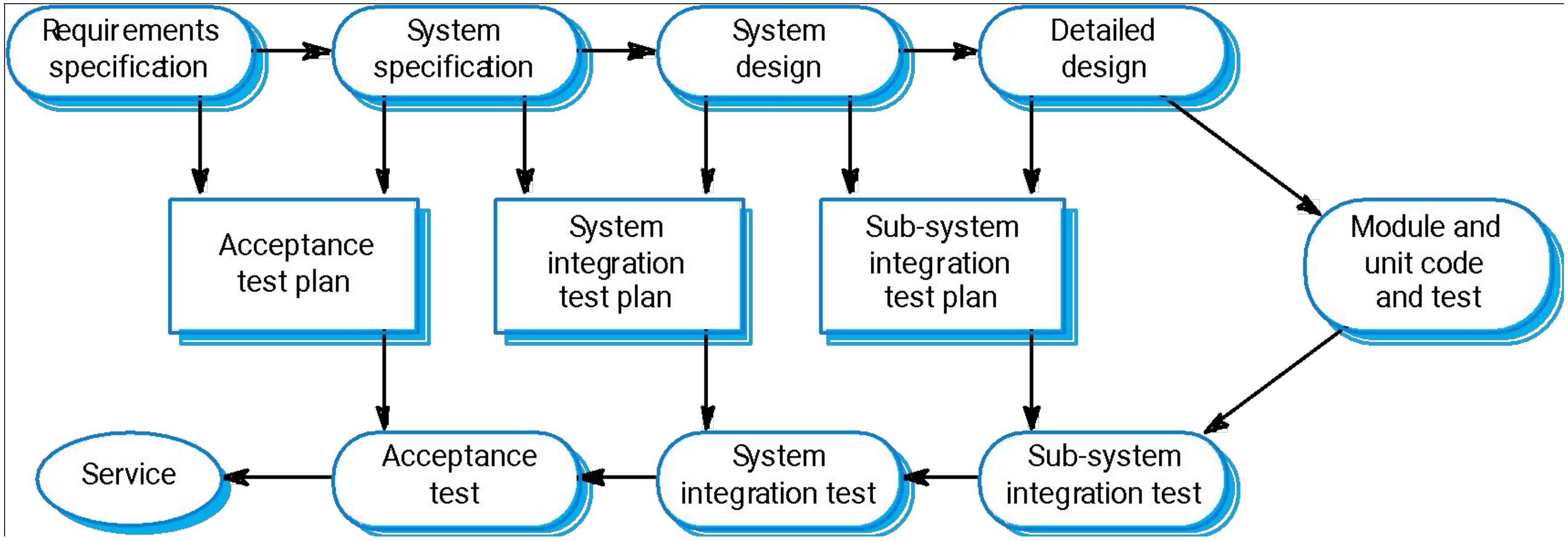
CSC4311: SOFTWARE ENGINEERING

First Semester 2023/2024 Academic Session

Software reuse

Conventional Software Engineering


Requirements specification
System specification
System design
Detailed design
Acceptance test plan
System integration test plan
Sub-system integration test plan
Module and unit code and test
Service
Acceptance test
System integration test
Sub-system integration test

