

# DOMAIN MODEL VISUALIZING CONCEPTS

Biju R Mohan

Lecture 12

# Agenda

- Identify conceptual classes related to the current iteration requirements.
- Create an initial domain model.
- Distinguish between correct and incorrect attributes.
- Add *specification conceptual classes, when appropriate.*
- Compare and contrast conceptual and implementation views.

# What is domain model ?

- A domain model illustrates meaningful conceptual classes in a problem domain; it is the most important artifact to create during object-oriented analysis.
- Identifying a rich set of objects or conceptual classes is at the heart of object-oriented analysis, and well worth the effort in terms of payoff during the design and implementation work.
- The identification of conceptual classes is part of an investigation of the problem domain. The UML contains notation in the form of class diagrams to illustrate domain models.

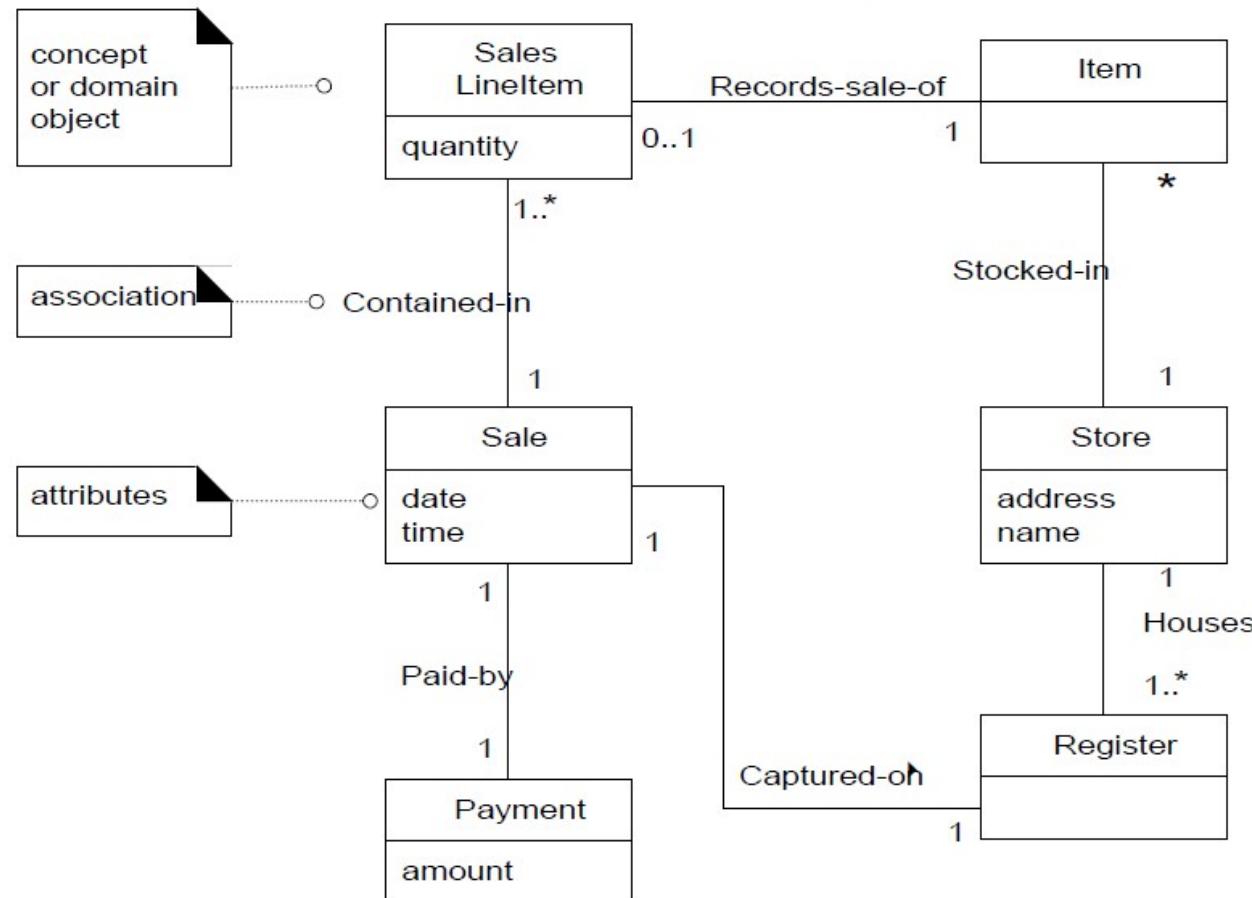
# Key Point

- A domain model is a representation of real-world conceptual classes, not of software components. It is *not a set of diagrams describing software classes, or software objects with responsibilities.*

# Domain Model

- The UP defines a Domain Model as one of the artifacts that may be created in the Business Modeling discipline.
- Using UML notation, a domain model is illustrated with a set of **class diagrams**
  - domain objects or conceptual classes
  - associations between conceptual classes
  - attributes of conceptual classes

# Partial Domain Model of POS problem



# *Domain Models Are not Models of Software Components*

- A domain model is a visualization of things in the realworld domain of interest, *not of software components such as a Java or C++*
- Therefore, the following elements are not suitable in a domain model:
  - Software artifacts, such as a window or a database, unless the domain being modeled is of software concepts, such as a model of graphical user interfaces.
  - Responsibilities or methods.

# Example

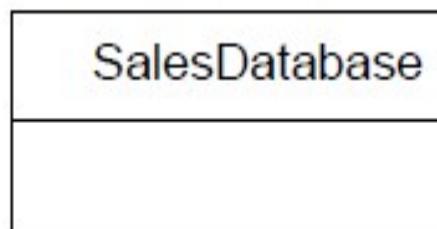


visualization of a real world concept in the domain of interest

it is *an* a picture of a software class

# Avoid

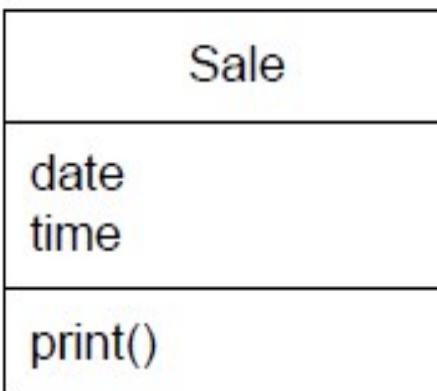
avoid



0.....

software artifact; not part  
of domain model

avoid



0.....

software class; not part  
of domain model

# *Conceptual Classes*

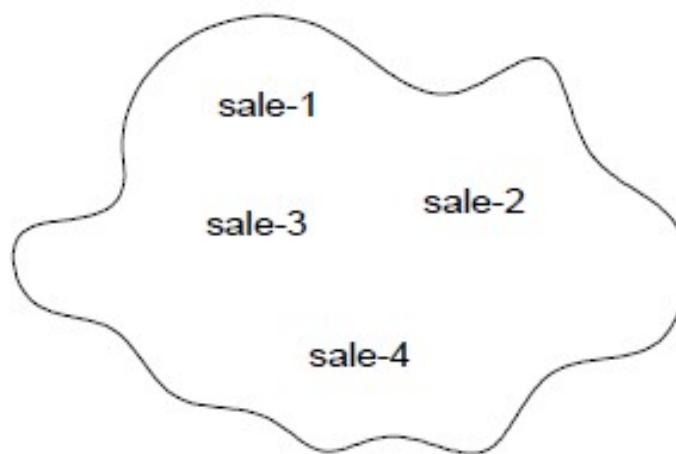
- Informally, a conceptual class is an idea, thing, or object. More formally, a conceptual class may be considered in terms of its symbol, intension, and extension
  - **Symbol**—words or images representing a conceptual class.
  - **Intension**—the definition of a conceptual class.
  - **Extension**—the set of examples to which the conceptual class applies



concept's symbol

"A sale represents the event of a purchase transaction. It has a date and time."

concept's intension



concept's extension

# OOAD Vs Structured Analysis

- A central distinction between object-oriented and structured analysis is: division by conceptual classes (objects) rather than division by functions

# Conceptual Class Identification

- Two techniques are presented in the following sections:
  - 1. Use a conceptual class category list.
  - 2. Identify noun phrases.

Conceptual Class Category	Examples
physical or tangible objects	<i>Register</i> <i>Airplane</i>
specifications, designs, or descriptions of things	<i>ProductSpecification</i> <i>FlightDescription</i>
places	<i>Store</i> <i>Airport</i>
transactions	<i>Sale</i> , <i>Payment</i> <i>Reservation</i>
transaction line items	<i>SalesLineItem</i>
roles of people	<i>Cashier</i> <i>Pilot</i>
containers of other things	<i>Store</i> , <i>Bin</i> <i>Airplane</i>
things in a container	<i>Item</i> <i>Passenger</i>

# *Finding Conceptual Classes with Noun Phrase Identification*

## **Main Success Scenario (or Basic Flow):**

1. Customer arrives at a **POS** checkout with **goods** and/or **services** to purchase.
2. Cashier starts a new **sale**.
3. Cashier enters **item identifier**.
4. System records **sale** line **item** and presents **item description**, **price**, and running **total**. Price calculated from a set of price rules.  
Cashier repeats steps 2-3 until indicates done.
5. System presents total with **taxes** calculated.
6. Cashier tells Customer the total, and asks for **payment**.
7. Customer pays and System handles payment.
8. System logs the completed **sale** and sends sale and payment information to the external **Accounting** (for accounting and commissions) and **Inventory** systems (to update inventory).
9. System presents **receipt**.
10. Customer leaves with receipt and goods (if any).

# Candidate Conceptual Classes for the Sales Domain

*Register*

*ProductSpecification*

*Item*

*SalesLineItem*

*Store*

*Cashier*

*Sale*

*Customer*

*Payment*

*Manager*

*ProductCatalog*

# *How to Make a Domain Model*

1. List the candidate conceptual classes using the Conceptual Class Category List and noun phrase identification techniques related to the current requirements under consideration.
2. Draw them in a domain model.
3. Add the associations necessary to record relationships for which there is a need to preserve some memory (discussed in a subsequent chapter).
4. Add the attributes necessary to fulfill the information requirements (discussed in a subsequent chapter).

# *On Naming and Modeling Things*

Make a domain model in the spirit of how a cartographer or mapmaker works:

- Use the existing names in the territory.
- Exclude irrelevant features.
- Do not add things that are not there.

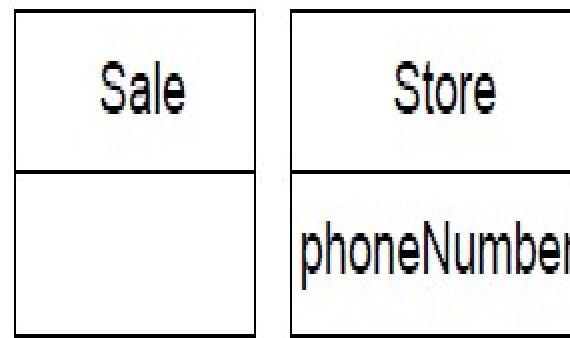
# *A Common Mistake in Identifying Conceptual Classes*

- If we do not think of some conceptual class X as a number or text in the real world, X is probably a conceptual class, not an attribute.

