Software Development

Basic principles

Software Development Panorama

- RApid software Development (RAD)
  - RAD tools
  - Reuse of existing off the shelf systems
  - Agile Developent
- Reuse-based Software Engineering
- Component-based Software Engineering

Rapid Software Development

Rapid software development

- Rapidly changing business environments
- Businesses may be willing to accept lower quality software if rapid delivery of essential functionality is possible.

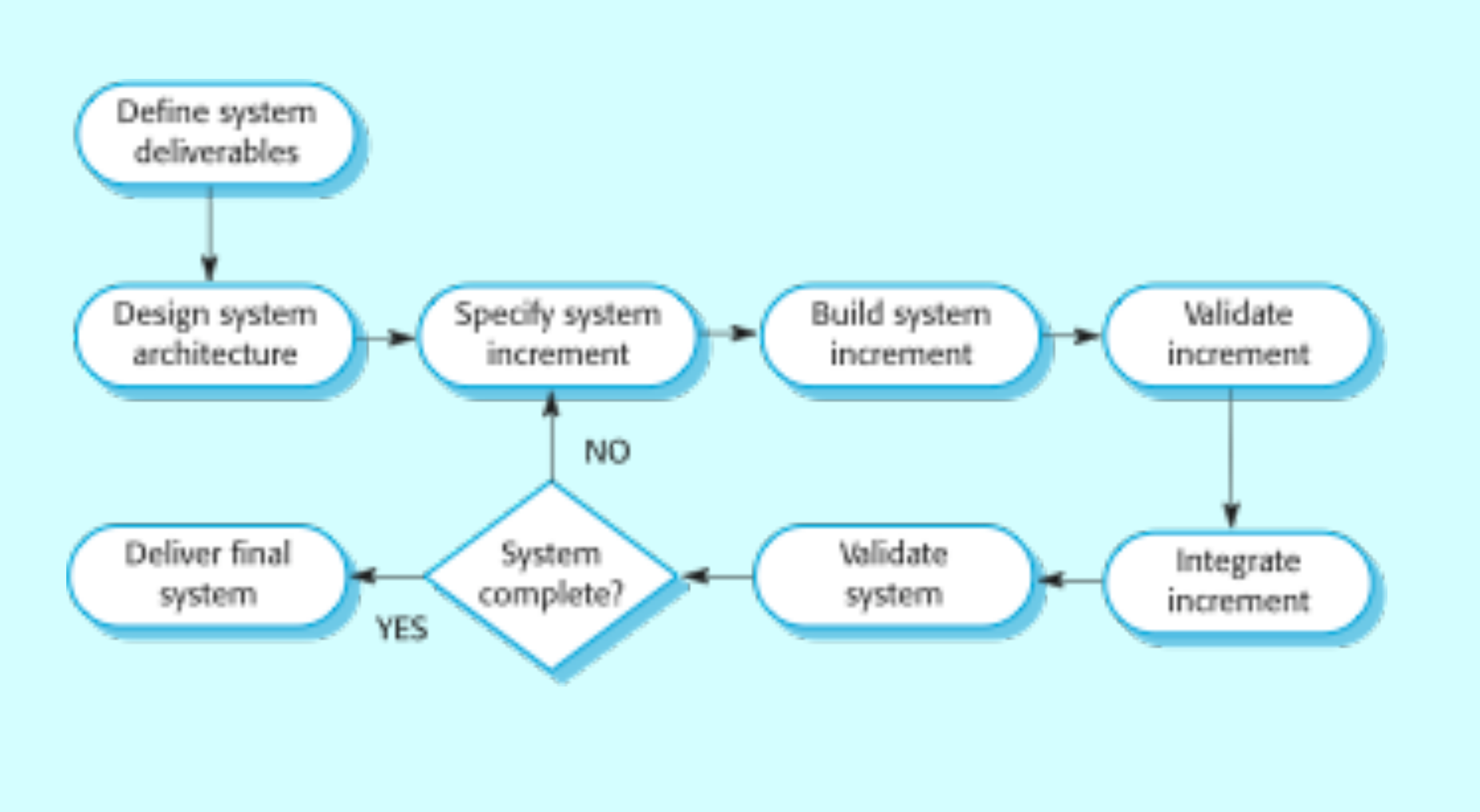
Requirements

- Because of the changing environment, it is often impossible to arrive at a stable, consistent set of system requirements.
- A waterfall model of development is impractical
- Iterative specification and delivery is the only way to deliver software quickly.

Characteristics of RAD processes

- The processes of specification, design and implementation are concurrent.
- There is no detailed specification and design documentation is minimized.
- The system is developed in a series of increments. End users evaluate each increment and make proposals for later increments.
- System user interfaces are usually developed using an interactive development system.

An iterative development process



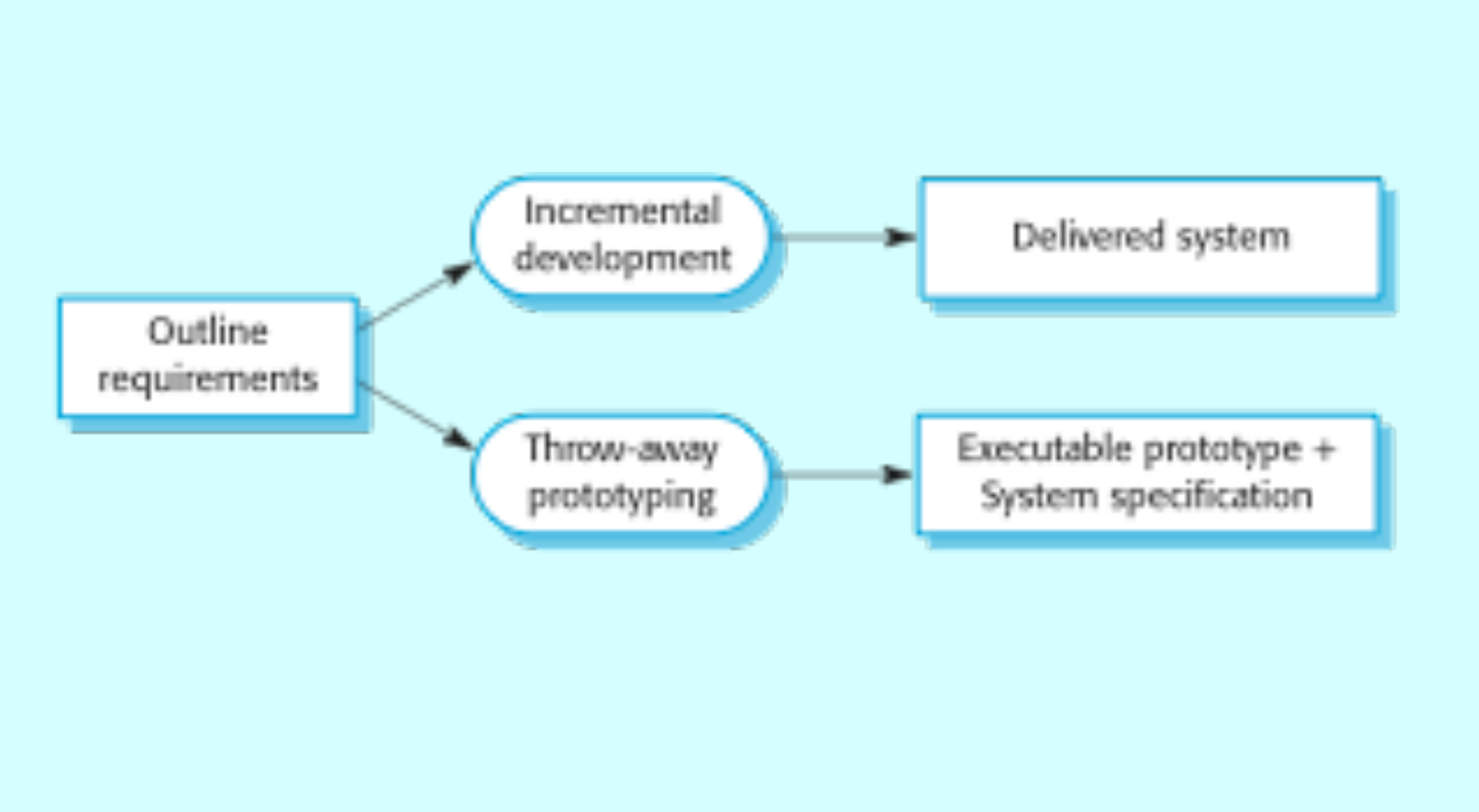
Advantages of incremental development

- Accelerated delivery of customer services. Each increment delivers the highest priority functionality to the customer.
- User engagement with the system. Users have to be involved in the development which means the system is more likely to meet their requirements and the users are more committed to the system.

Problems with incremental development

- Management problems
  - Progress can be hard to judge and problems hard to find because there is no documentation to demonstrate what has been done.
- Contractual problems
  - The normal contract may include a specification; without a specification, different forms of contract have to be used.
- Validation problems
  - Without a specification, what is the system being tested against?
- Maintenance problems
  - Continual change tends to corrupt software structure making it more expensive to change and evolve to meet new requirements.

Incremental development and prototyping



Conflicting objectives

- The objective of incremental development is to deliver a working system to end-users. The development starts with those requirements which are best understood.
- The objective of throw-away prototyping is to validate or derive the system requirements. The prototyping process starts with those requirements which are poorly understood.