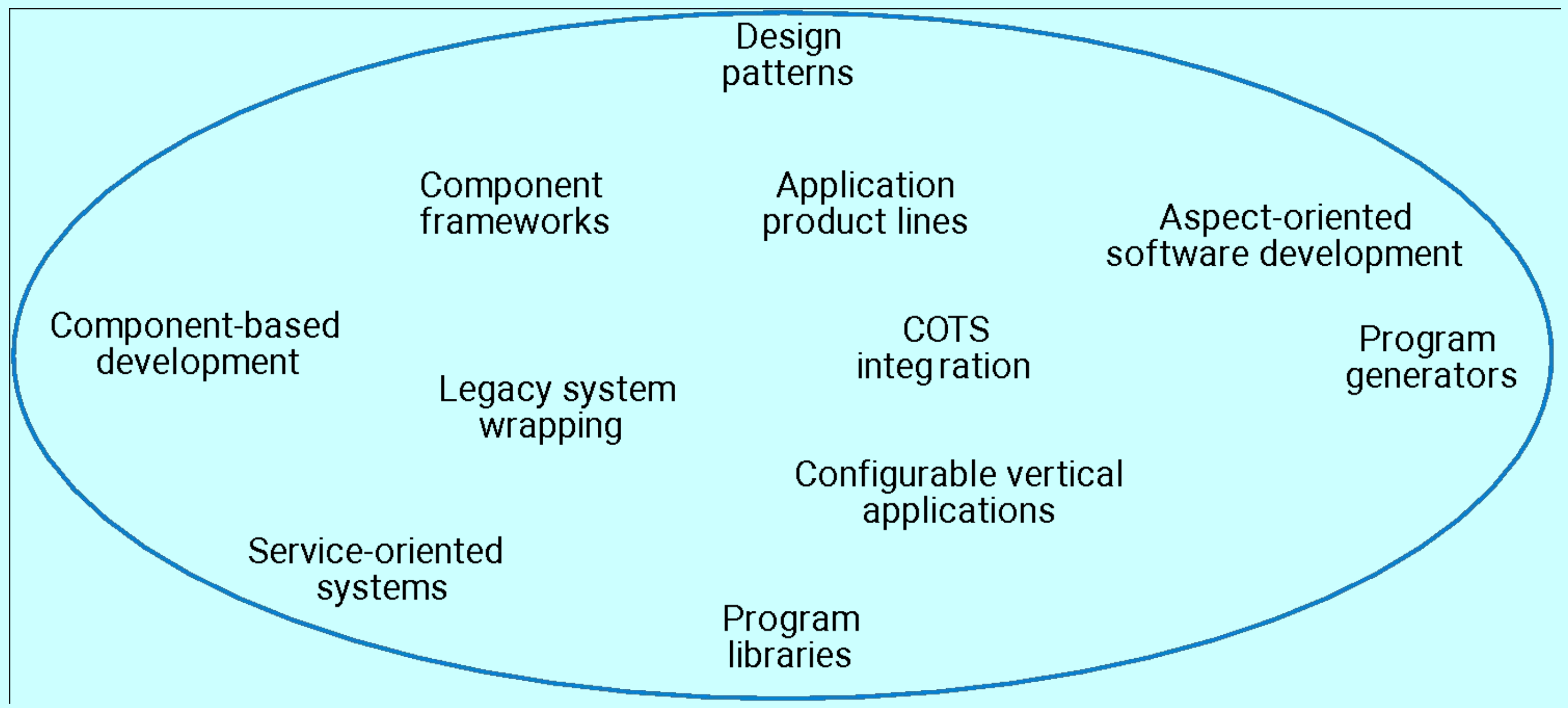
Software reuse

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

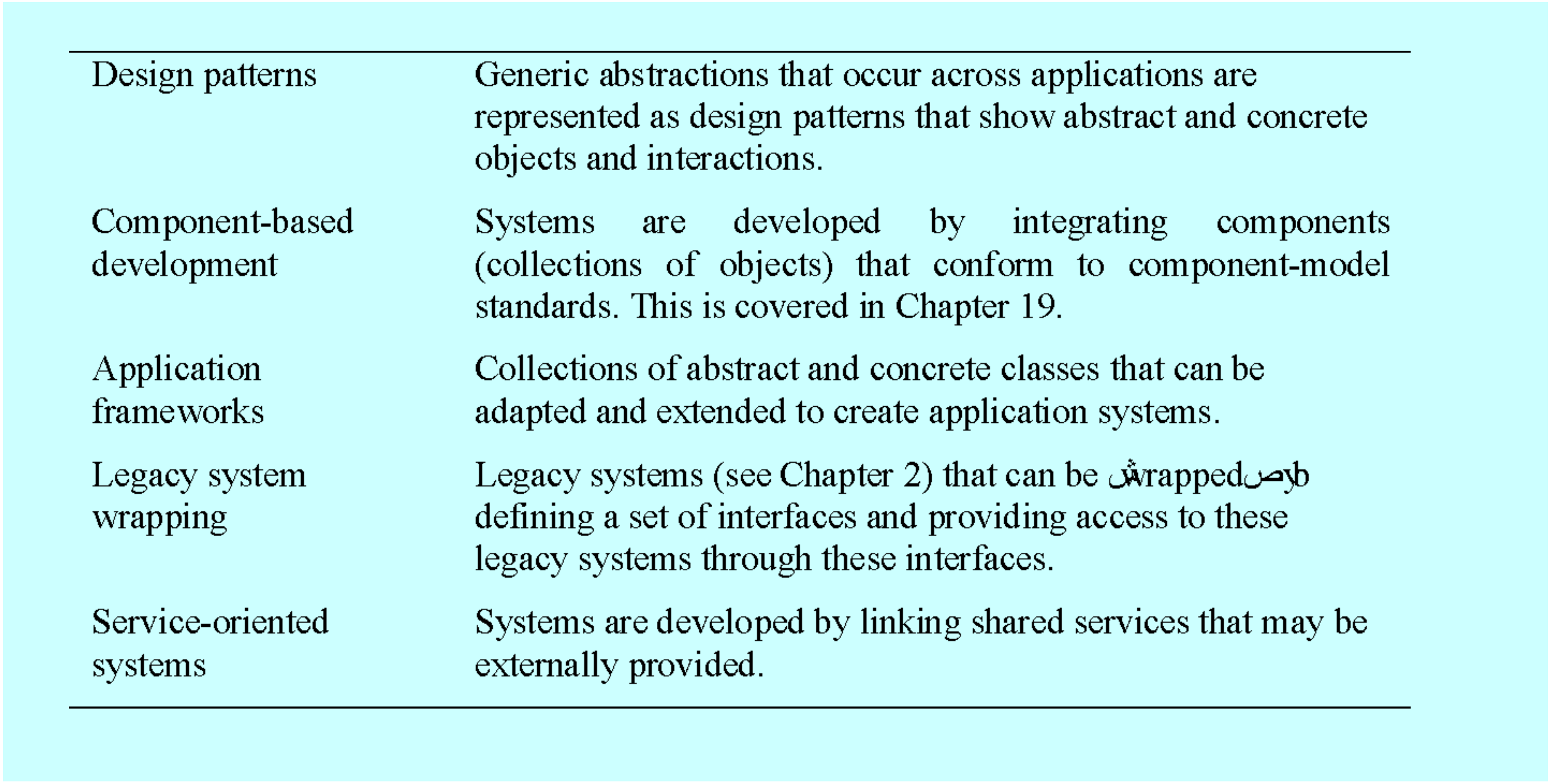
Scope of reuse

- Fine-grained reuse:
  - involves reusing small, specific functionality within a system. This could include functions, classes, or even individual lines of code
- Coarse-grained reuse
  - involves reusing larger, more comprehensive components or systems, such as entire libraries, frameworks, or even entire applications.