# Example



or... ?



# Modeling the *Unreal World*

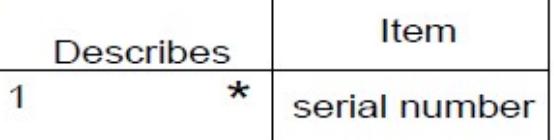
- Some software systems are for domains that find very little analogy in natural or business domains; software for telecommunications is an example.
- For example, here are some candidate conceptual classes related to a telecommunication
- switch: *Message, Connection, Port, Dialog, Route, Protocol.*

# *The Need for Specification or Description Conceptual Classes*

Item
description
price
serial number
itemID

**Worse**

ProductSpecification
description
price
itemID



**Better**

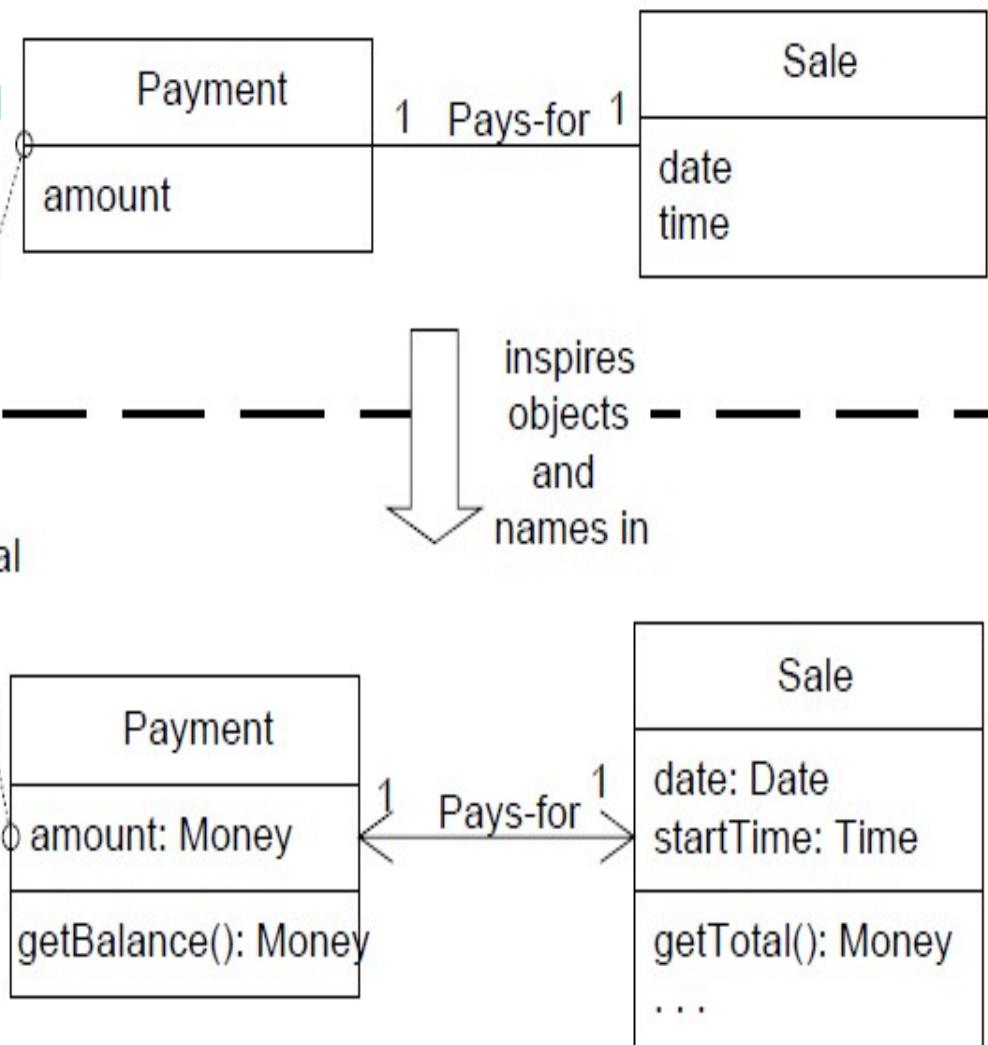
## UP Domain Model

Stakeholder's view of the noteworthy concepts in the domain.

A Payment in the Domain Model is a concept, but a Payment in the Design Model is a software class. They are not the same thing, but the former inspired the naming and definition of the latter.

This reduces the representational gap.

This is one of the big ideas in object technology.



## UP Design Model

The object-oriented developer has taken inspiration from the real world in creating software classes.

# References

- Chapter 10 Applying UML Patterns (**Applying UML Patterns: An Introduction To Object-Oriented Analysis And Design**) Craig L

# DOMAIN MODEL: ADDING ASSOCIATIONS

Biju R Mohan

Lecture 13

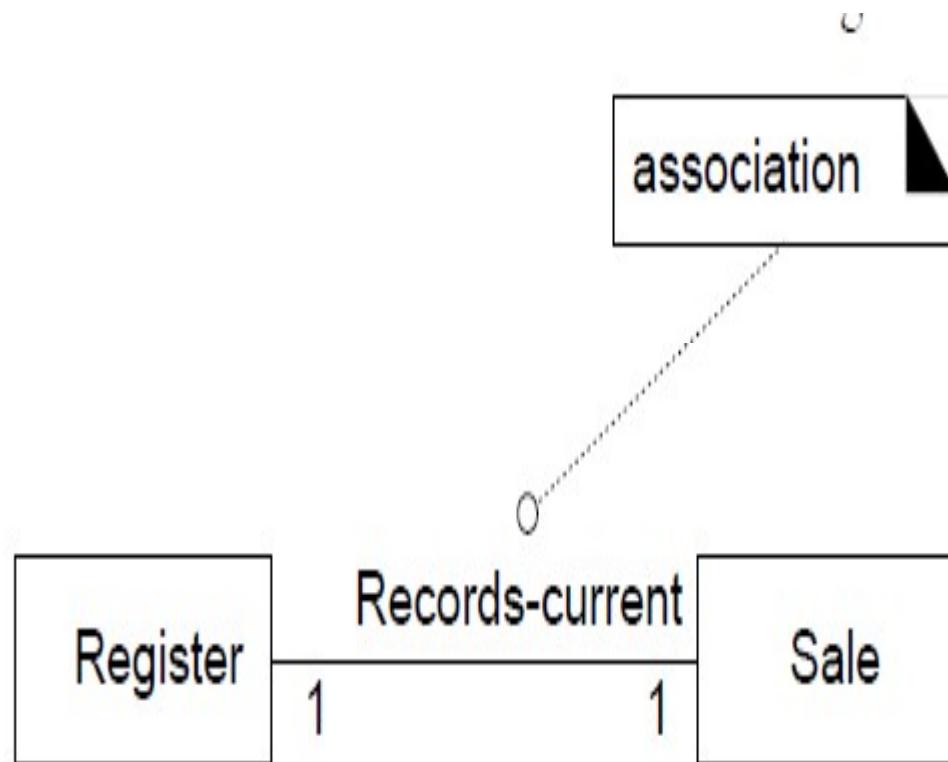
# Agenda

- Identify associations for a domain model.
- Distinguish between need-to-know and comprehension-only associations.

# Associations

- An **association** is a relationship between types (or more specifically, instances of those types) that indicates some meaningful and interesting connection

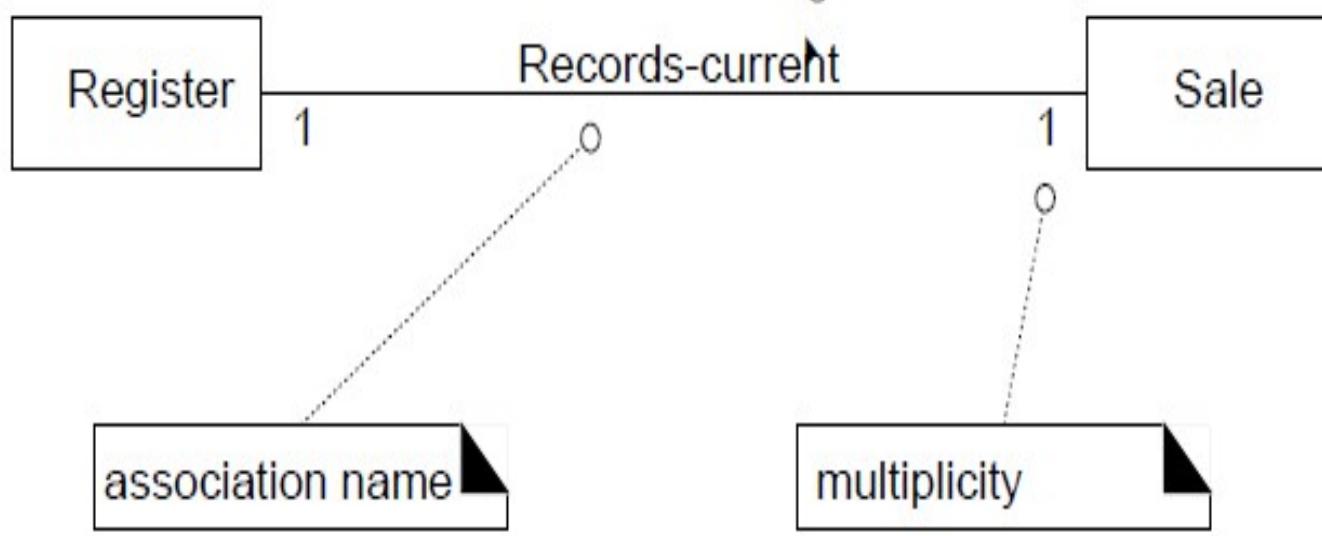
# UML Notation of Association



# Types

- Consider including the following associations in a domain model:
  - Associations for which knowledge of the relationship needs to be preserved for some duration ("need-to-know" associations).
  - Associations derived from the Common Associations List.