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.

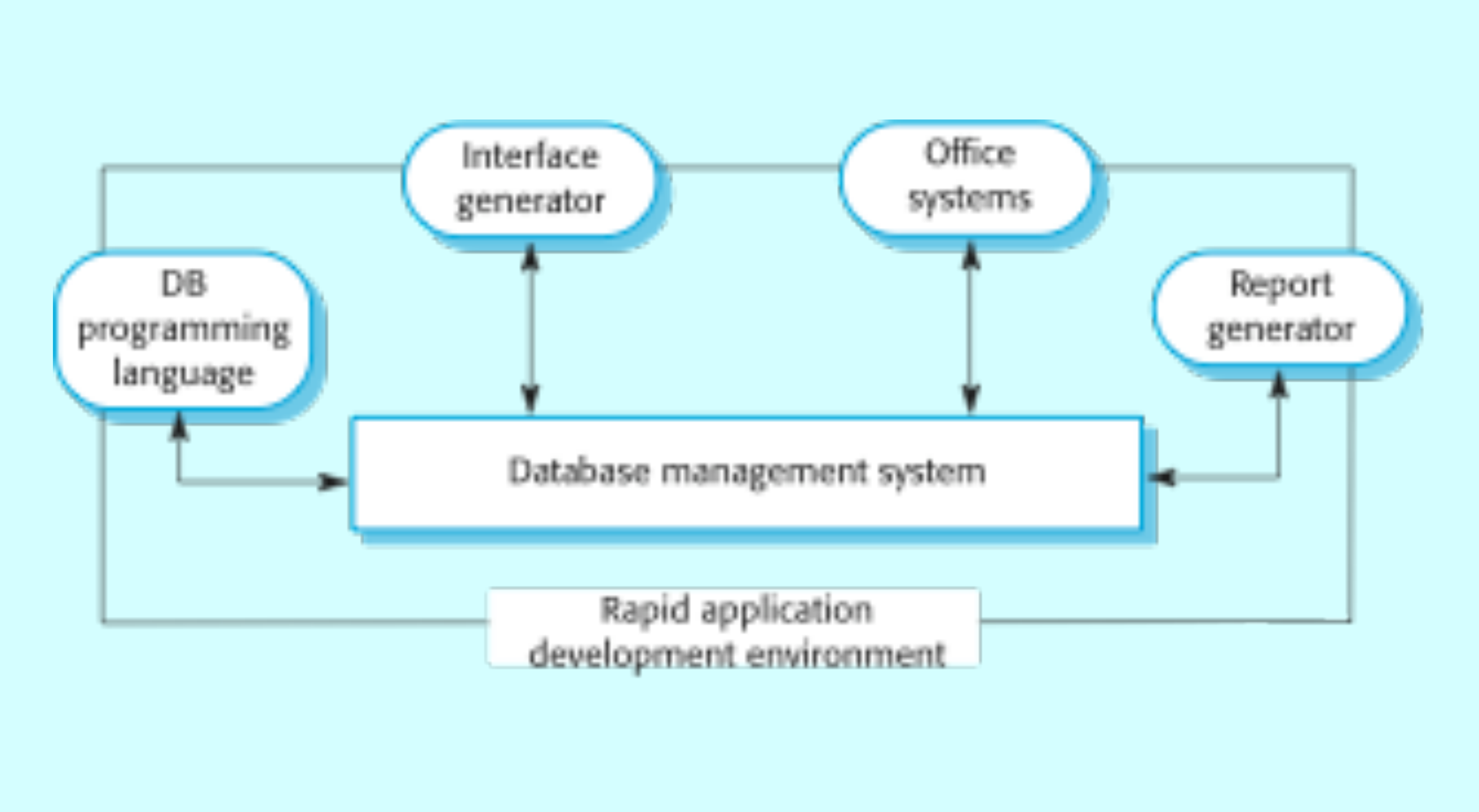
Throw-away prototypes

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

RAD environments

- RAD are designed to develop data-intensive business applications and rely on programming and presenting information from a database.
- Tools
    - Database programming language
    - Interface generator
    - Links to office applications
    - Report generators

A RAD environment



COTS reuse

- An effective approach to rapid development is to **configure and link existing off the shelf systems.**
- For example, a requirements management system could be built by using:
  - A database to store requirements;
  - A word processor to capture requirements and format reports;
  - A spreadsheet for traceability management;

Agile methods

- Dissatisfaction with the overheads involved in design methods led to the creation of agile methods. These methods:
    - Focus on the code rather than the design;
    - Are based on an iterative approach to software development;
    - Are intended to deliver working software quickly and evolve this quickly to meet changing requirements.
- Agile methods are probably best suited to small/ medium-sized business systems or PC products.