Reuse Landscape

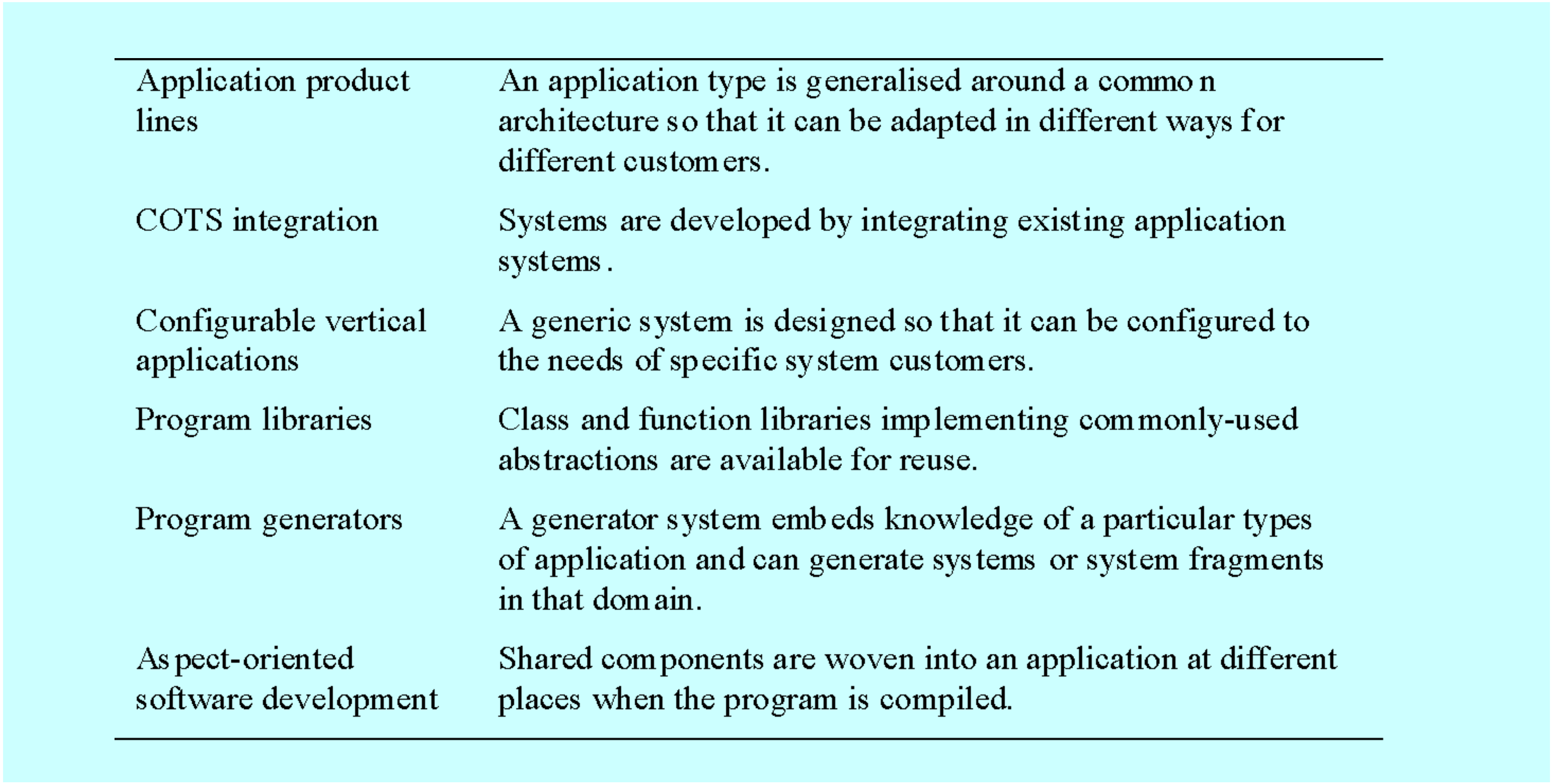


Reuse approaches 1



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 شwrappedطصb 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.

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.

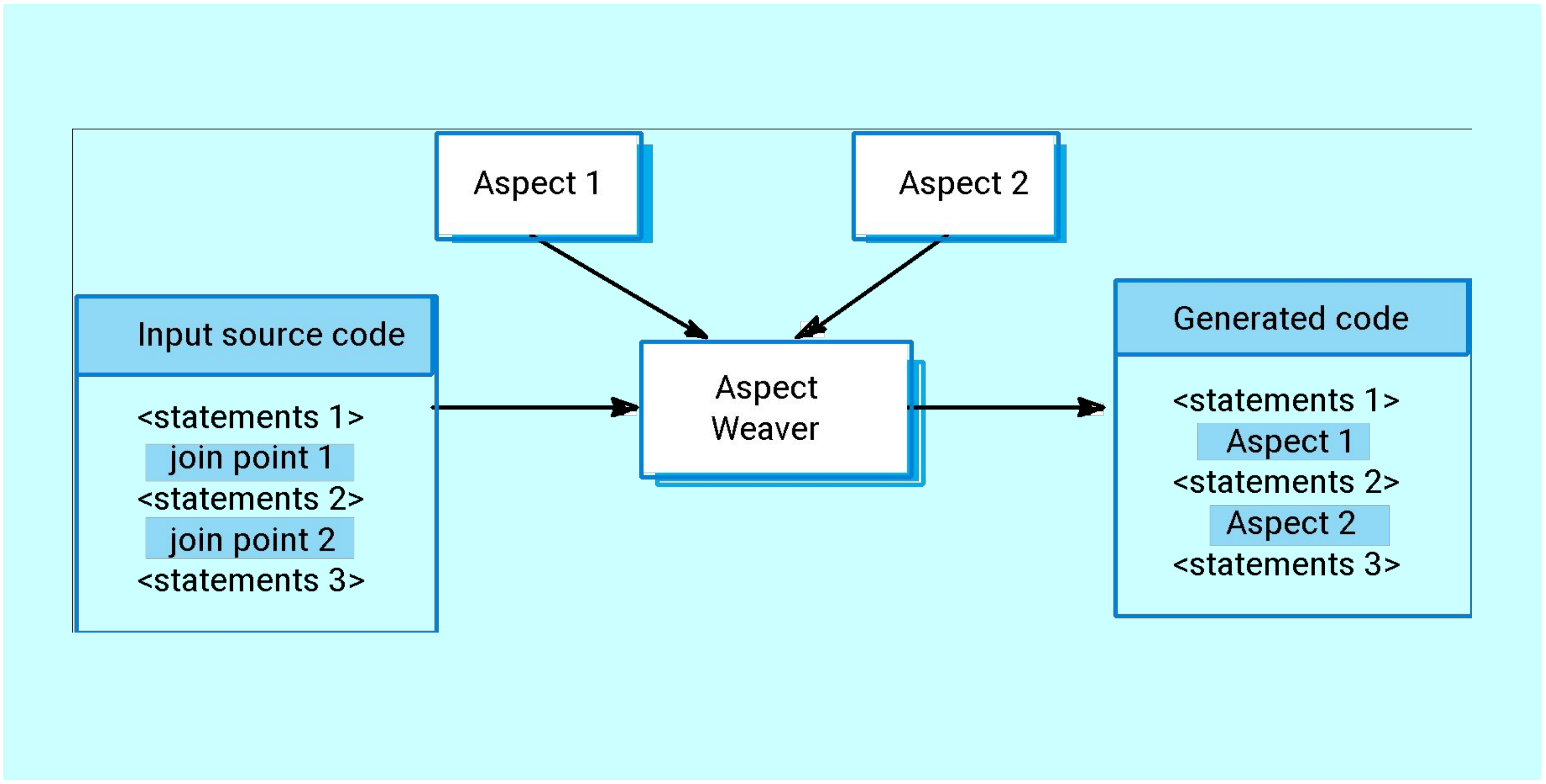
Generator-based reuse

- Program generators involve the reuse of standard patterns and algorithms.
- These are embedded in the generator and parameterised by user commands. A program is then automatically generated.
- Generator-based reuse is possible when domain abstractions and their mapping to executable code can be identified.
- A domain specific language is used to compose and control these abstractions.

Aspect-oriented development

- Aspect-oriented development addresses a major software engineering problem - the separation of concerns.
- Concerns are often not simply associated with application functionality but are cross-cutting - e.g. all components may monitor their own operation, all components may have to maintain security, etc.
- Cross-cutting concerns are implemented as aspects and are woven into a program. The concern code is reuse and the new system is generated by the aspect weaver.