- "reading direction arrow"  
- it has **no** meaning except to indicate direction  
reading the association label  
- often excluded



Category	Examples
A is a physical part of B	<i>Drawer</i> — <i>Register</i> (or more specifically, a <i>POST</i> ) <i>Wing</i> — <i>Airplane</i>
A is a logical part of B	<i>SalesLineItem</i> — <i>Sale</i> <i>FlightLeg</i> — <i>FlightRoute</i>
A is physically contained in/on B	<i>Register</i> — <i>Store</i> , <i>Item</i> — <i>Shelf</i> <i>Passenger</i> — <i>Airplane</i>
A is logically contained in B	<i>ItemDescription</i> — <i>Catalog</i> <i>Flight</i> — <i>FlightSchedule</i>
A is a description for B	<i>ItemDescription</i> — <i>Item</i> <i>FlightDescription</i> — <i>Flight</i>
A is a line item of a transaction or report B	<i>SalesLineItem</i> — <i>Sale</i> <i>Maintenance Job</i> — <i>Maintenance-Log</i>
A is known/logged/recorded/reported/captured in B	<i>Sale</i> — <i>Register</i> <i>Reservation</i> — <i>FlightManifest</i>
A is a member of B	<i>Cashier</i> — <i>Store</i> <i>Pilot</i> — <i>Airline</i>

A is a member of B	<i>Cashier</i> — <i>Store</i> <i>Pilot</i> — <i>Airline</i>
A is an organizational subunit of B	<i>Department</i> — <i>Store</i> <i>Maintenance</i> — <i>Airline</i>
A uses or manages B	<i>Cashier</i> — <i>Register</i> <i>Pilot</i> — <i>Airplane</i>
A communicates with B	<i>Customer</i> — <i>Cashier</i> <i>Reservation Agent</i> — <i>Passenger</i>
A is related to a transaction B	<i>Customer</i> — <i>Payment</i> <i>Passenger</i> — <i>Ticket</i>
A is a transaction related to another transaction B	<i>Payment</i> — <i>Sale</i> <i>Reservation</i> — <i>Cancellation</i>
A is next to B	<i>SalesLineItem</i> — <i>SalesLineItem</i> <i>City</i> — <i>City</i>

Category	Examples
A is owned by B	<i>Register</i> — <i>Store</i> <i>Plane</i> — <i>Airline</i>
A is an event related to B	<i>Sale</i> — <i>Customer</i> , <i>Sale</i> — <i>Store</i> <i>Departure</i> — <i>Flight</i>

## *High-Priority Associations*

Here are some high-priority association categories that are invariably useful to include in a domain model:

- A is a *physical or logical part* of B.
- A is *physically or logically contained* in/on B.
- A is *recorded in* B.

# Association Guidelines

- Focus on those associations for which knowledge of the relationship needs to be preserved for some duration ("need-to-know" associations).
- It is more important to identify *conceptual classes* than to identify associations.
- Too many associations tend to confuse a domain model rather than illuminate it. Their discovery can be time-consuming, with marginal benefit.
- Avoid showing redundant or derivable associations.

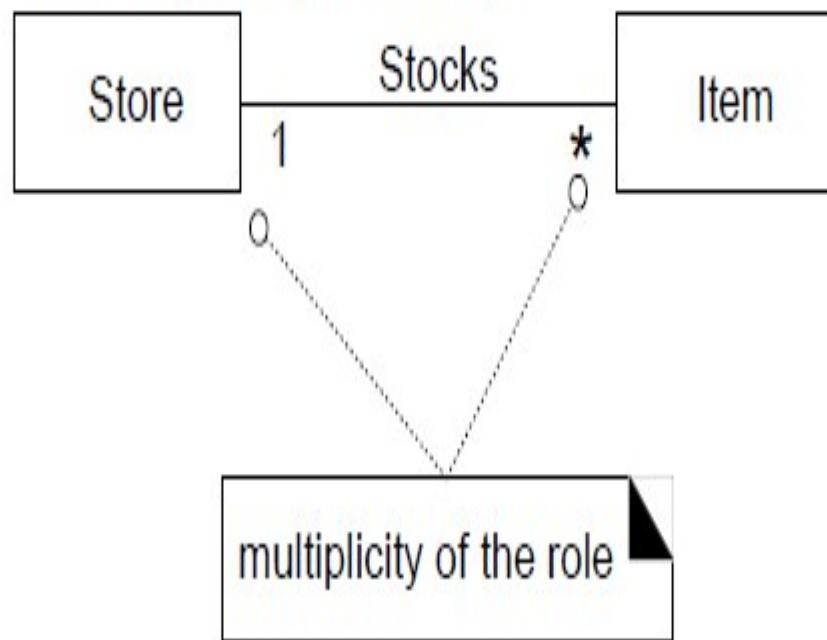
# Roles

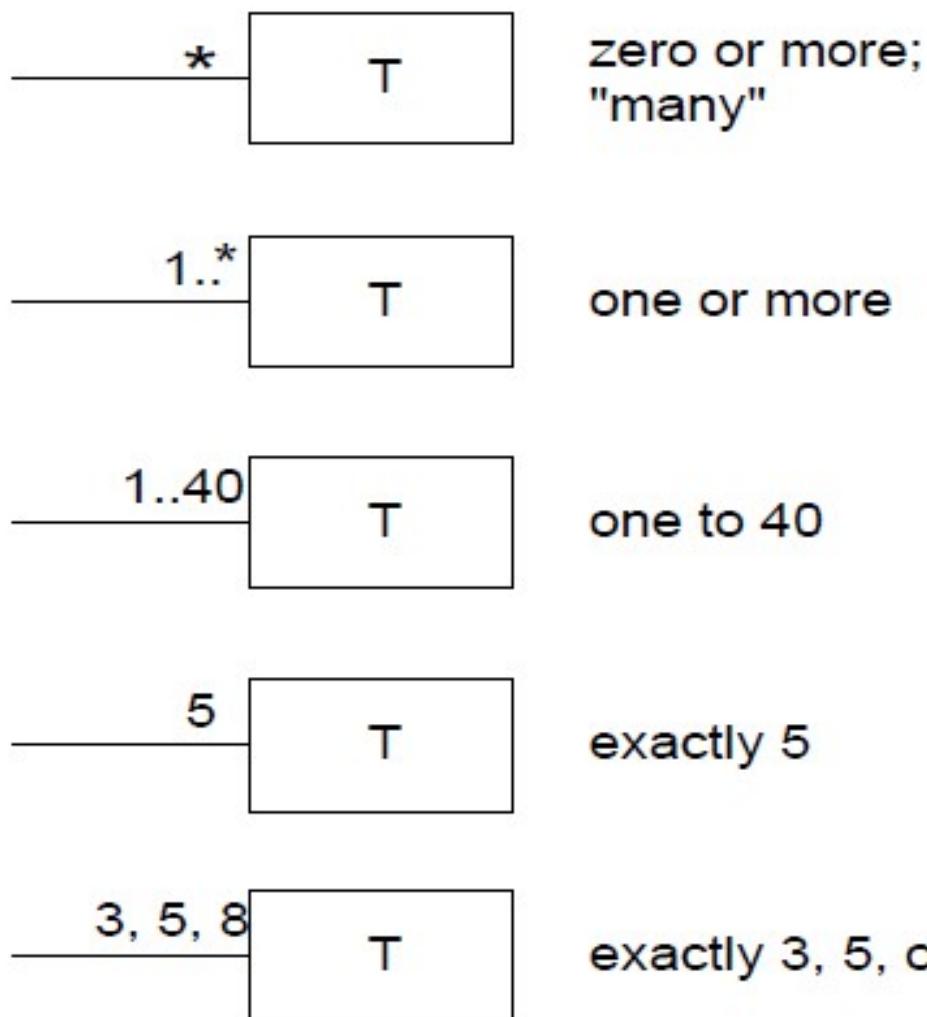
Each end of an association is called a **role**. Roles may optionally have:

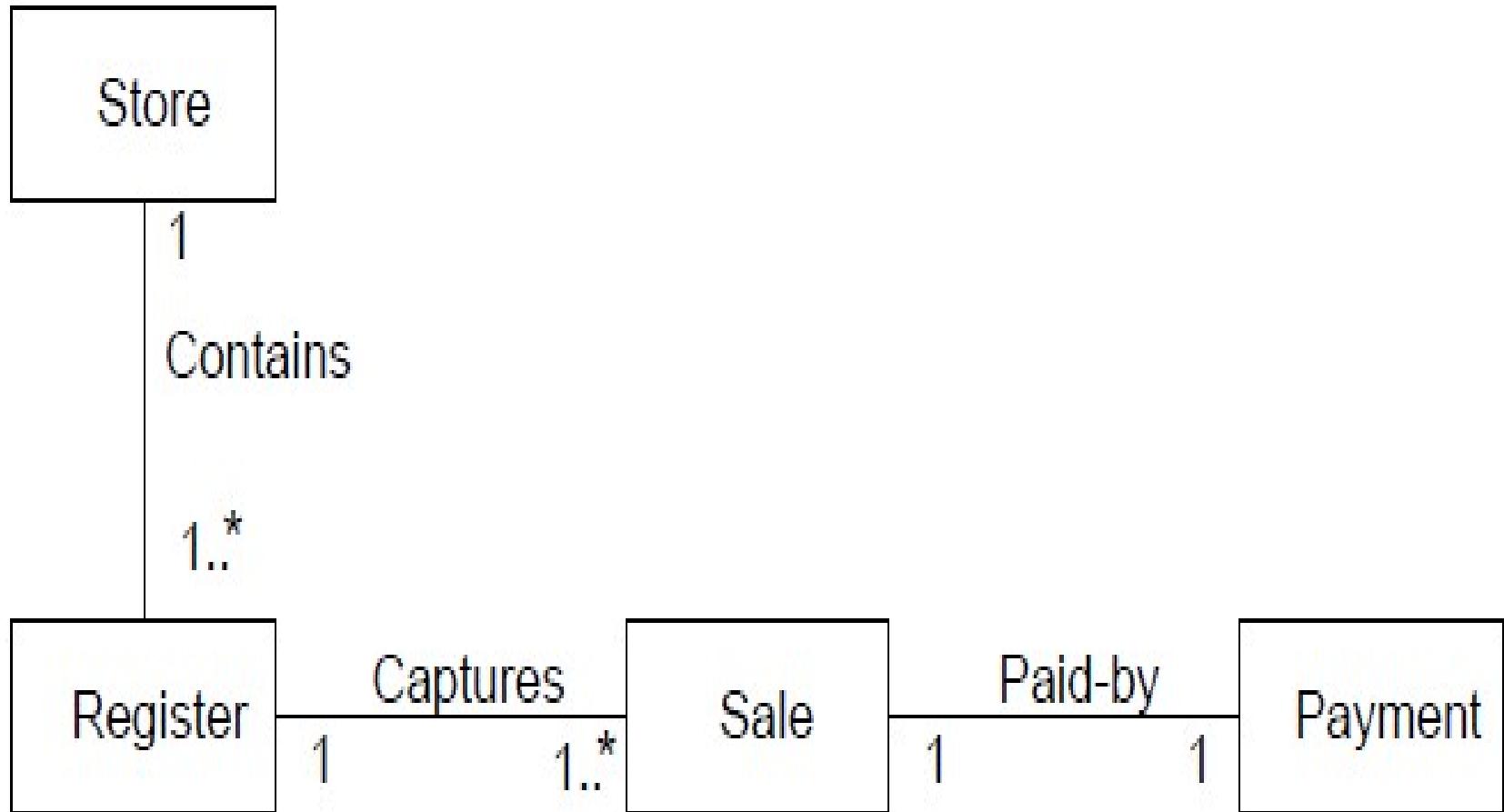
- name
- multiplicity expression
- navigability