Principles of agile methods

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

Problems with agile methods

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

Extreme programming

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

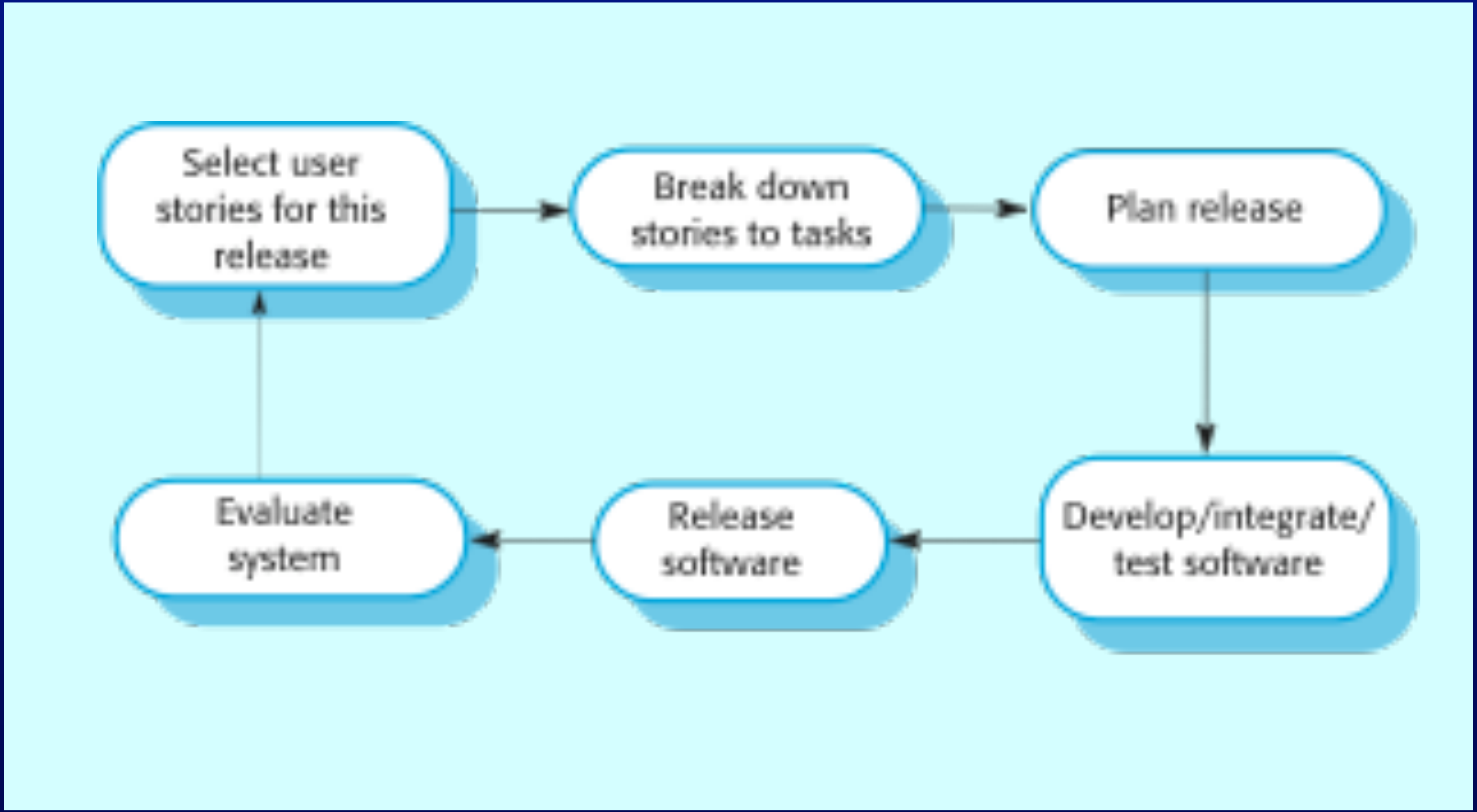
Extreme programming practices 1

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

Extreme programming practices 2

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

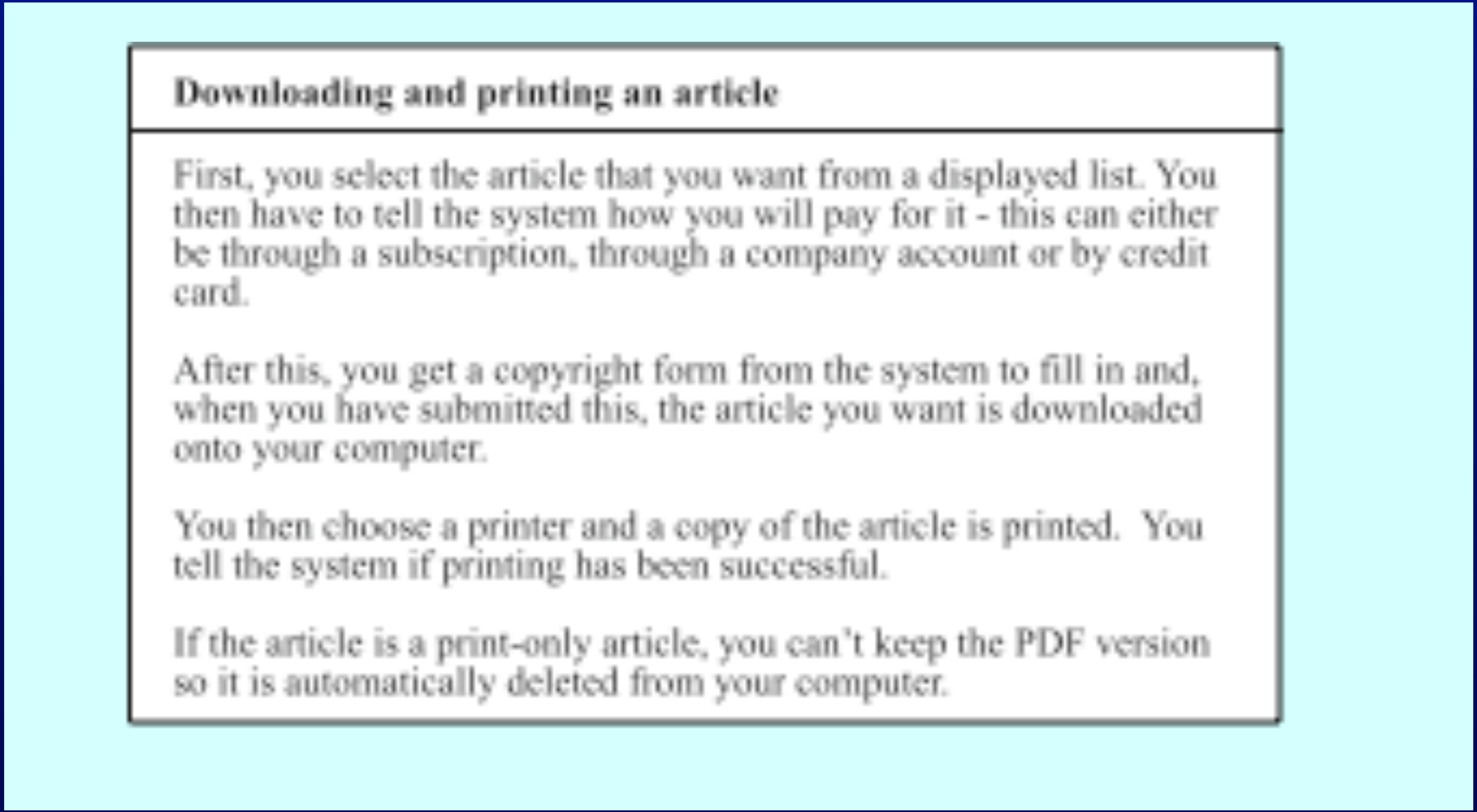
The XP release cycle


Select user stories for this release
Break down stories to tasks
Plan release
Develop/integrate/ test software
Release software
Evaluate system

Requirements scenarios

- In XP, user requirements are expressed as scenarios or user stories.
  - written on cards
  - break down into implementation tasks.
    - These tasks are the basis of schedule and cost estimates.
- The customer chooses the stories for inclusion in the next release based on their priorities and the schedule estimates.

Story card for document downloading



Downloading and printing an article

First, you select the article that you want from a displayed list. You then have to tell the system how you will pay for it - this can either be through a subscription, through a company account or by credit card.

After this, you get a copyright form from the system to fill in and, when you have submitted this, the article you want is downloaded onto your computer.

You then choose a printer and a copy of the article is printed. You tell the system if printing has been successful.

If the article is a print-only article, you can't keep the PDF version so it is automatically deleted from your computer.

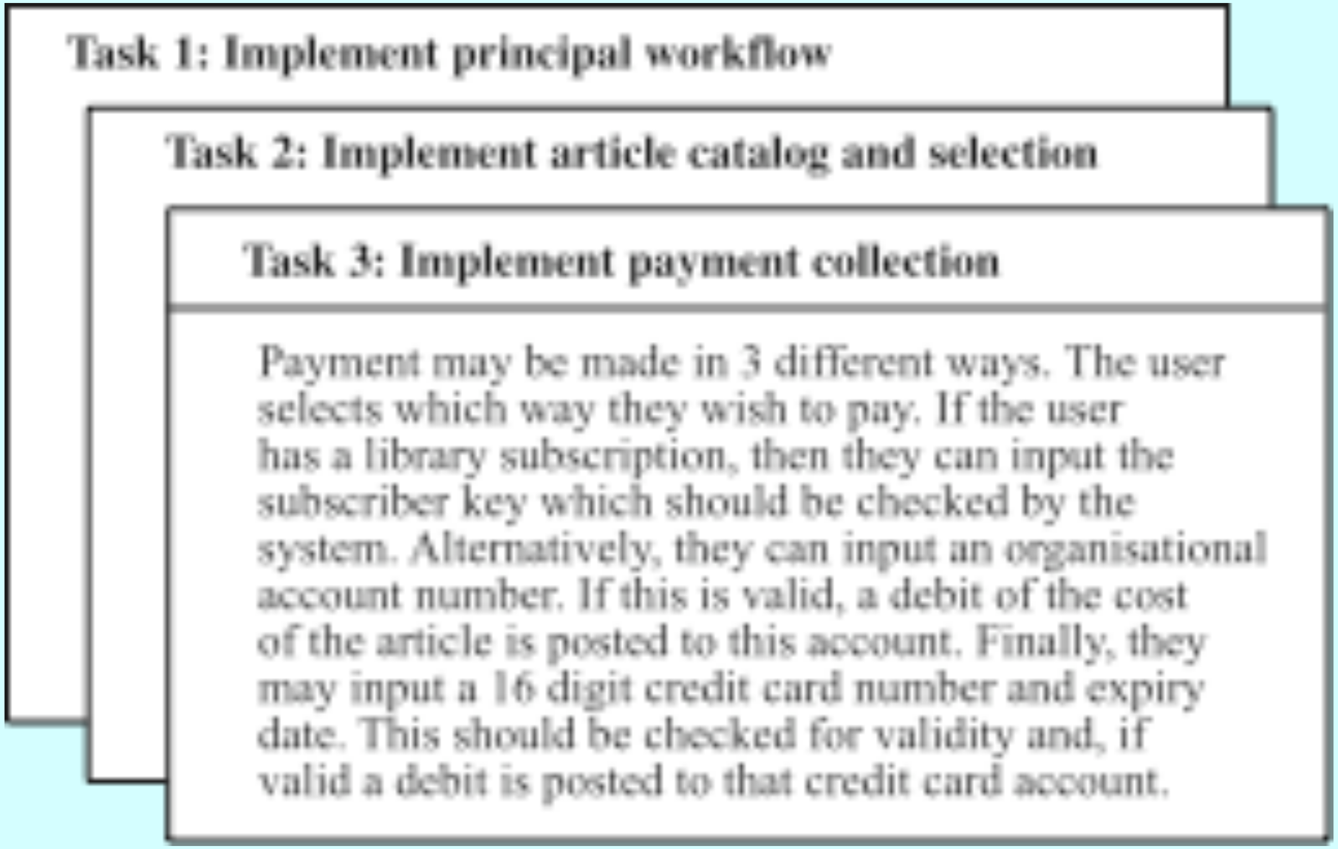
Testing in XP

- Test-first development.
- Incremental test development from scenarios.
- User involvement in test development and validation.
- Automated test harnesses are used to run all component tests each time that a new release is built.

Test-first development

- Writing tests before code clarifies the requirements to be implemented.
- Tests are written as programs
  - can be executed automatically.
- All previous and new tests are automatically run when new functionality is added.
  - Thus checking that the new functionality has not introduced errors.