Aspect-oriented development



Framework

- A framework is a generic skeletal application or structure that can be adapted to create a specific application
- It may consist of concrete classes, abstract classes, and interfaces

White-box vs black-box framework

White-box:

- the framework contains many abstract classes and therefore requires sub-classing

Black-box:

- The framework contains ready-made components that can be adapted

Grey box:

- Combination of the two above

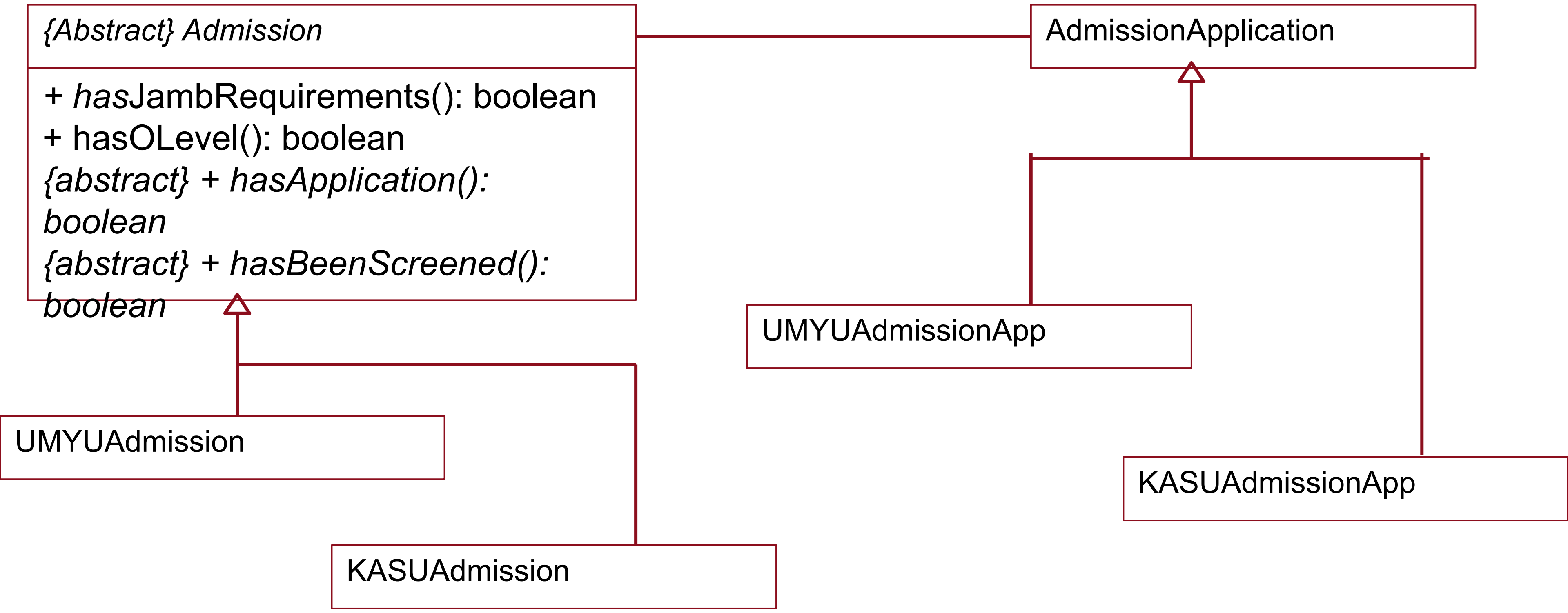
Template Method Pattern

- The Template Method Pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses.
- Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure.
- Frameworks (especially white-box) are often implemented using Template Method Pattern

Template Method Pattern example

- Let’s say we are creating a mini framework for processing candidates’ admission into universities.
- The framework can contain concrete implementation of checking JAMB and O-level requirements.
- The framework may also define abstract process of requirement for separate application and Post-UTME screening/test
- The framework will then have to be customized for a specific university.

Template Method Pattern example



Abstract Admission

```
public abstract class Admmission {
    public boolean hasJambrequirements() {
        checkJambScore();
        checkJambCombination();
        return false;    }
    public boolean checkOlevel(){
//detail implementation
        return false;    }
    public abstract boolean hasApplication();
    public abstract boolean hasBeenScreened();
    private boolean checkJambScore() {
//details implementation
        return false;
    }
    private boolean checkJambCombination() {
//detailed implementation
        return false;
    }}
```

KASUAdmission

```
public class KASUAdmission extends Admission {
    @Override
    public boolean hasApplication() {
        boolean hasApplication =false;
     return hasApplication ;
    }
    @Override
    public boolean hasBeenScreened() {
        float postUtmeScore =0;
        if (postUtmeScore < 40){
       return false;
        }
        else {
            return true;
        }
    }

}
```

UmyuAdmission

```
public class UmyuAdmission extends Admission {

    @Override
    public boolean hasApplication() {
      return true;
    }

    @Override
    public boolean hasBeenScreened() {
       boolean academicOfficeCertified = false;
       return academicOfficeCertified;
    }

}
```

UmyuAdmissionApp

```
public class AdmissionApp {

    public static  boolean hasApplication = false;
static void main(String[] args) {
        Admission a = new UmyuAdmission ();
        a.checkJambrequirements();
        a.checkOlevel();
        if (hasApplication){
        a.hasApplication();
        }
        a.hasBeenScreened();
    }

}
```

Software product lines

- Software product lines or application families are applications with generic functionality that can be adapted and configured for use in a specific context.