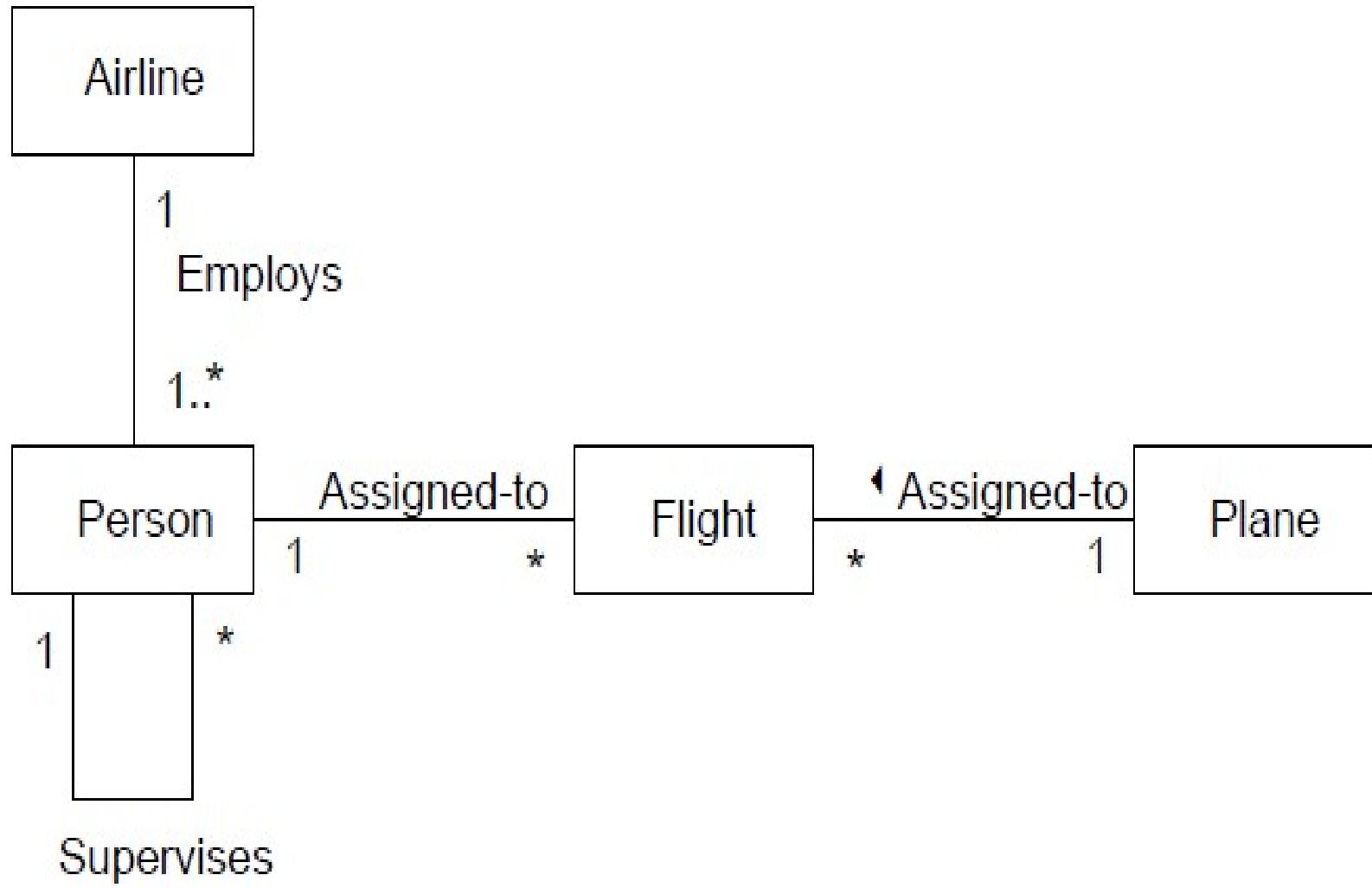
# *Multiplicity*

**Multiplicity** defines how many instances of a class *A* can be associated with one instance of a class *B* (see Figure 11.3).

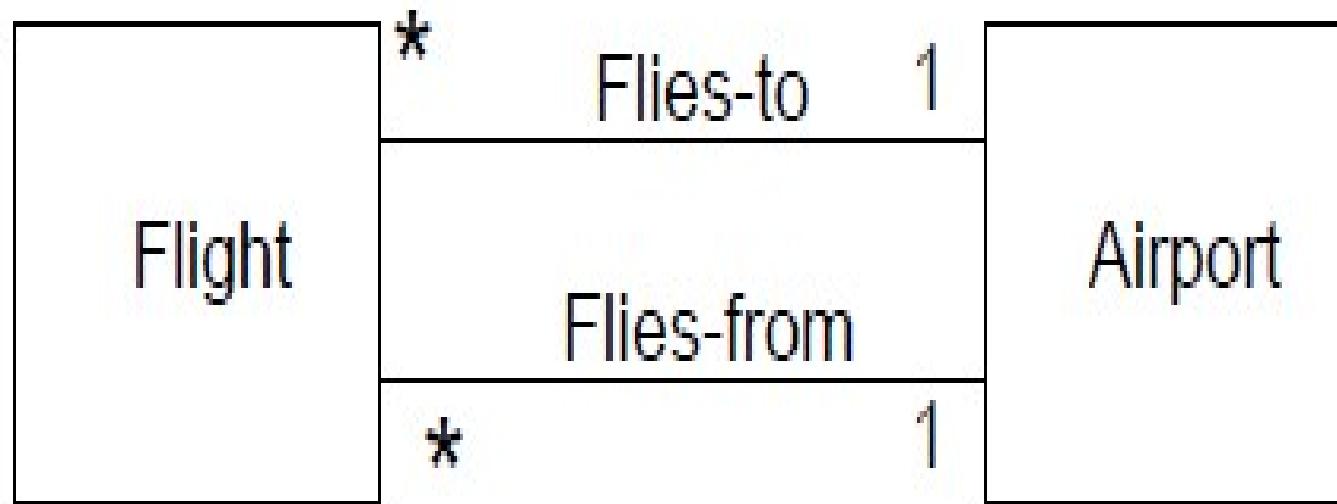


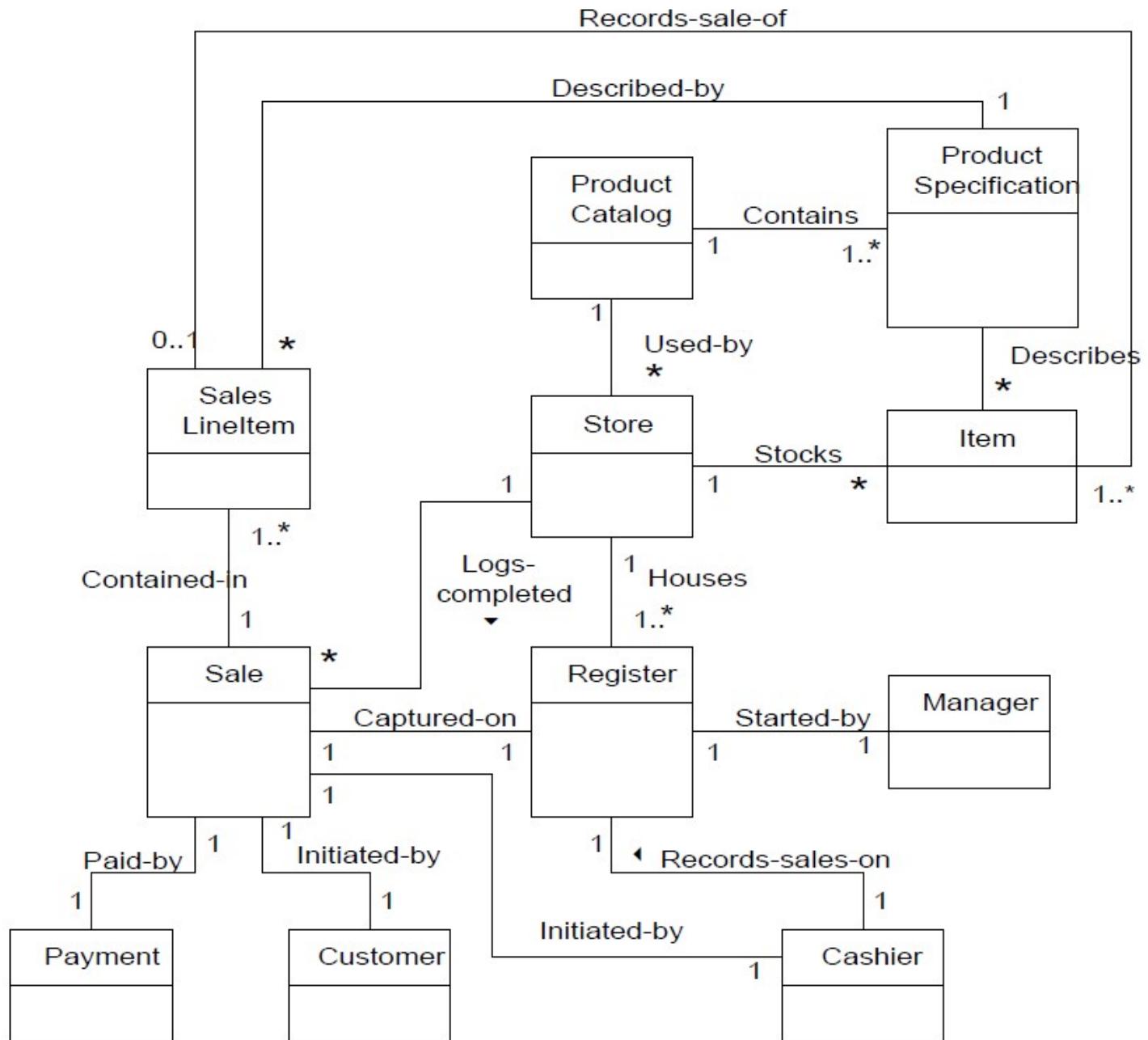






# Multiple Associations Between Two Types





# Reference

- Chapter 11 Applying UML Patterns (**Applying UML Patterns: An Introduction To Object-Oriented Analysis And Design**) Craig L

# Lecture 14

Design Patterns

# What is a Design Pattern?

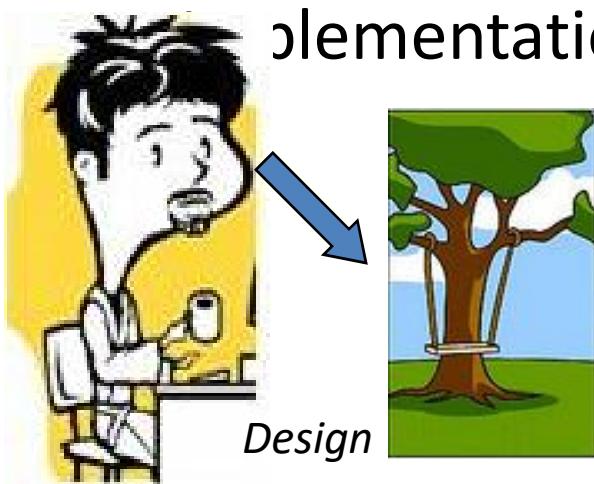
- A (Problem, Solution) pair.
- A technique to repeat designer success.
- Borrowed from Civil and Electrical Engineering domains.