Task cards for document downloading



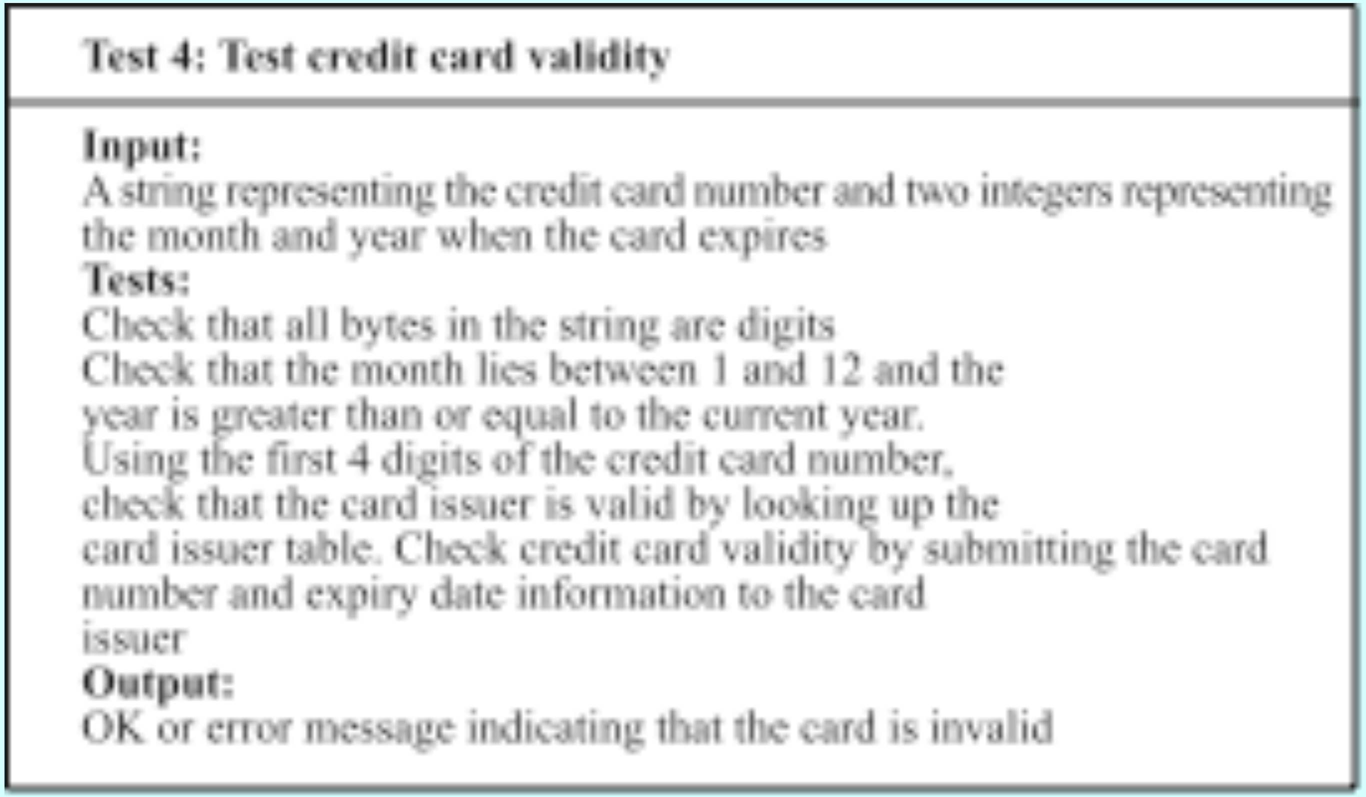
Task 1: Implement principal workflow

Task 2: Implement article catalog and selection

Task 3: Implement payment collection

Payment may be made in 3 different ways. The user selects which way they wish to pay. If the user has a library subscription, then they can input the subscriber key which should be checked by the system. Alternatively, they can input an organisational account number. If this is valid, a debit of the cost of the article is posted to this account. Finally, they may input a 16 digit credit card number and expiry date. This should be checked for validity and, if valid a debit is posted to that credit card account.

Test case description



Test 4: Test credit card validity

Input:
A string representing the credit card number and two integers representing the month and year when the card expires

Tests:
Check that all bytes in the string are digits
Check that the month lies between 1 and 12 and the year is greater than or equal to the current year.
Using the first 4 digits of the credit card number, check that the card issuer is valid by looking up the card issuer table. Check credit card validity by submitting the card number and expiry date information to the card issuer

Output:
OK or error message indicating that the card is invalid

Pair programming

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

Software Reuse

Software reuse

- Systems are often designed by composing existing components that have been used in other systems.
- *Systematic software reuse* may achieve better software, more quickly and at lower cost.

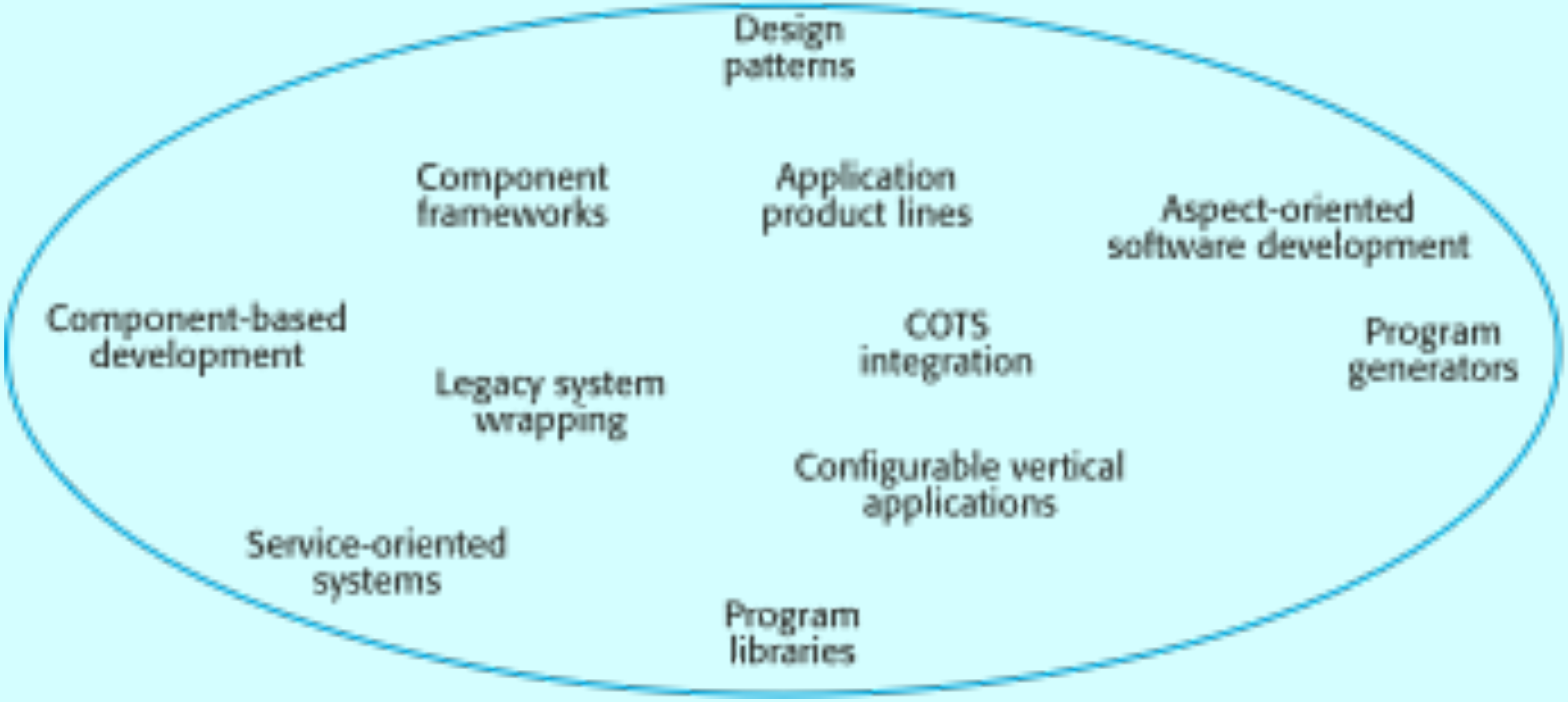
Reuse-based software engineering

- Application system reuse
  - The whole of an application system may be reused either by incorporating it without change into other systems (COTS reuse) or by developing application families.
- Component reuse
  - Components of an application from sub-systems to single objects may be reused.
- Object and function reuse
  - Software components that implement a single well-defined object or function may be reused.