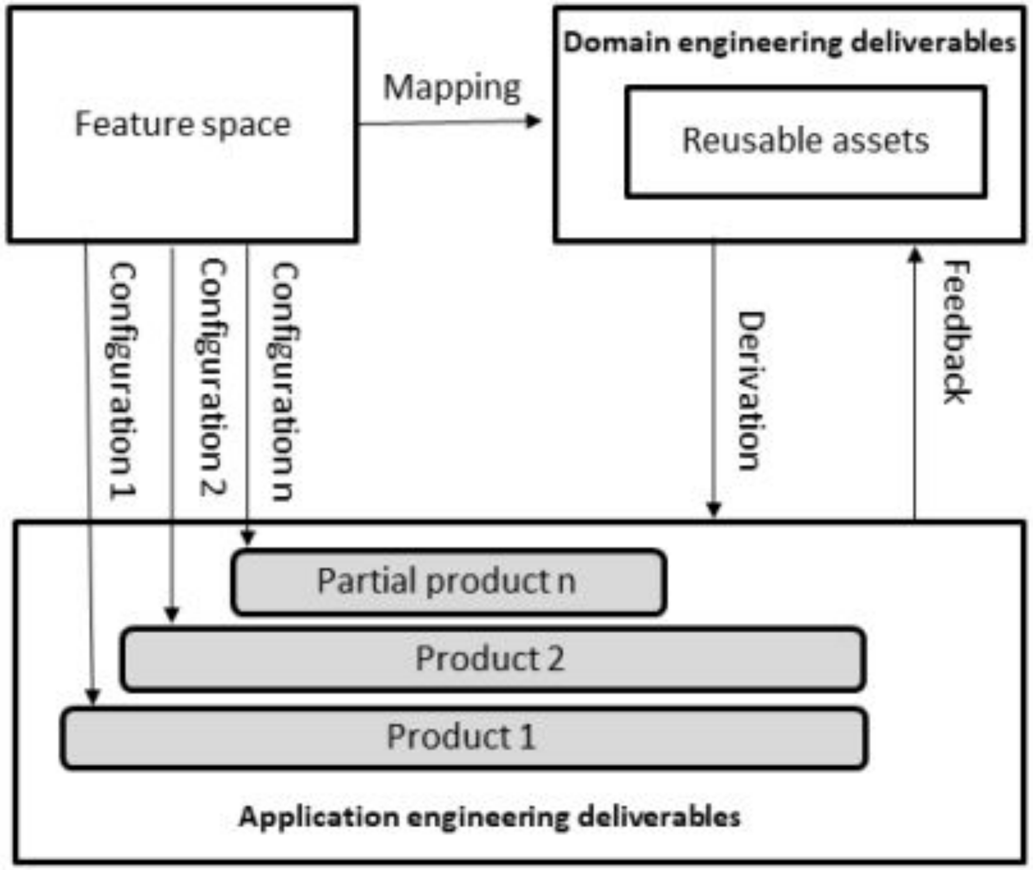
Adaptation may involve:

- Component and system configuration;
- Adding new components to the system;
- Selecting from a library of existing components;
- Modifying components to meet new requirements.

Feature-oriented Software Product Line (SPLE)

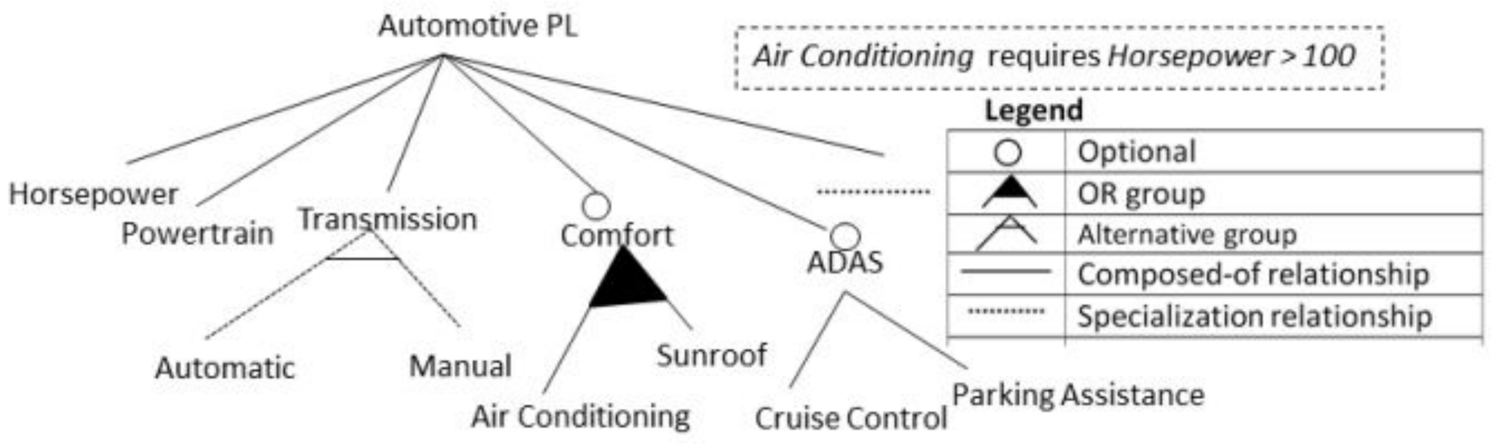
- Throughout the history of software engineering, X-oriented may be used to describe an engineering activity if X is the dominant technique, concept or an abstraction used throughout the activity
- In feature-oriented SPLE, a feature is used as the key design abstraction to differentiate between the products of the same family.
- The motivation behind feature-orientation is that a feature is recognized by different stakeholders.
- For example, customers recognized a feature as a service or capability of a product that satisfies their needs.
- Requirement engineers can use a feature to describe functions that need to be developed.
- Developers can also use a feature to describe a unit of functionality that needs to be designed, developed, tested and maintained.
- Marketers can use a feature to promote a product to potential customers.

Activities in Feature-oriented Software Product Line (SPLE)