# How Patterns are used?

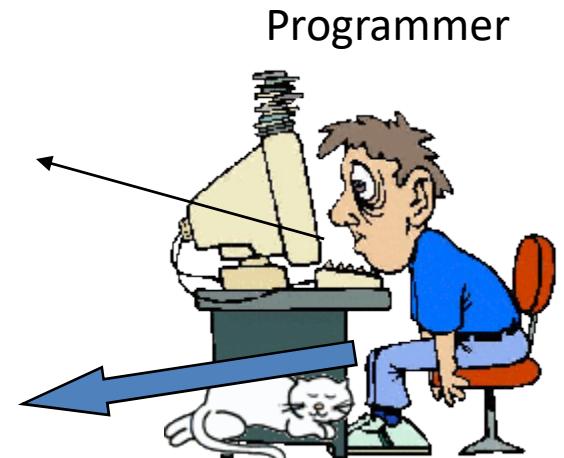
Designer

- Design Problem.
- Solution.

Implementation details.



Reduce gap



Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design patterns: elements of reusable object-oriented software*. 1995.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: *Pattern-oriented software architecture: a system of patterns*. 2002.

# Design patterns you have already seen

- Encapsulation (Data Hiding)
- Subclassing (Inheritance)
- Iteration
- Exceptions

# Encapsulation pattern

- **Problem:** Exposed fields are directly manipulated from outside, leading to undesirable dependences that prevent changing the implementation.
- **Solution:** Hide some components, permitting only stylized access to the object.

# Subclassing pattern

- **Problem:** Similar abstractions have similar members (fields and methods). Repeating these is tedious, error-prone, and a maintenance headache.
- **Solution:** Inherit default members from a superclass; select the correct implementation via run-time dispatching.

# Iteration pattern

- **Problem:** Clients that wish to access all members of a collection must perform a specialized traversal for each data structure.
- **Solution:** Implementations perform traversals. The results are communicated to clients via a standard interface.

# Exception pattern

- **Problem:** Code is cluttered with error-handling code.
- **Solution:** Errors occurring in one part of the code should often be handled elsewhere. Use language structures for throwing and catching exceptions.

# Derived Conclusion

- Patterns are Programming language features.
- Programming languages are moving towards Design.
- Many patterns are being implemented in programming languages.

# Pattern Categories

- **Creational Patterns** concern the process of object creation.
- **Structural Patterns** concern with integration and composition of classes and objects.
- **Behavioral Patterns** concern with class or object communication.

# What is the addressing Quality Attribute?

- Modifiability, Exchangeability, Reusability, Extensibility, Maintainability.

What properties these patterns provide?

- More general code for better Reusability.
- Redundant code elimination for better Maintainability.

# What is the Singleton Pattern?

- The Singleton pattern ensures that a class is only instantiated once and provides a global access point for this instance

# Creational Pattern

- Abstracts the instantiation process
- Object Creational Pattern
  - Delegates the instantiation to another object

# Why use the Singleton Pattern?

- It is important for some classes to only have one instance
- It is also important that this single instance is easily accessible
- Why not use a global object?
  - Global variables make objects accessible, but they don't prevent the instantiation of multiple objects.

# Solution

- Make the class responsible for keeping track of its only instance
  - The class can ensure that no other instances are created
  - This provides a useful way to access the instance
  - This is the Singleton pattern

# Examples of Singleton Patterns

- Print Spooler
- File Systems
- Window Managers
- Digital Filters
- Accounting Systems

# Sample Class

```
class Singleton
{
    // static variable single_instance of type Singleton
    private static Singleton single_instance = null;

    // variable of type String
    public String s;
```

# Sample Implementation

```
// private constructor restricted to this class itself
private Singleton()
{
    s = "Hello I am a string part of Singleton class";
}
// static method to create instance of Singleton class

} e;
};
```

```
public static Singleton getInstance()
{
    if (single_instance == null)
        single_instance = new Singleton();

    return single_instance;
}
```

# Things to Notice

- `getInstance()` function ensures that only one instance is created and that it is initialized before use
- Singleton constructor is declared as `private`
  - Direct instantiation causes errors at compile time

# Benefits of the Singleton Pattern

- Controlled access to sole instance
- Reduced name space
- Allows refinement of operations and representation
- Allows a variable number of instances
- More flexible than class operations

# Use the Singleton Pattern when...

- There must be exactly one instance of a class
  - This instance must be accessible from a well-known access point
- The instance should be extensible by subclassing
  - Clients should be able to use a derived class without modifying their code

# Facade Pattern

Lecture 15

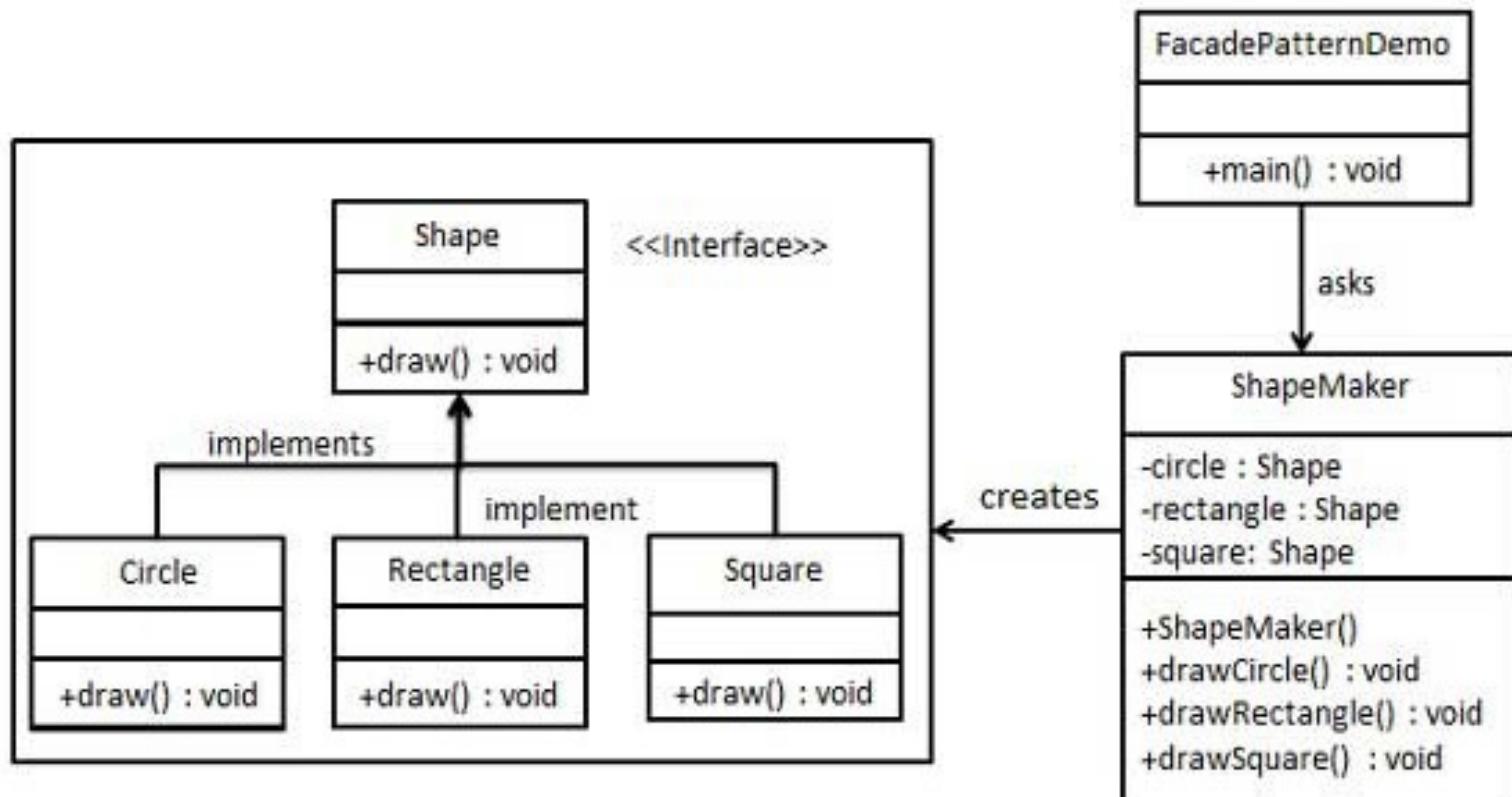
# Facade

- “Provide a unified interface to a set of interfaces in a collection of subsystems.”
- Facade defines a higher-level interface that makes the subsystems easier to use.”

# Motivation

- “Common design goal is to minimize the communication and dependencies between subsystems. One way to achieve this goal is to introduce a facade object that provides a single, simplified interface to the more general facilities of a subsystem.”
- With façade clients don't have to be concerned with exactly which object in a subsystem they're dealing with. They just call methods on the facade in blissful ignorance.

# Implementation



## Step 1

Create an interface.

*Shape.java*

```
public interface Shape {  
    void draw();  
}
```

## Step 2

Create concrete classes implementing the same interface.

*Rectangle.java*

```
public class Rectangle implements Shape {  
  
    @Override  
    public void draw() {  
        System.out.println("Rectangle::draw()");  
    }  
}
```

### *Square.java*

```
public class Square implements Shape {  
  
    @Override  
    public void draw() {  
        System.out.println("Square::draw()");  
    }  
}
```

### *Circle.java*

```
public class Circle implements Shape {  
  
    @Override  
    public void draw() {  
        System.out.println("Circle::draw()");  
    }  
}
```

```
public class ShapeMaker {  
    private Shape circle;  
    private Shape rectangle;  
    private Shape square;  
  
    public ShapeMaker() {  
        circle = new Circle();  
        rectangle = new Rectangle();  
        square = new Square();  
    }  
  
    public void drawCircle(){  
        circle.draw();  
    }  
    public void drawRectangle(){  
        rectangle.draw();  
    }  
    public void drawSquare(){  
        square.draw();  
    }  
}
```

Use the facade to draw various types of shapes.

### *FacadePatternDemo.java*

```
public class FacadePatternDemo {  
    public static void main(String[] args) {  
        ShapeMaker shapeMaker = new ShapeMaker();  
  
        shapeMaker.drawCircle();  
        shapeMaker.drawRectangle();  
        shapeMaker.drawSquare();  
    }  
}
```

## Step 5

Verify the output.

```
Circle::draw()  
Rectangle::draw()  
Square::draw()
```

# References

- [https://www.tutorialspoint.com/design\\_pattern/facade\\_pattern.htm](https://www.tutorialspoint.com/design_pattern/facade_pattern.htm)

# Builder Pattern

Lecture 16

# Builder Pattern

- The builder pattern is a design pattern designed to provide a flexible solution to various object creation problems in object-oriented programming.
- The intent of the Builder design pattern is to separate the construction of a complex object from its representation.