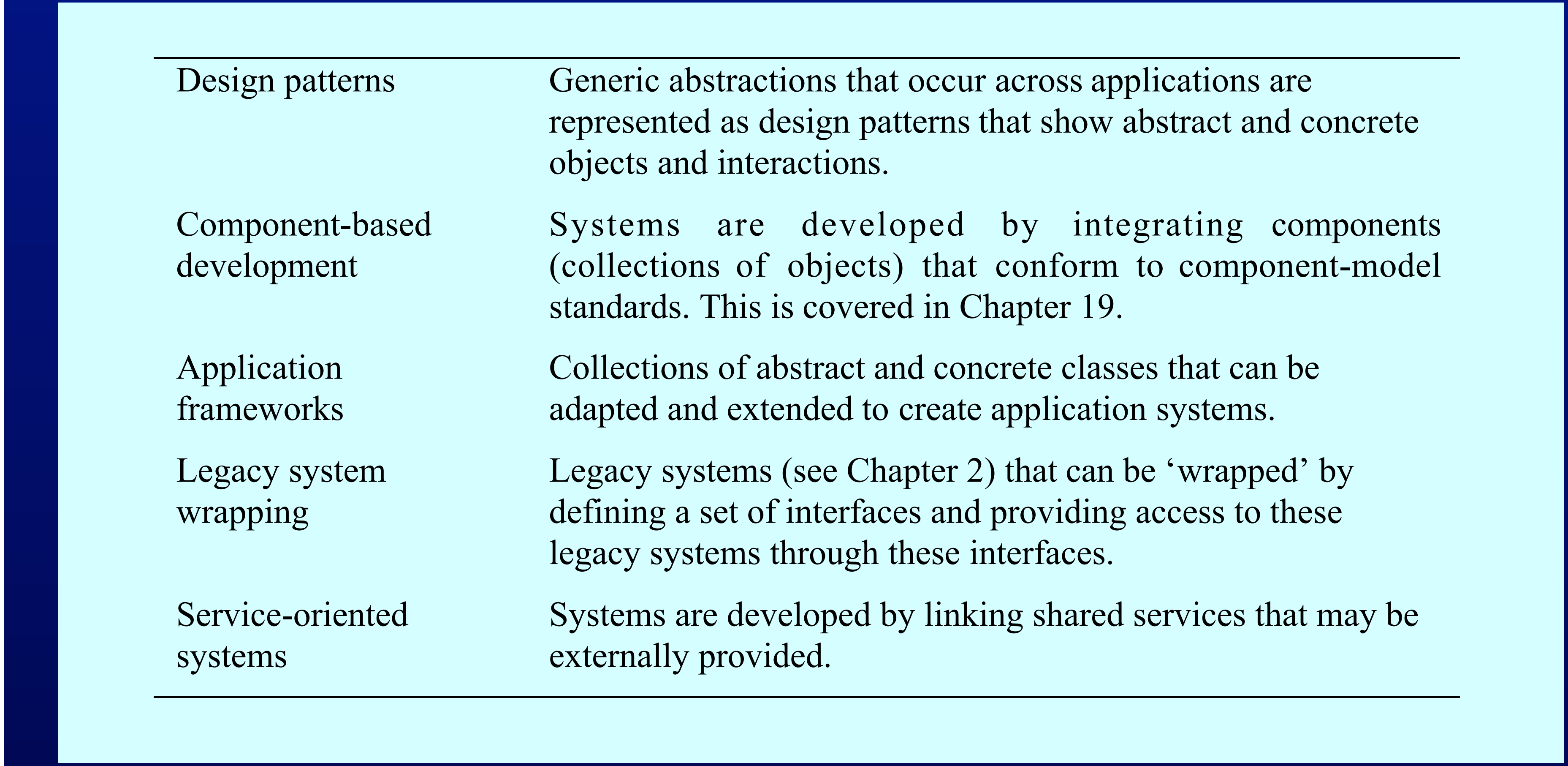
The reuse landscape

- There are many different approaches to reuse that may be used.
- Reuse is possible at a range of levels
  - from simple functions to complete application systems.

The reuse landscape



Reuse approaches 1



Design patterns	Generic abstractions that occur across applications are represented as design patterns that show abstract and concrete objects and interactions.
Component-based development	Systems are developed by integrating components (collections of objects) that conform to component-model standards. This is covered in Chapter 19.
Application frameworks	Collections of abstract and concrete classes that can be adapted and extended to create application systems.
Legacy system wrapping	Legacy systems (see Chapter 2) that can be ‘wrapped’ by defining a set of interfaces and providing access to these legacy systems through these interfaces.
Service-oriented systems	Systems are developed by linking shared services that may be externally provided.

Reuse approaches 2

Application product lines	An application type is generalised around a common architecture so that it can be adapted in different ways for different customers.
COTS integration	Systems are developed by integrating existing application systems.
Configurable vertical applications	A generic system is designed so that it can be configured to the needs of specific system customers.
Program libraries	Class and function libraries implementing commonly-used abstractions are available for reuse.
Program generators	A generator system embeds knowledge of a particular types of application and can generate systems or system fragments in that domain.
Aspect-oriented software development	Shared components are woven into an application at different places when the program is compiled.

Concept reuse

- Follow the design decisions made by the original developer of the component.
- Main approaches:
  - Design patterns;
  - Generative programming.

Design patterns

- A design pattern is a way of reusing abstract knowledge about a problem and its solution.
- A pattern is a description of the problem and the essence of its solution.
- It should be sufficiently abstract to be reused in different settings.
- Patterns often rely on object characteristics such as inheritance and polymorphism.

Types of program generator

- Program generators involve the reuse of standard patterns and algorithms.
  - A program is then automatically generated.
- Types of program generator
  - Application generators for business data processing;
  - Parser and lexical analyser generators for language processing;
  - Code generators in CASE tools.
- Generator-based reuse is
  - very cost-effective
  - applicability is limited to a small number of appl. domains.
  - easier for end-users (w.r.t. component-based approaches)

Application frameworks

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

Application system reuse

- Involves the reuse of entire application systems
  - by configuring a system
  - by integrating two or more systems
- Two approaches covered here:
  - COTS product integration;
  - Product line development.

COTS product reuse

- COTS - Commercial Off-The-Shelf systems.
- COTS systems are usually complete application systems that offer an API (Application Programming Interface).
- Building large systems by integrating COTS systems is now a viable development strategy for some types of system such as E-commerce systems.
- The key benefit is faster application development and, usually, lower development costs.

COTS system integration problems

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

Software product lines

- Software product lines or application families
  are applications with generic functionality
  - can be adapted and configured
- Adaptation may involve:
  - Component and system configuration;
  - Adding new components to the system;
  - Selecting from a library of existing components;
  - Modifying components to meet new requirements.

ERP systems

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

Component-based development

Component-based development

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

CBSE essentials

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

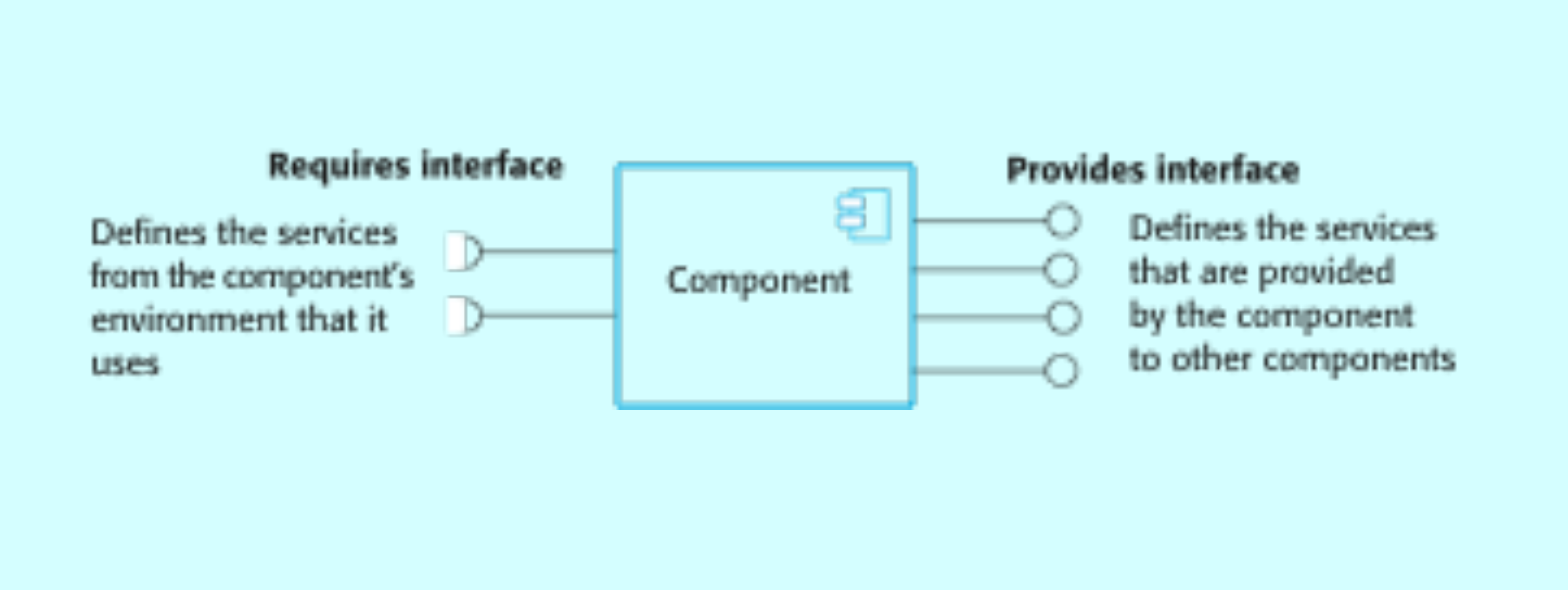
CBSE and design principles

- Apart from the benefits of reuse, CBSE is based on sound software engineering design principles:
  - Components are independent so do not interfere with each other;
  - Component implementations are hidden;
  - Communication is through well-defined interfaces;
  - Component platforms are shared and reduce development costs.