Feature Model

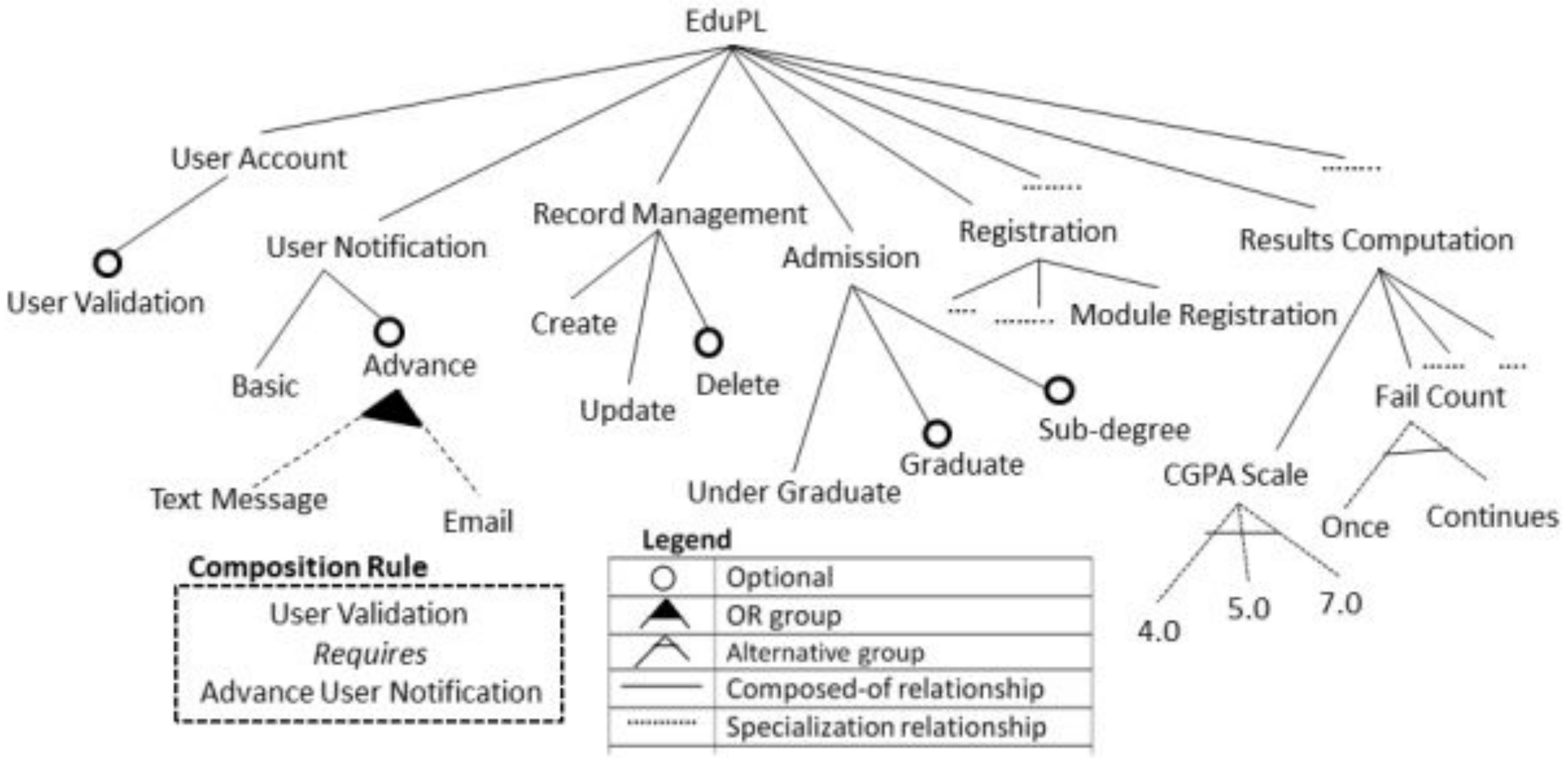
- Partial feature model of an automotive domain



Use case in feature-oriented SPL

- To validate the identified features using the use case model, the domain engineers should do the following checks:
  - Check if each use case can be mapped to a feature. If not, there may be a feature that is missing from the feature model.
  - Check if all the features are addressed by the use cases. If not, there may be use cases that are missing.
  - Check if variations of use cases and variations of features are consistent.
  - Check if the use cases are consistent with the semantics of the features defined in the dictionary.

EduPL Product line



Composition Rule

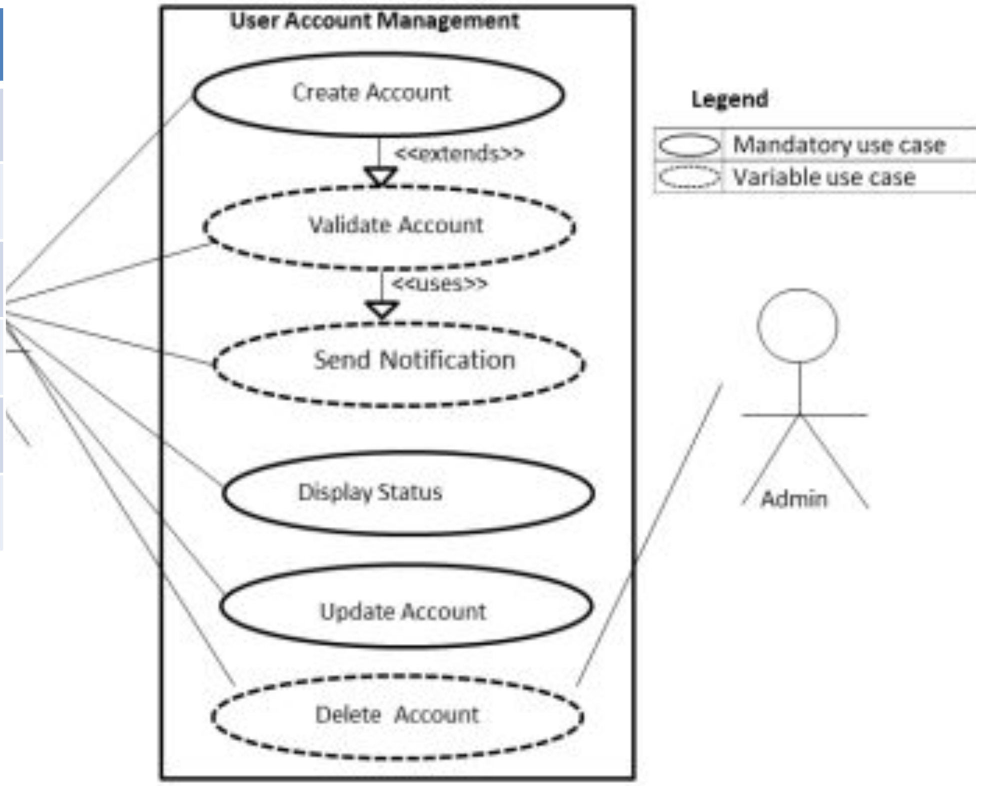
User Validation
Requires
Advance User Notification

Legend

○	Optional
▲	OR group
△	Alternative group
─	Composed-of relationship
┄	Specialization relationship

Use case of a subsystem in EduPL

Feature	Use case
Create	Create Account
Validate user	Validate Account
Advance Not.	Send Notification
Update	Update Account
Basic Not.	Display Status
Delete	Delete Account