# Problem

- The Builder design pattern solves problems like:
  - How can a class (the same construction process) create different representations of a complex object?
  - How can a class that includes creating a complex object be simplified?

# Solution

- The Builder design pattern describes how to solve such problems:
- Encapsulate creating and assembling the parts of a complex object in a separate Builder object.
- A class delegates object creation to a Builder object instead of creating the objects directly.

# Intent

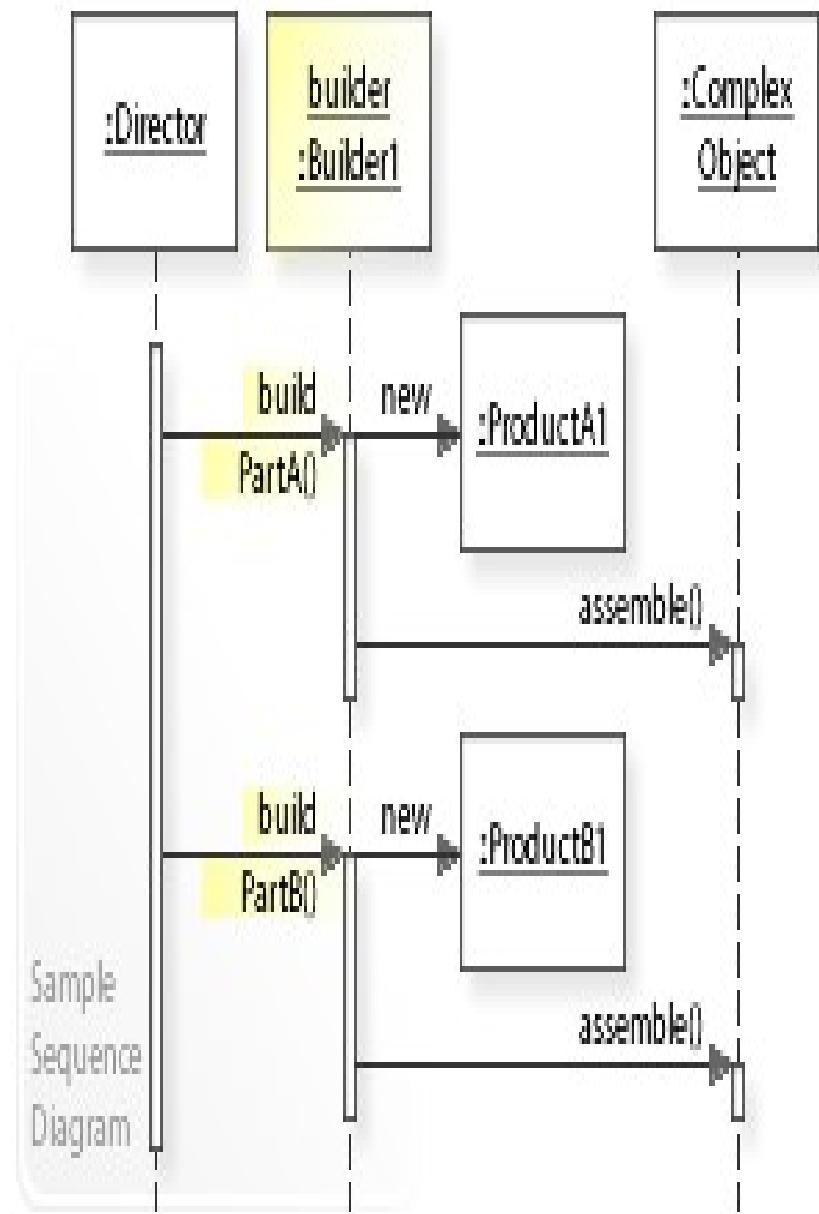
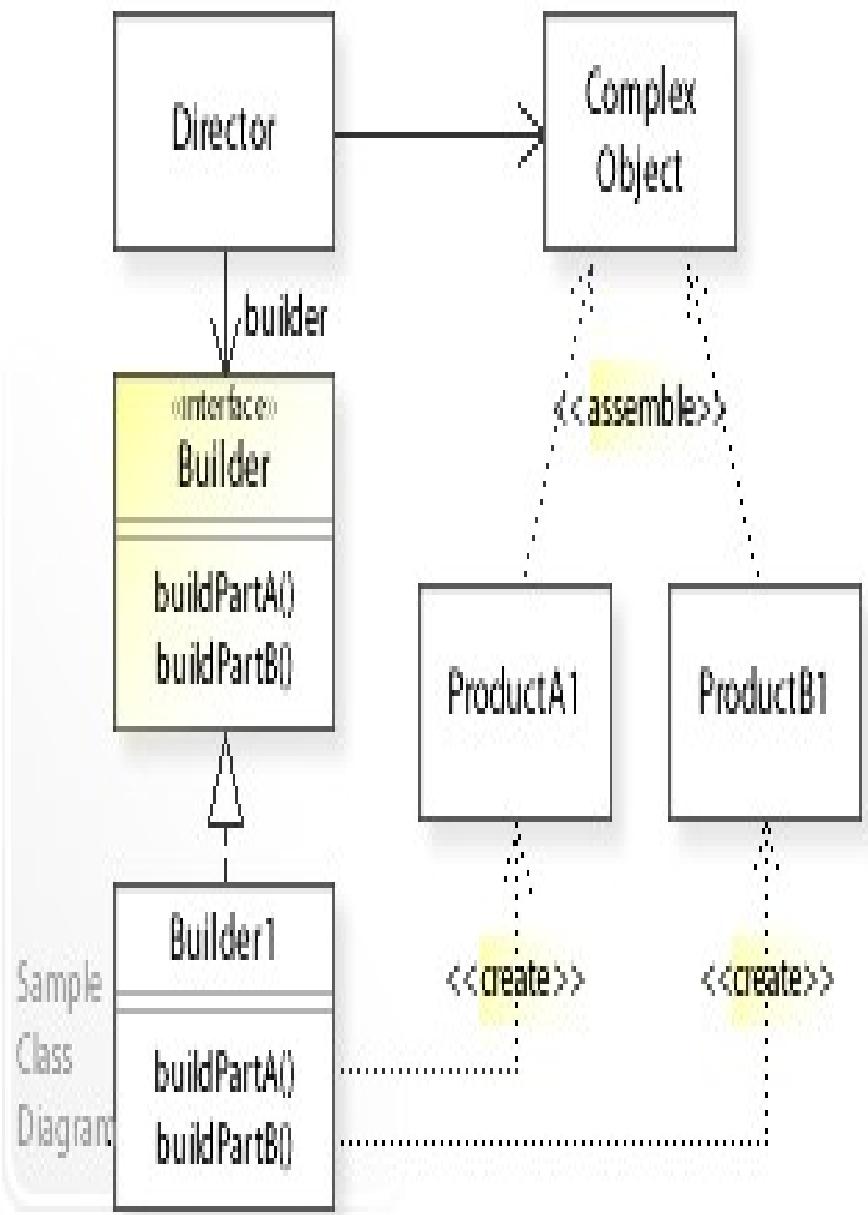
- The intent of the Builder design pattern is to separate the construction of a complex object from its representation. By doing so the same construction process can create different representations.

# Advantages

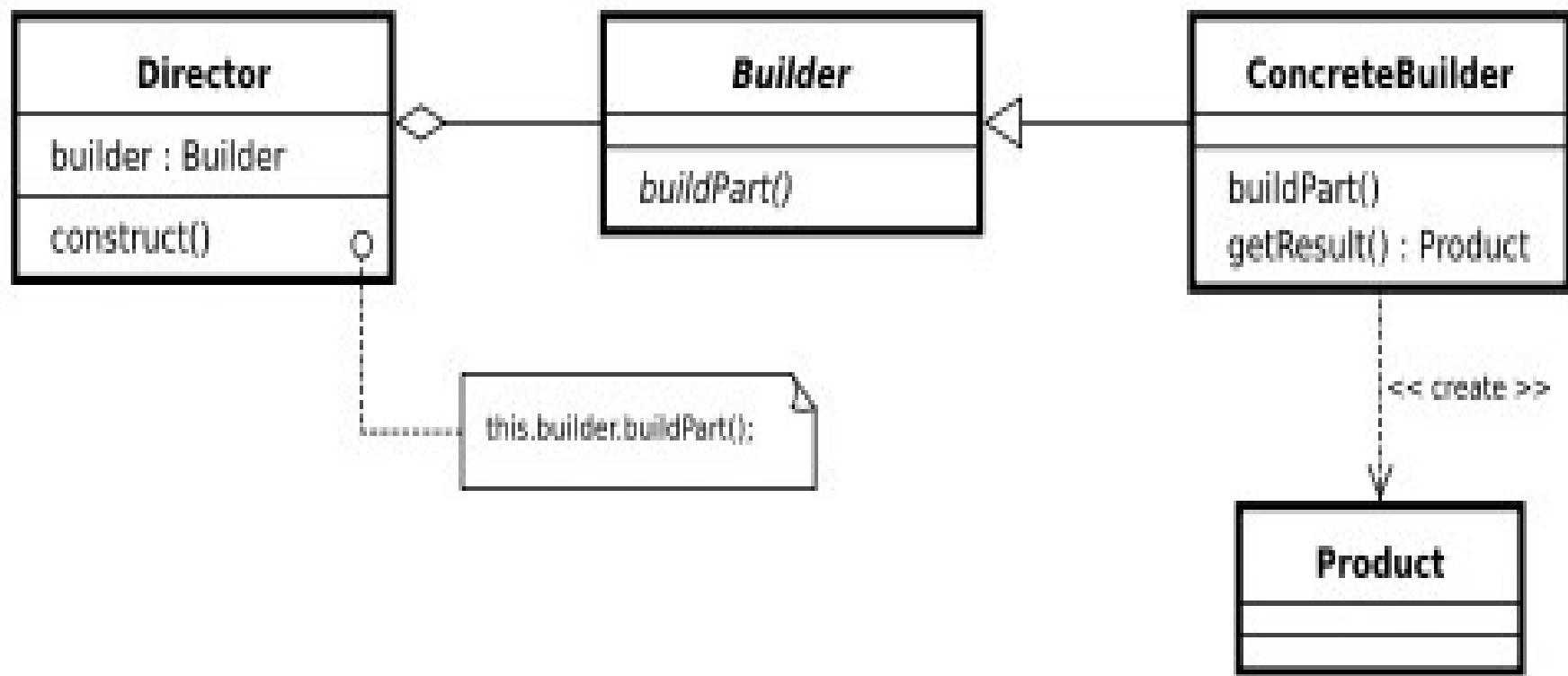
- Advantages of the Builder pattern include:
  - Allows you to vary a product's internal representation.
  - Encapsulates code for construction and representation.
  - Provides control over steps of construction process.

# Disadvantages

- Disadvantages of the Builder pattern include:
  - Requires creating a separate `ConcreteBuilder` for each different type of product.
  - Requires the builder classes to be mutable.
  - Dependency injection may be less supported.