Components

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

Component interfaces



Requires interface
Defines the services from the component's environment that it uses
Component
Provides interface
Defines the services that are provided by the component to other components

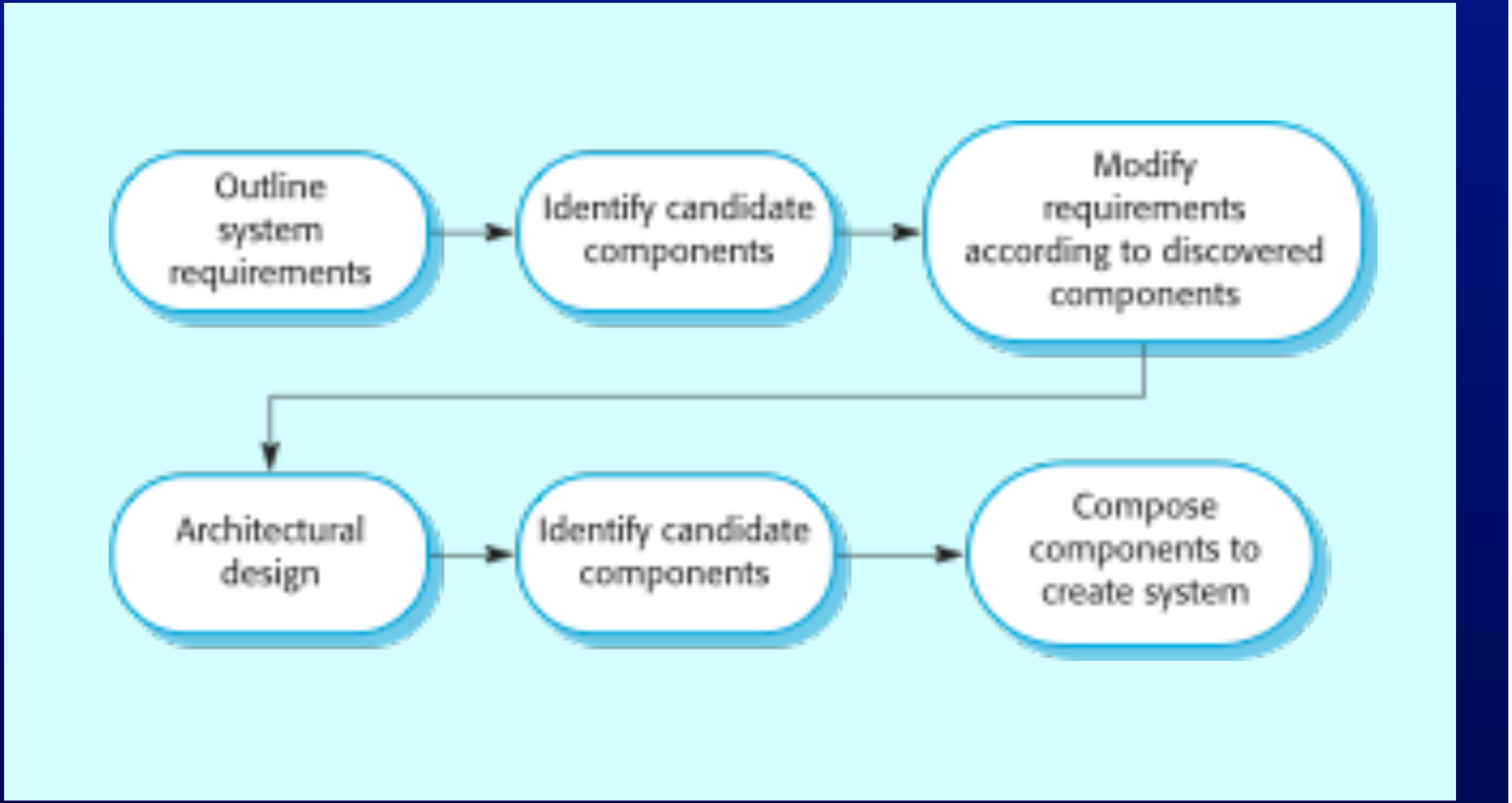
Components and objects

- Components are deployable entities.
- Components do not define types.
- Component implementations are opaque.
- Components are language-independent.
- Components are standardized.

Component models

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

The CBSE process

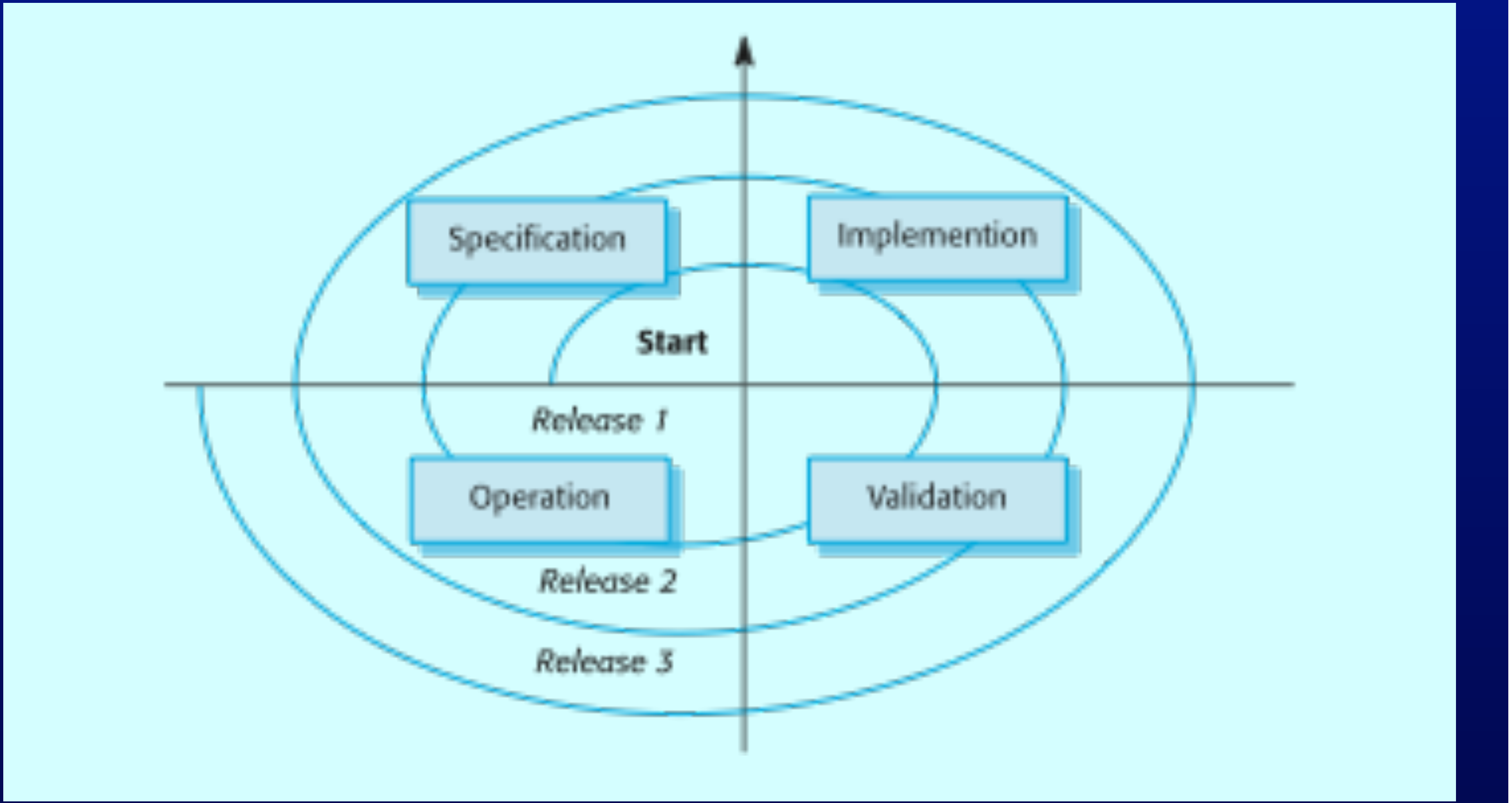

Outline system requirements
Identify candidate components
Modify requirements according to discovered components
Architectural design
Identify candidate components
Compose components to create system