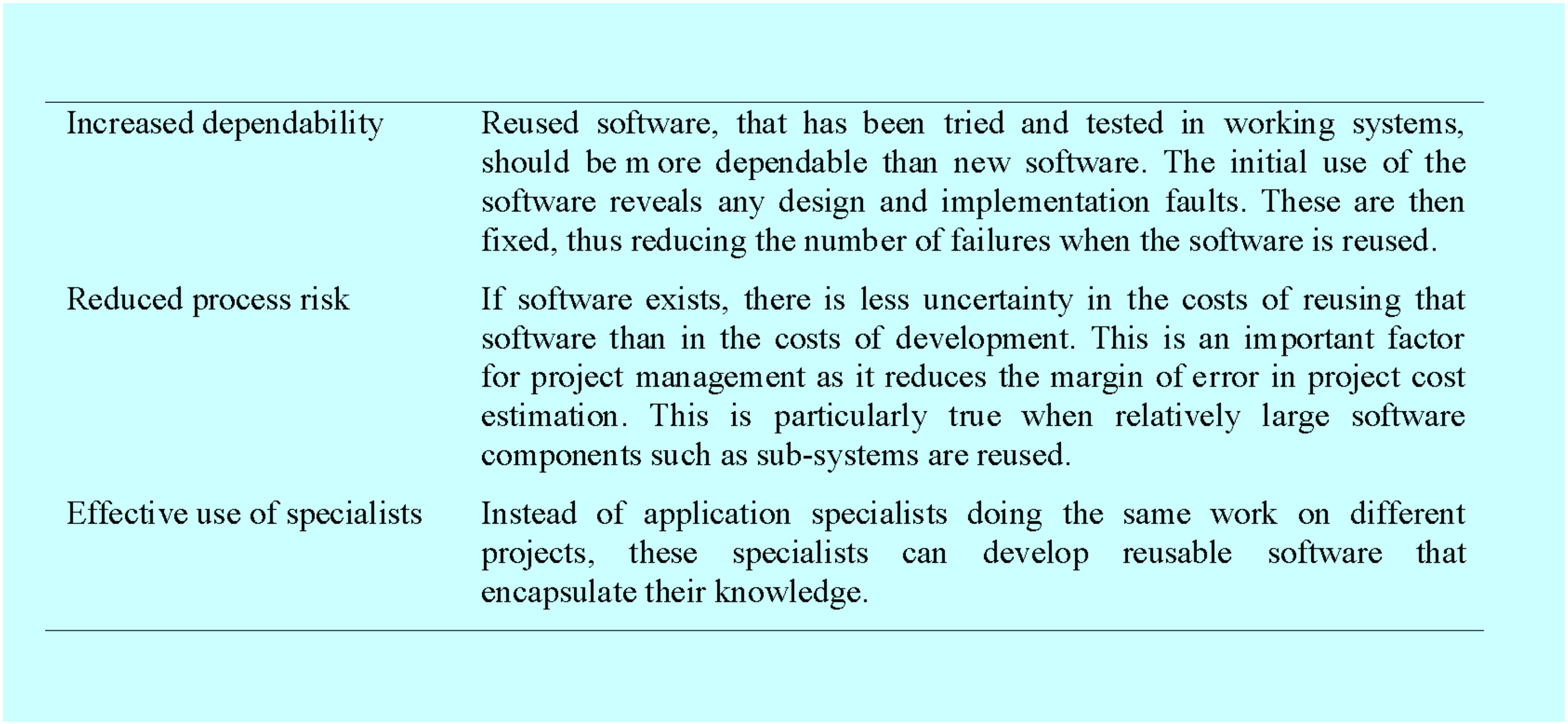
Design for reuse

- This approach involves designing software components or systems with the explicit intention of making them reusable in future projects or within the same project.
- Developers anticipate potential reuse scenarios and design components with flexibility, generality, and modularity in mind.
- Emphasis is placed on creating software artifacts that are easily adaptable and can be seamlessly integrated into different contexts without extensive modification.
- Design for reuse typically involves upfront investment in designing flexible and modular architectures, which may require additional time and effort during the initial development phase.

Design with reuse

- This approach focuses on leveraging existing reusable components or resources during the design and development process of a new software system.
- Developers identify and utilize reusable components, libraries, frameworks, or patterns that already exist within the organization or in the broader software ecosystem.
- Emphasis is placed on selecting appropriate reusable assets and integrating them effectively into the current project to expedite development and improve overall quality.
- Design with reuse allows developers to take advantage of existing solutions and reduce development time and costs by leveraging pre-existing components rather than reinventing the wheel.

Reuse benefits 1



Increased dependability	Reused software, that has been tried and tested in working systems, should be m ore dependable than new software. The initial use of the software reveals any design and implementation faults. These are then fixed, thus reducing the number of failures when the software is reused.
Reduced process risk	If software exists, there is less uncertainty in the costs of reusing that software than in the costs of development. This is an im portant factor for project management as it reduces the margin of error in project cost estimation. This is particularly true when relatively large software components such as sub-systems are reused.
Effective use of specialists	Instead of application specialists doing the same work on different projects, these specialists can develop reusable software that encapsulate their knowledge.

Reuse benefits 2

Standards compliance	Some standards, such as user interface standards, can be implemented as a set of standard reusable components. For example, if menus in a user interfaces are implemented using reusable components, all applications present the same menu formats to users. The use of standard user interfaces improves dependability as users are less likely to make mistakes when presented with a familiar interface.
Accelerated development	Bringing a system to market as early as possible is o ften more important than overall development costs. Reusing software can speed up system production because both development and validation time should be reduced.

Reuse problems 1

Increased maintenance costs	If the source code of a reused software system or component is not available then maintenance costs may be increased as the reused elements of the system may become increasingly incompatible with system changes.
Lack of tool support	CASE toolsets may not support development with reuse. It may be difficult or impossible to integrate these tools with a component library system. The software process assumed by these tools may not take reuse into account.
Not-invented-here syndrome	Some software engineers sometimes prefer to re-write components as they believe that they can improve on the reusable component. This is partly to do with trust and partly to do with the fact that writing original software is s een as more challenging than reusing other peopleصs software.

Reuse problems 2

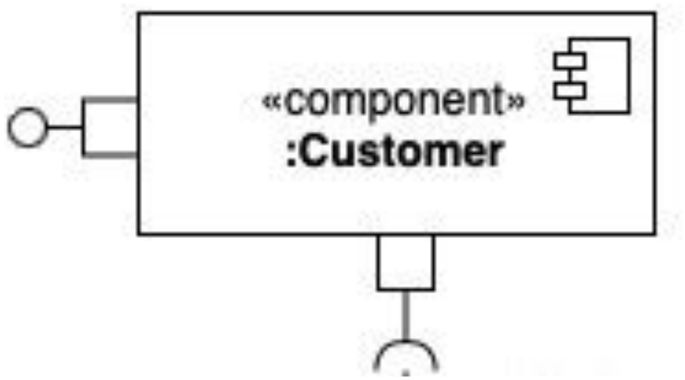
Creating and maintaining a component library	Populating a reusable component library and ensuring the software developers can use this library can be expensive. Our current techniques for classifying, cataloguing and retrieving software components are immature.
Finding, understanding and adapting reusable components	Software components have to be discovered in a library, understood and, sometimes, adapted to work in a n ew environment. Engineers must be reasonably confident of finding a component in the library before they will make routinely include a component search as part of their normal development process.