# Class diagram



# Represents a product created by the builder

```
public class Car
{
    public string Make { get; set; }
    public string Model { get; set; }
    public int NumDoors { get; set; }
    public string Colour { get; set; }

    public Car(string make, string model, string colour, int
numDoors)
    {
        Make = make;      Model = model;
        Colour = colour;  NumDoors = numDoors;
    }
}
```

```
/// The builder abstraction
public interface ICarBuilder
{
    string Colour { get; set; }
    int NumDoors { get; set; }

    Car GetResult();
}
```

```
/// Concrete builder implementation
public class FerrariBuilder : ICarBuilder
{
    public string Colour { get; set; }
    public int NumDoors { get; set; }

    public Car GetResult()
    {
        return NumDoors == 2 ? new Car("Ferrari",
            "488 Spider", Colour, NumDoors) : null;
    }
}
```

```
/// The director
public class SportsCarBuildDirector
{
    private ICarBuilder _builder;
    public SportsCarBuildDirector(ICarBuilder builder)
    {
        _builder = builder;
    }

    public void Construct()
    {
        _builder.Colour = "Red";
        _builder.NumDoors = 2;
    }
}
```

```
public class Client
{
    public void DoSomethingWithCars()
    {
        var builder = new FerrariBuilder();
        var director = new
SportsCarBuildDirector(builder);

        director.Construct();
        Car myRaceCar = builder.GetResult();
    }
}
```

# References

- [https://en.wikipedia.org/wiki/Builder\\_pattern](https://en.wikipedia.org/wiki/Builder_pattern)

# Factory Method

Lecture 17

# Factory Method

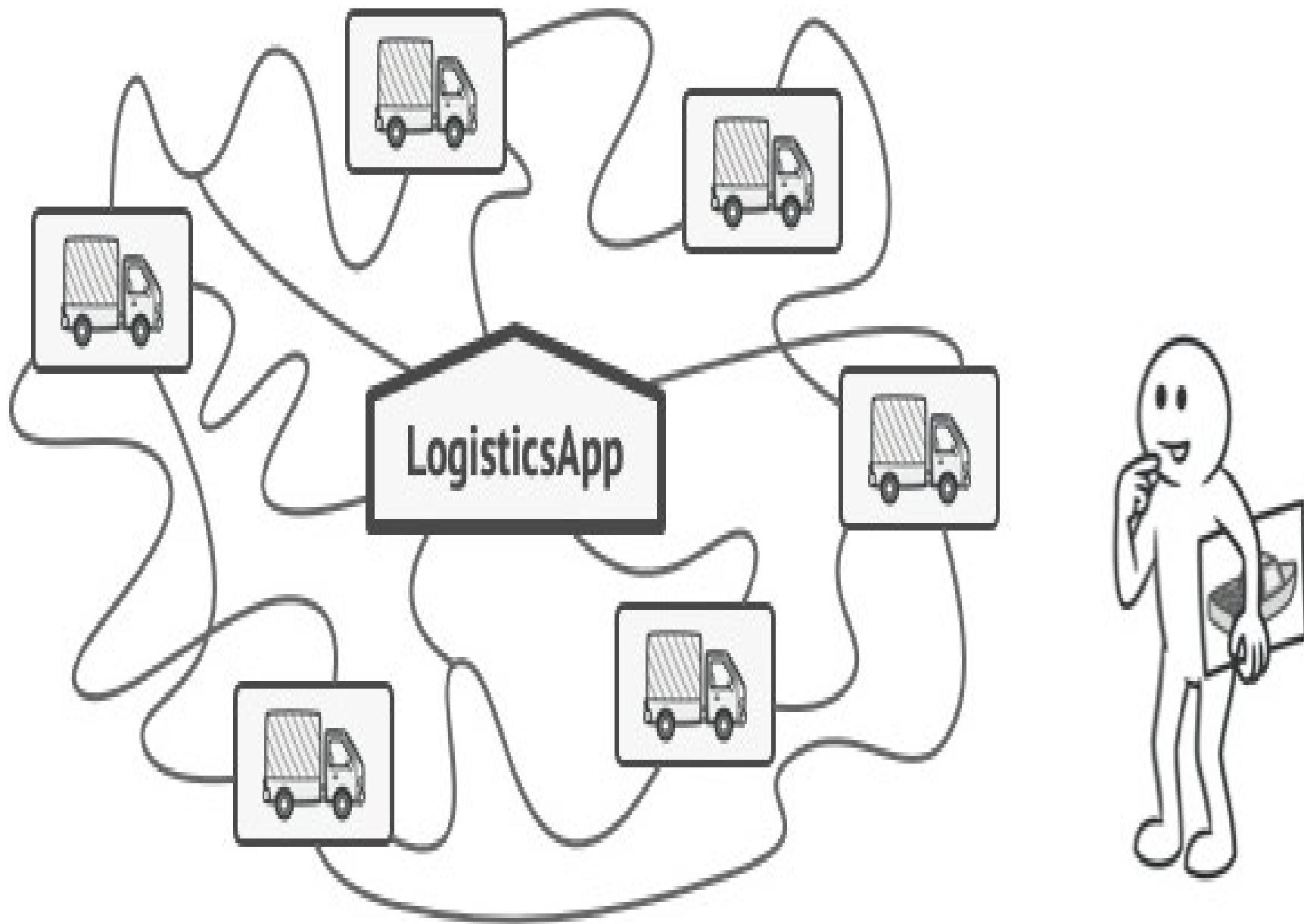
- Factory pattern is one of the most used design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.
- In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

# Intent

- **Factory Method** is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

# Problem

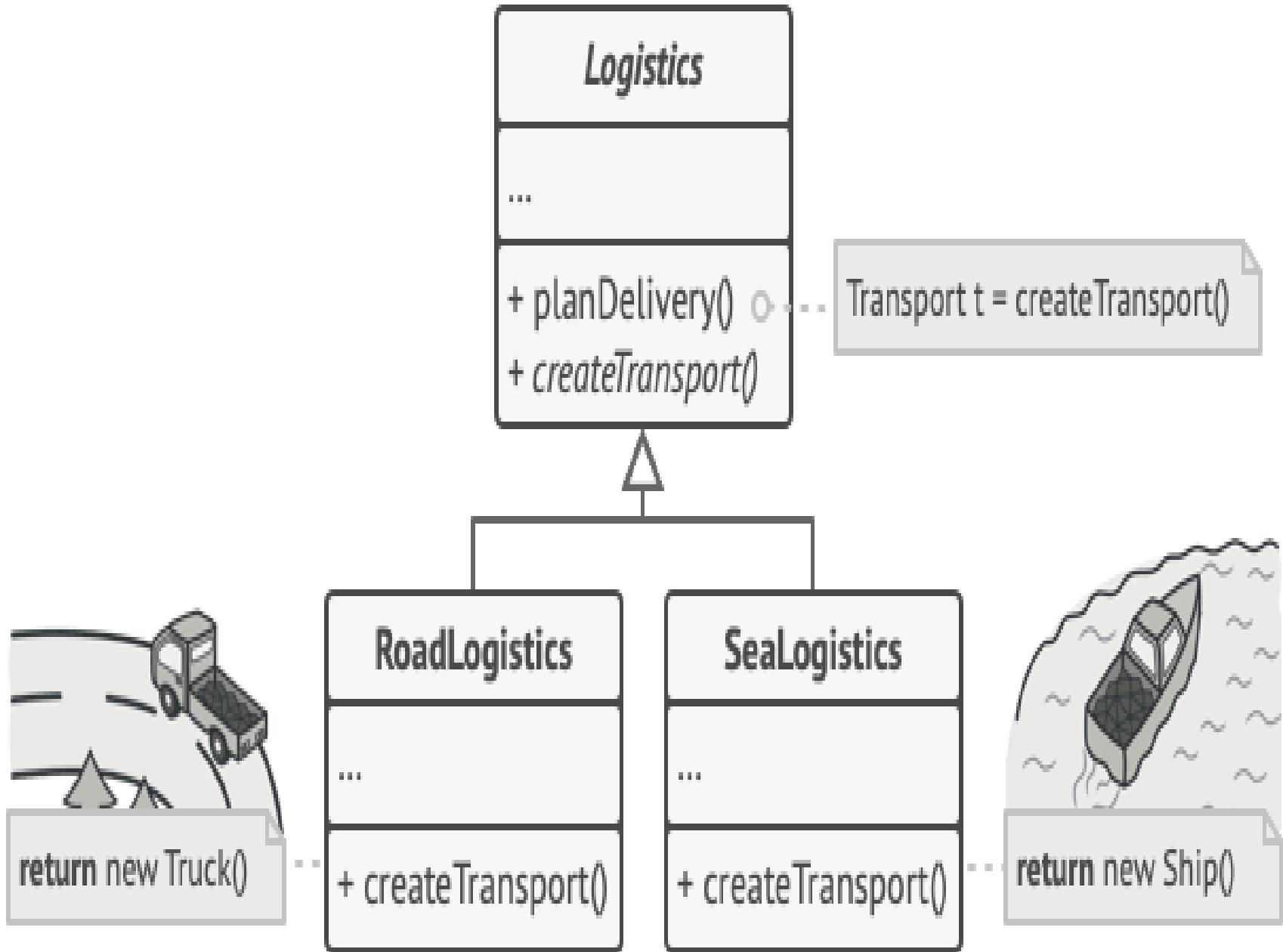
- Imagine that you're creating a logistics management application. The first version of your app can only handle transportation by trucks, so the bulk of your code lives inside the Truck class.
- After a while, your app becomes pretty popular. Each day you receive dozens of requests from sea transportation companies to incorporate sea logistics into the app.
-



- At present, most of your code is coupled to the Truck class.
- Adding Ships into the app would require making changes to the entire codebase. Moreover, if later you decide to add another type of transportation to the app, you will probably need to make all of these changes again.

# Solution

- The Factory Method pattern suggests that you replace direct object construction calls (using the new operator) with calls to a special *factory* method.
- The objects are still created via the new operator, but it's being called from within the factory method.
- Objects returned by a factory method are often referred to as *products*.

