- ✧ **Dynamic metrics** are closely related to software quality attributes
 - It is relatively easy to measure the response time of a system (performance attribute) or the number of failures (reliability attribute).
- ✧ **Static metrics** have an indirect relationship with quality attributes
 - You need to try and derive a relationship between these metrics and properties such as complexity, understandability and maintainability.

Static software product metrics

Software metric	Description
Fan-in/Fan-out	<p>Fan-in is a measure of the number of functions or methods that call another function or method (say X). Fan-out is the number of functions that are called by function X. A high value for fan-in means that X is tightly coupled to the rest of the design and changes to X will have extensive knock-on effects. A high value for fan-out suggests that the overall complexity of X may be high because of the complexity of the control logic needed to coordinate the called components.</p>
Length of code	<p>This is a measure of the size of a program. Generally, the larger the size of the code of a component, the more complex and error-prone that component is likely to be. Length of code has been shown to be one of the most reliable metrics for predicting error-proneness in components.</p>

Static software product metrics

Software metric	Description
Cyclomatic complexity	This is a measure of the control complexity of a program. This control complexity may be related to program understandability.
Length of identifiers	This is a measure of the average length of identifiers (names for variables, classes, methods, etc.) in a program. The longer the identifiers, the more likely they are to be meaningful and hence the more understandable the program.
Depth of conditional nesting	This is a measure of the depth of nesting of if-statements in a program. Deeply nested if-statements are hard to understand and potentially error-prone.
Fog index	This is a measure of the average length of words and sentences in documents . The higher the value of a document's Fog index, the more difficult the document is to understand.