Software evolution

Software change

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

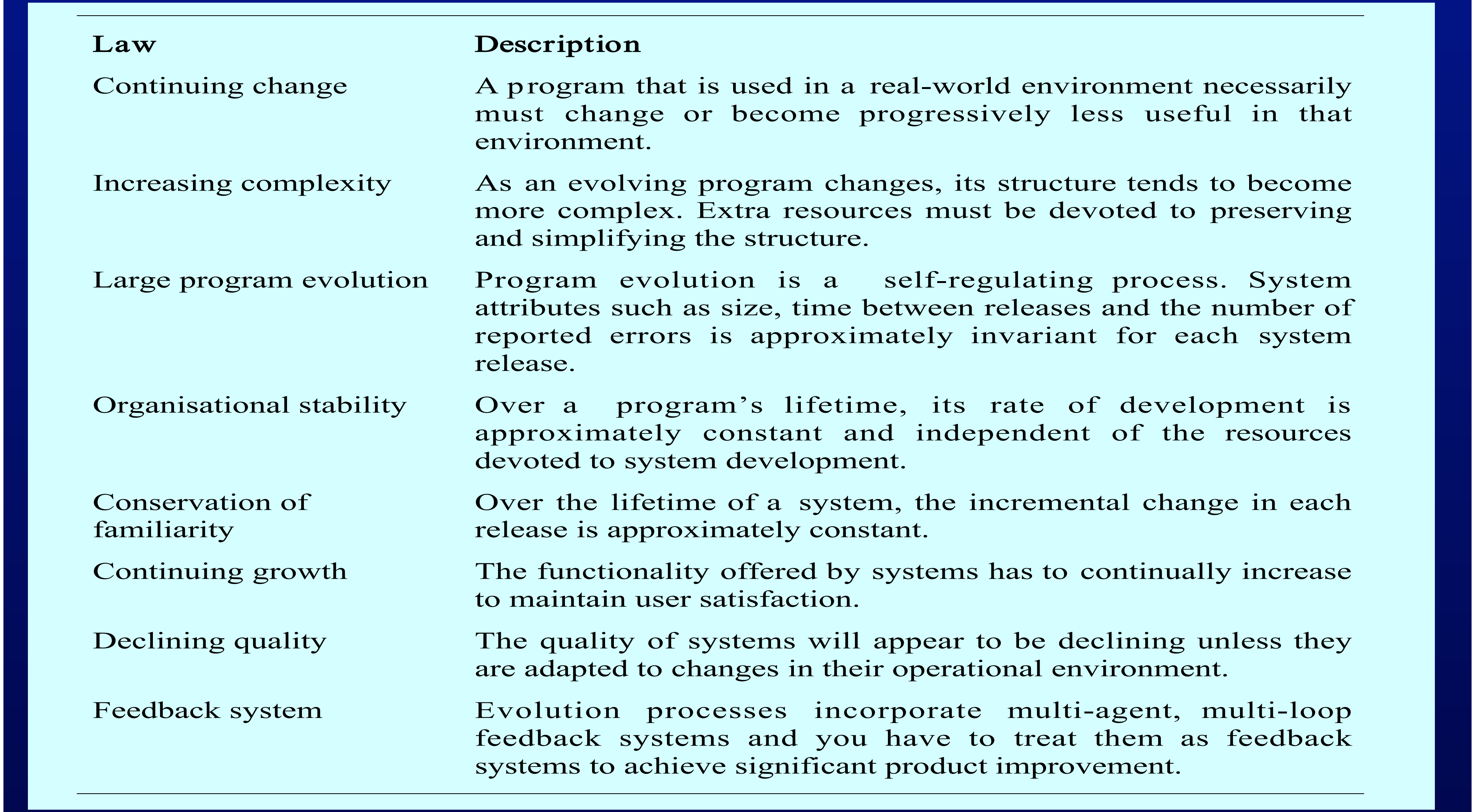
Spiral model of evolution



Program evolution dynamics

- Program evolution dynamics is the study of the processes of system change.
- Lehman and Belady proposed a number of ‘laws’
  - Deducted after major empirical studies
  - Applied to all systems as they evolved.
  - There are sensible observations rather than laws.
  - Applicable to large systems developed by large organisations. Perhaps less applicable in other cases.

Lehman’ s laws

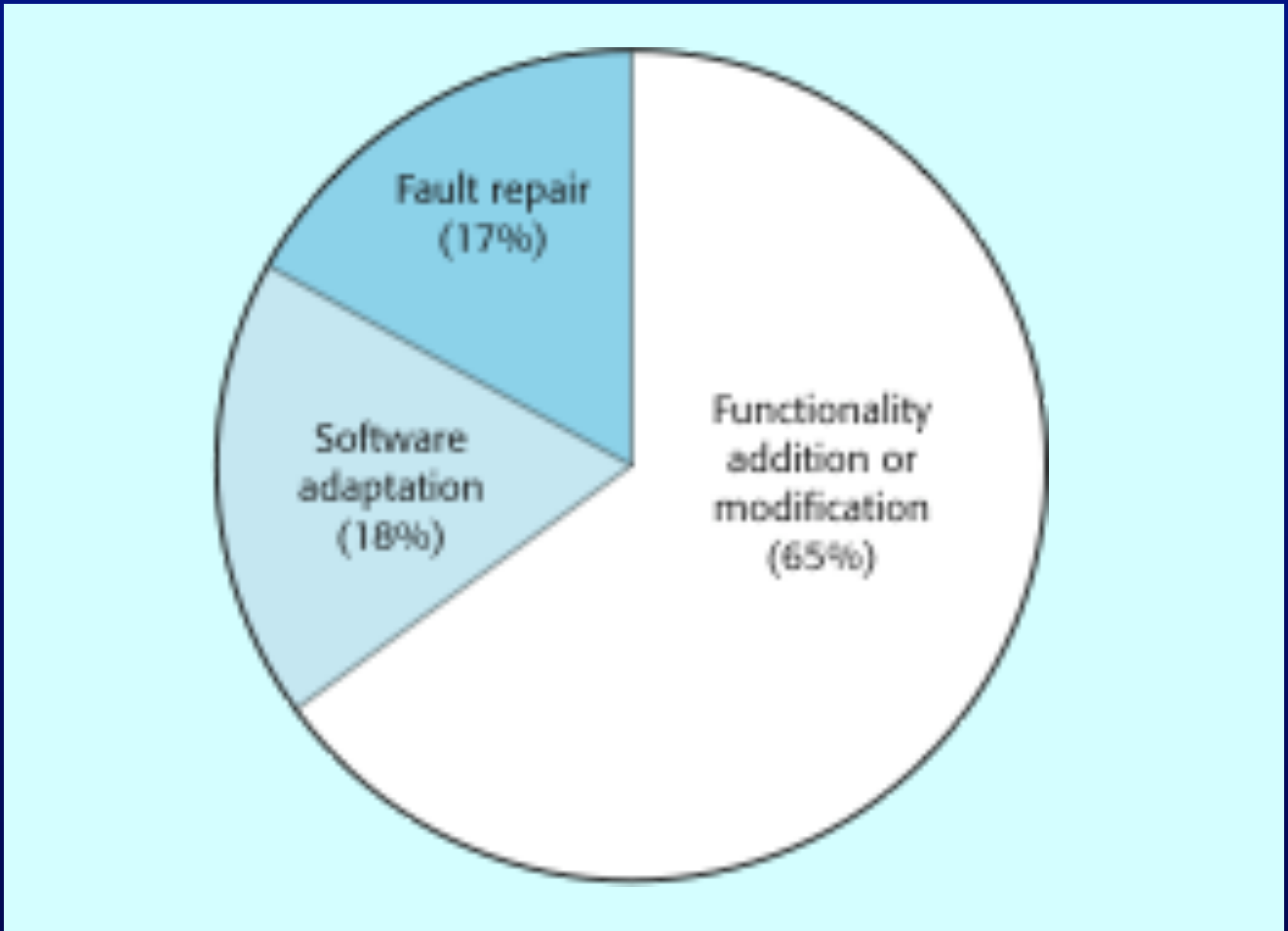


Law	Description
Continuing change	A program that is used in a real-world environment necessarily must change or become progressively less useful in that environment.
Increasing complexity	As an evolving program changes, its structure tends to become more complex. Extra resources must be devoted to preserving and simplifying the structure.
Large program evolution	Program evolution is a self-regulating process. System attributes such as size, time between releases and the number of reported errors is approximately invariant for each system release.
Organisational stability	Over a program’s lifetime, its rate of development is approximately constant and independent of the resources devoted to system development.
Conservation of familiarity	Over the lifetime of a system, the incremental change in each release is approximately constant.
Continuing growth	The functionality offered by systems has to continually increase to maintain user satisfaction.
Declining quality	The quality of systems will appear to be declining unless they are adapted to changes in their operational environment.
Feedback system	Evolution processes incorporate multi-agent, multi-loop feedback systems and you have to treat them as feedback systems to achieve significant product improvement.

Software maintenance

- Modifying a program after it has been put into use.
- Maintenance does not normally involve major changes to the system’s architecture.
- Changes are implemented by modifying existing components and adding new components to the system.
- Maintenance is inevitable
  - The environment is changing => requirements change
  - Systems MUST be maintained therefore if they are to remain useful in an environment.

Distribution of maintenance effort



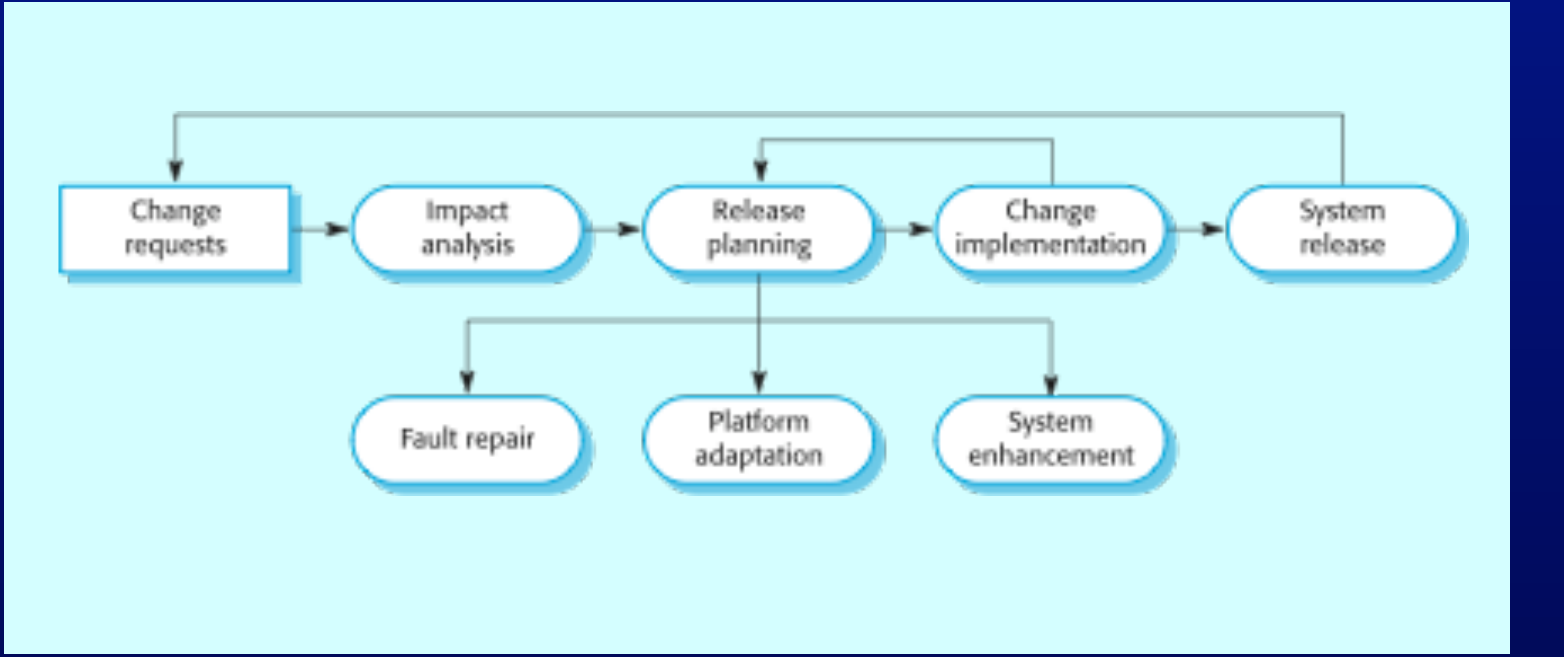
Maintenance cost factors

- Team stability
  - Maintenance costs are reduced if the same staff are involved with them for some time.
- Contractual responsibility
  - The developers of a system may have no contractual responsibility for maintenance so there is no incentive to design for future change.
- Staff skills
  - Maintenance staff are often inexperienced and have limited domain knowledge.
- Program age and structure
  - As programs age, their structure is degraded and they become harder to understand and change.