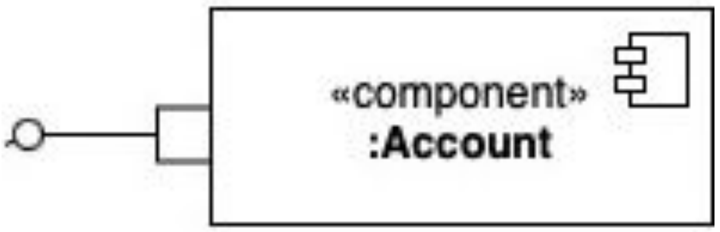
Component-based software engineering (CBSE)

- Component-based software engineering (CBSE) is an approach to software development that relies on 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 standalone service providers

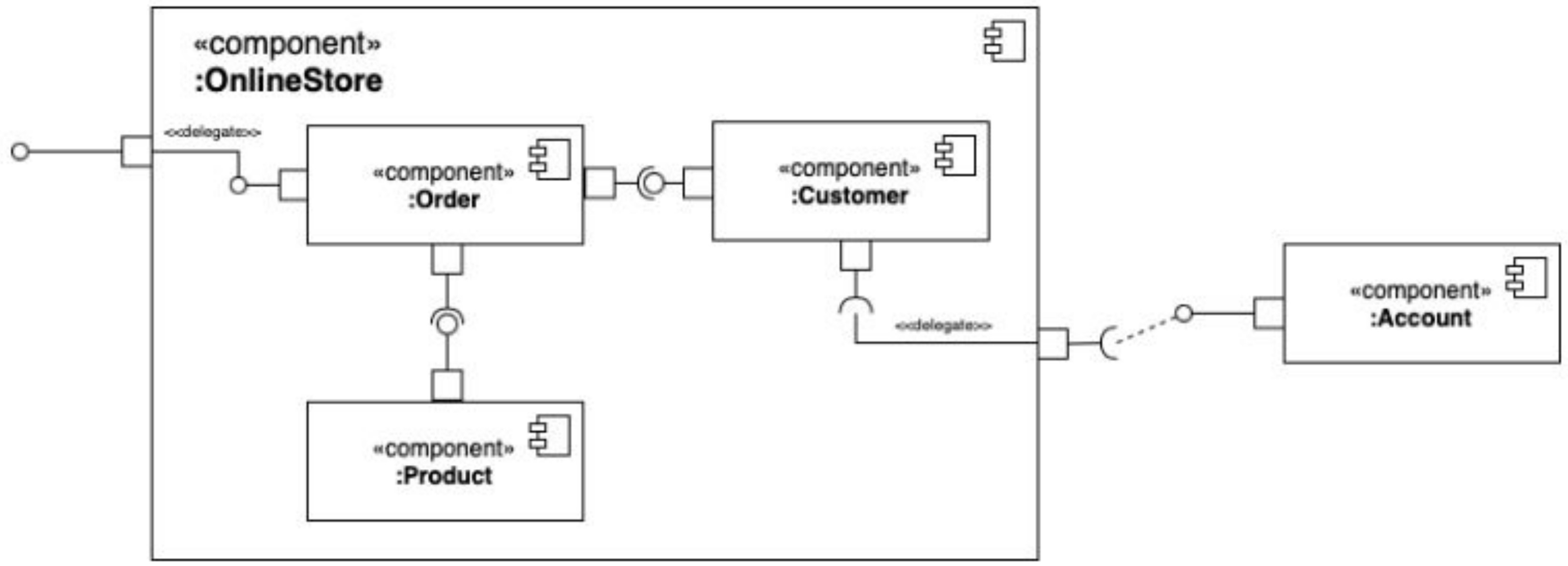
Component-based software engineering (CBSE)

- 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
- Components can range in size from simple functions to entire application systems





Component-based software engineering (CBSE)



Component-based software engineering (CBSE)

- Provides interface

– Defines the services that are provided by the component to other components

- Requires interface

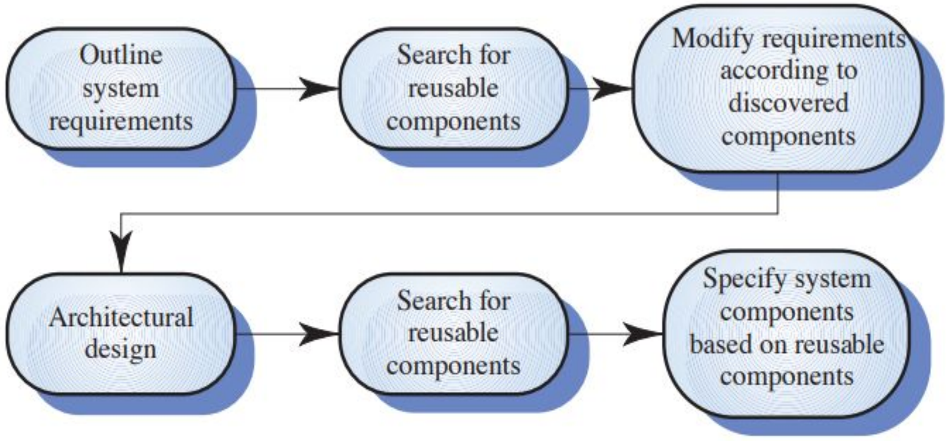
– Specifies what services must be made available for the component to execute

CBSE Process

- Component-based development can be integrated into a standard software process by incorporating a reuse activity in the process
- However, in reuse-driven development, the system requirements are modified to reflect the components that are available
- CBSE usually involves a prototyping or an incremental development process with components being ‘glued together’ using a scripting language

CBSE Process

- Component-based development can be integrated into a standard software process by incorporating a reuse activity in the process
- However, in reuse-driven development, the system requirements are modified to reflect the components that are available
- CBSE usually involves a prototyping or an incremental development process with components being ‘glued together’ using a scripting language


Outline system requirements
Search for reusable components
Modify requirements according to discovered components
Architectural design
Search for reusable components
Specify system components based on reusable components

CBSE Process

CBSE problems

- Component incompatibilities may mean that cost and schedule savings are less than expected
- Finding and understanding components
- Managing evolution as requirements change in situations where it may be impossible to change the system components

Component Model

- A component model is a specification or framework that defines the rules, standards, and guidelines for creating, assembling, deploying, and managing software components within a particular environment or platform.
- Examples of component models include:
  - COM/DCOM (Component Object Model/Distributed Component Object Model) in the Microsoft Windows environment;
  - JavaBeans in the Java platform;
  - CORBA (Common Object Request Broker Architecture) in distributed systems.