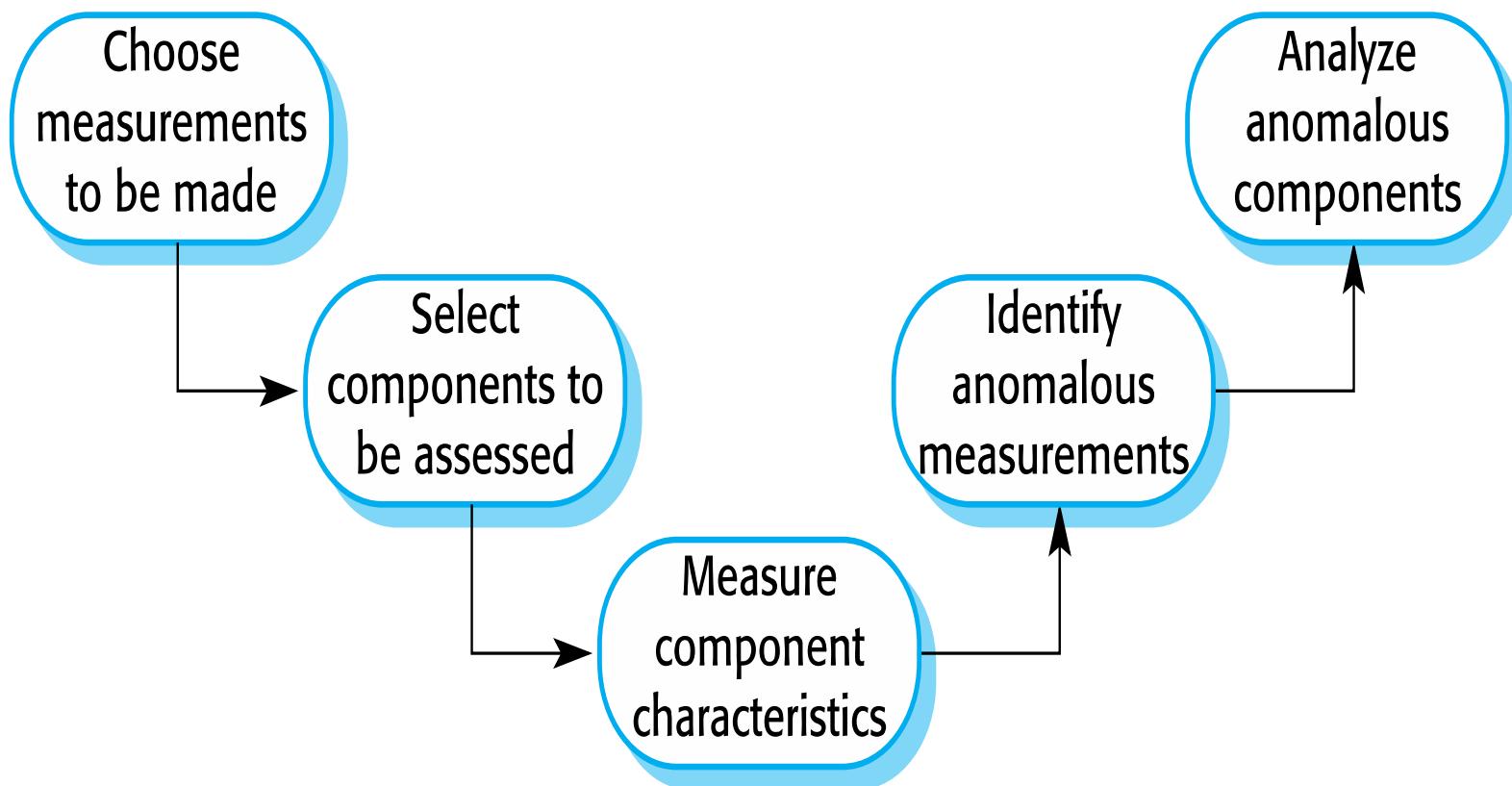
The CK object-oriented metrics suite

Object-oriented metric	Description
Weighted methods per class (WMC)	This is the number of methods in each class, weighted by the complexity of each method. Therefore, a simple method may have a complexity of 1, and a large and complex method a much higher value. The larger the value for this metric, the more complex the object class. Complex objects are more likely to be difficult to understand. They may not be logically cohesive, so cannot be reused effectively as superclasses in an inheritance tree.
Depth of inheritance tree (DIT)	This represents the number of discrete levels in the inheritance tree where subclasses inherit attributes and operations (methods) from superclasses. The deeper the inheritance tree, the more complex the design. Many object classes may have to be understood to understand the object classes at the leaves of the tree.
Number of children (NOC)	This is a measure of the number of immediate subclasses in a class. It measures the breadth of a class hierarchy, whereas DIT measures its depth. A high value for NOC may indicate greater reuse. It may mean that more effort should be made in validating base classes because of the number of subclasses that depend on them.

Software component analysis

- ✧ System component can be analyzed separately using a range of metrics.
- ✧ The values of these metrics may then compared for different components and, perhaps, with historical measurement data collected on previous projects.
- ✧ **Anomalous measurements**, which deviate significantly from the norm, may imply that there are problems with the quality of these components.

The process of product measurement



Measurement ambiguity

- ✧ When you collect quantitative data about software and software processes, you have to analyze that data to understand its meaning.
- ✧ It is easy to misinterpret data and to make inferences that are incorrect.
- ✧ You cannot simply look at the data on its own. You must also consider the context where the data is collected.

Measurement surprises

- ✧ Reducing the number of faults in a program leads to an increased number of help desk calls
 - The program is now thought of as more reliable and so has a wider more diverse market. The percentage of users who call the help desk may have decreased but the total may increase;
 - A more reliable system is used in a different way from a system where users work around the faults. This leads to more help desk calls.

Software context

- ✧ Processes and products that are being measured **are not insulated from their environment.**
- ✧ The business environment is constantly changing and it is impossible to avoid changes to work practice just because they may make comparisons of data invalid.
- ✧ Data about human activities cannot always be taken at face value. The reasons why a measured value changes are often ambiguous. These reasons must be investigated in detail before drawing conclusions from any measurements that have been made.

Software analytics

- ✧ **Software analytics** is analytics on software data for managers and software engineers with the aim of empowering software development individuals and teams to gain and share insight from their data to make better decisions.

Software analytics enablers

- ✧ The automated collection of user data by software product companies when their product is used.
 - If the software fails, information about the failure and the state of the system can be sent over the Internet from the user's computer to servers run by the product developer.
- ✧ The use of open source software available on platforms such as Sourceforge and GitHub and open source repositories of software engineering data.
 - The source code of open source software is available for automated analysis and this can sometimes be linked with data in the open source repository.

Analytics tool use

- ✧ Tools should be easy to use as managers are unlikely to have experience with analysis.
- ✧ Tools should run quickly and produce concise outputs rather than large volumes of information.
- ✧ Tools should make many measurements using as many parameters as possible. It is impossible to predict in advance what insights might emerge.
- ✧ Tools should be interactive and allow managers and developers to explore the analyses.

Status of software analytics

- ✧ Software analytics is still immature and it is too early to say what effect it will have.
- ✧ Not only are there general problems of ‘big data’ processing, our knowledge depends on collected data from large companies.
 - This is primarily from software products and it is unclear if the tools and techniques that are appropriate for products can also be used with custom software.
- ✧ Small companies are unlikely to invest in the data collection systems that are required for automated analysis so may not be able to use software analytics.

Quality Management: Key points (1 / 3)

- ✧ **Software quality management** is concerned with ensuring that software has a low number of defects and that it reaches the required standards of maintainability, reliability, portability etc.
- ✧ **Software standards** are important for quality assurance as they represent an identification of ‘best practice’. When developing software, standards provide a solid foundation for building good quality software.
- ✧ **Reviews of the software process deliverables** involve a team of people who check that quality standards are being followed. Reviews are the most widely used technique for assessing quality.

Quality Management: Key points (2 / 3)

- ✧ **In a program inspection or peer review**, a small team systematically checks the code. They read the code in detail and look for possible errors and omissions. The problems detected are discussed at a code review meeting.
- ✧ **Agile quality management** relies on establishing a quality culture where the development team works together to improve software quality.
- ✧ **Software measurement** can be used to gather quantitative data about software and the software process.

Quality Management: Key points (3 / 3)

- ✧ You may be able to use the values of the **software metrics** that are collected to make inferences about **product quality and process quality**.
- ✧ **Product quality metrics** are particularly useful for highlighting **anomalous components** that may have quality problems. These components should then be analyzed in more detail.
- ✧ **Software analytics** is the automated analysis of large volumes of software product and process data to **discover relationships that may provide insights** for project managers and developers.

Thank you.