Evolution processes

- Evolution processes depend on
  - The type of software being maintained;
  - The development processes used;
  - The skills and experience of the people involved.
- Proposals for change are the driver for system evolution. Change identification and evolution continue throughout the system lifetime.

The system evolution process



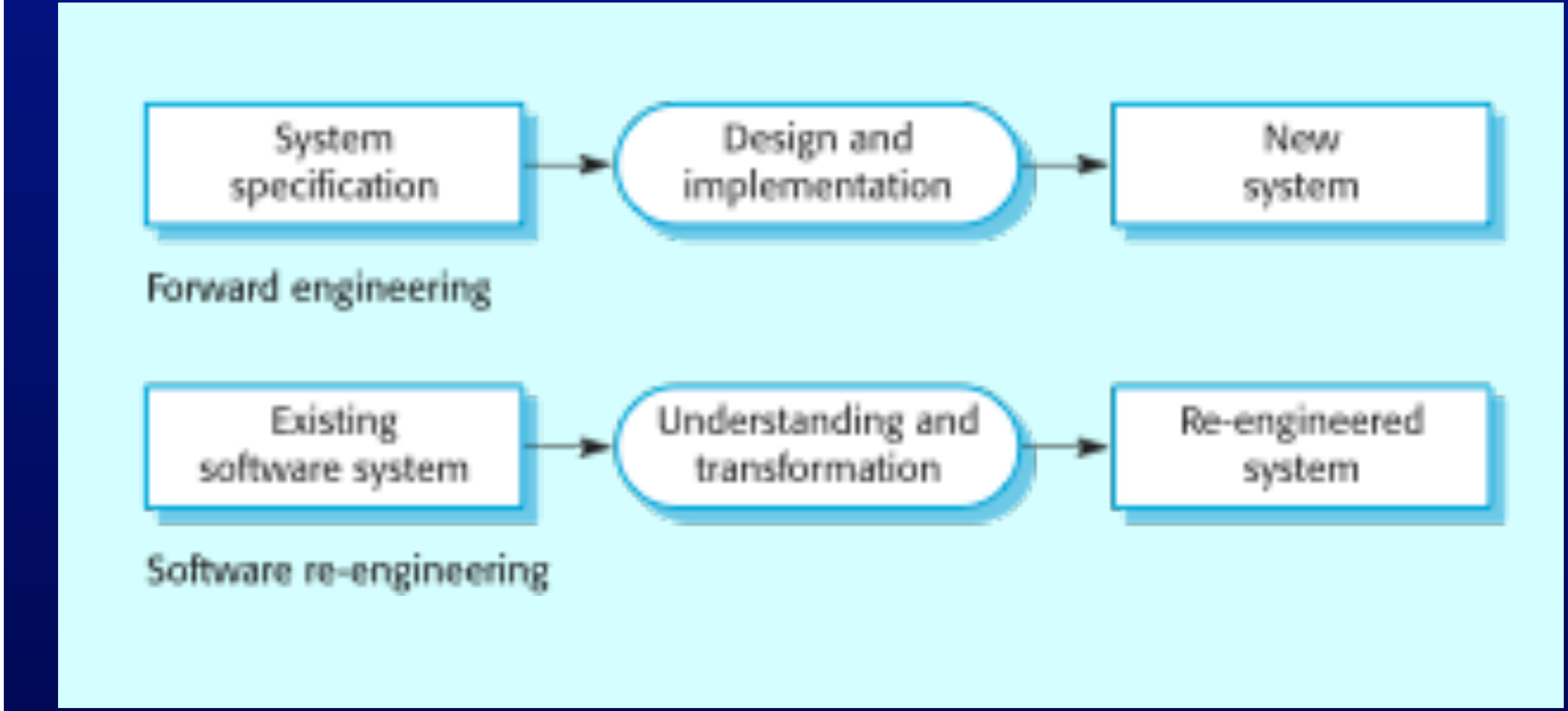
System re-engineering

- Re-structuring or re-writing part or all of a legacy system without changing its functionality.
- Applicable where some but not all sub-systems of a larger system require frequent maintenance.
- Re-engineering involves adding effort to make them easier to maintain. The system may be re-structured and re-documented.

Advantages of reengineering

- Reduced risk
  - There is a high risk in new software development. There may be development problems, staffing problems and specification problems.
- Reduced cost
  - The cost of re-engineering is often significantly less than the costs of developing new software.

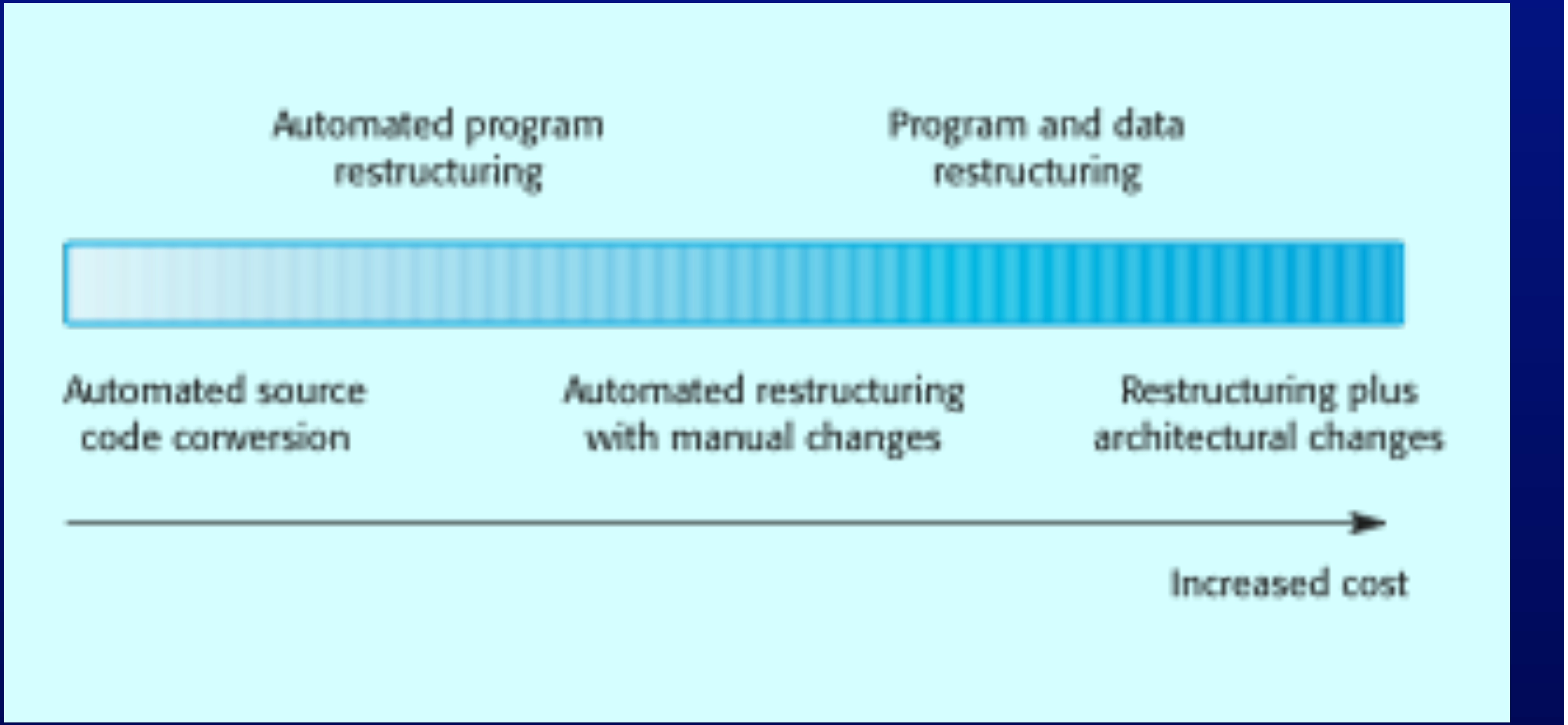
Forward and re-engineering



Reengineering process activities

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

Re-engineering approaches



Automated program restructuring

Program and data restructuring

Automated source code conversion

Automated restructuring with manual changes

Restructuring plus architectural changes

Increased cost

Legacy system evolution

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

Legacy system categories

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