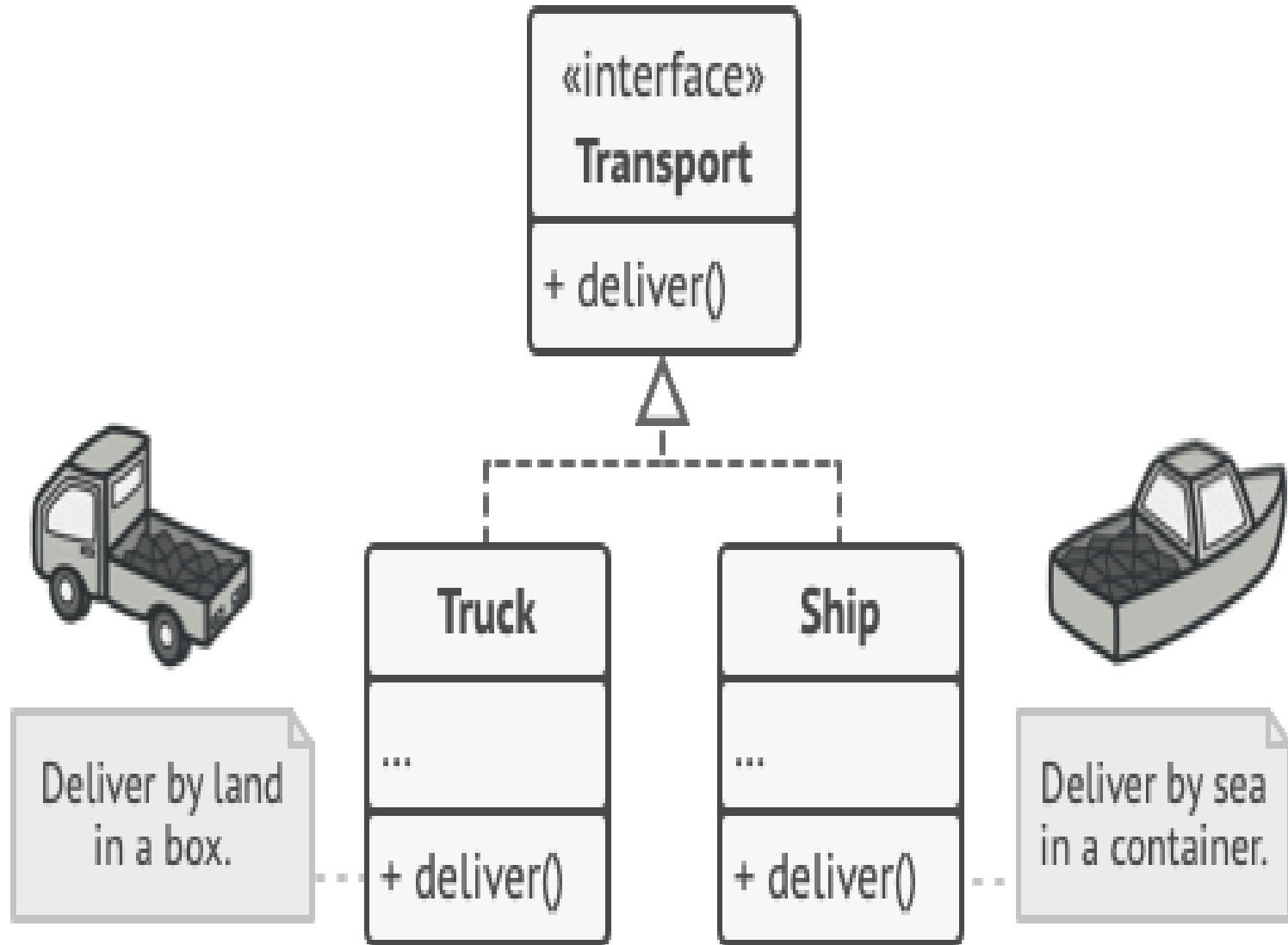


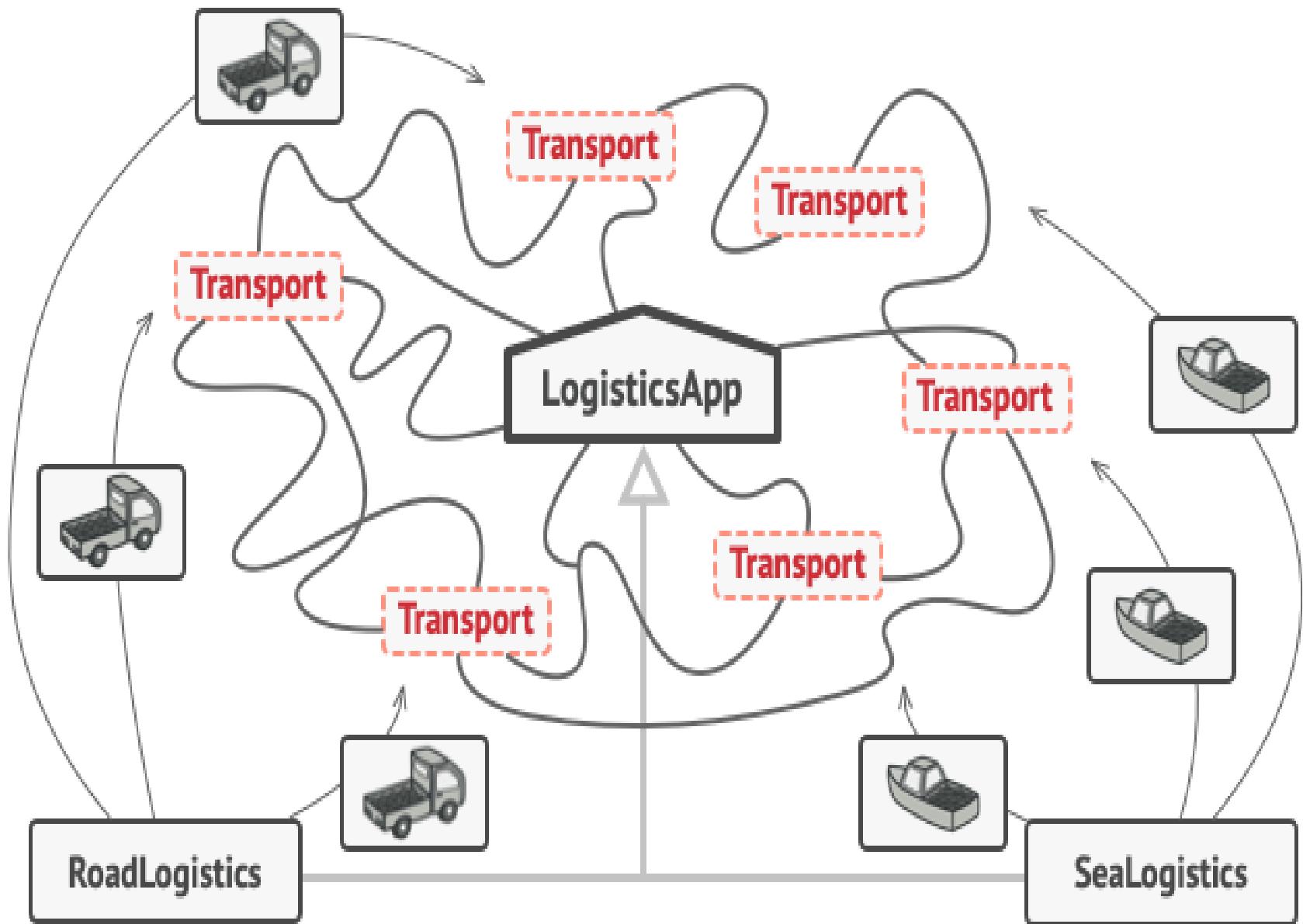
# Advantages

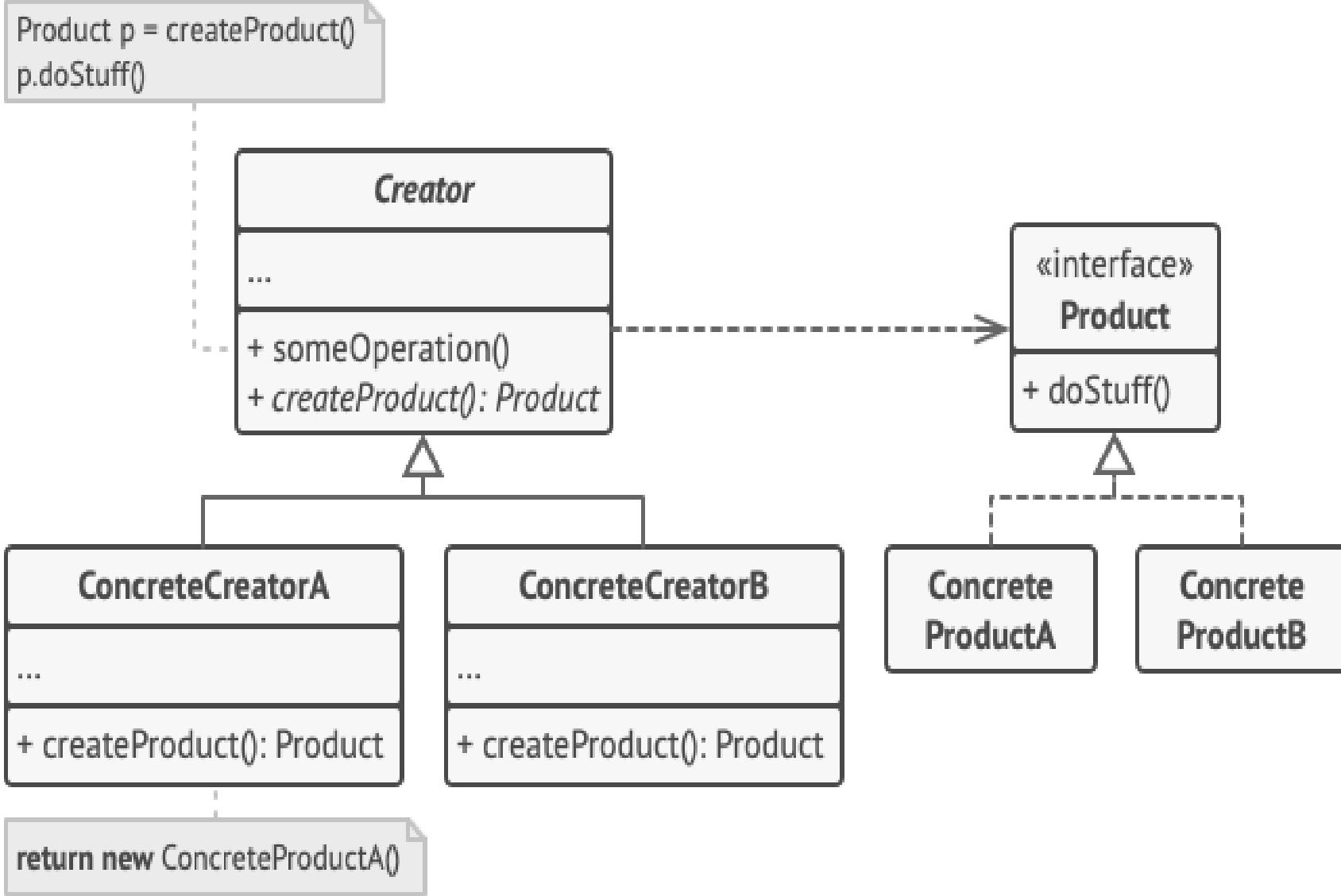
- At first glance, this change may look pointless: we just moved the constructor call from one part of the program to another. However, consider this: now you can override the factory method in a subclass and change the class of products being created by the method.

# Limitation

- There's a slight limitation though: subclasses may return different types of products only if these products have a common base class or interface. Also, the factory method in the base class should have its return type declared as this interface.

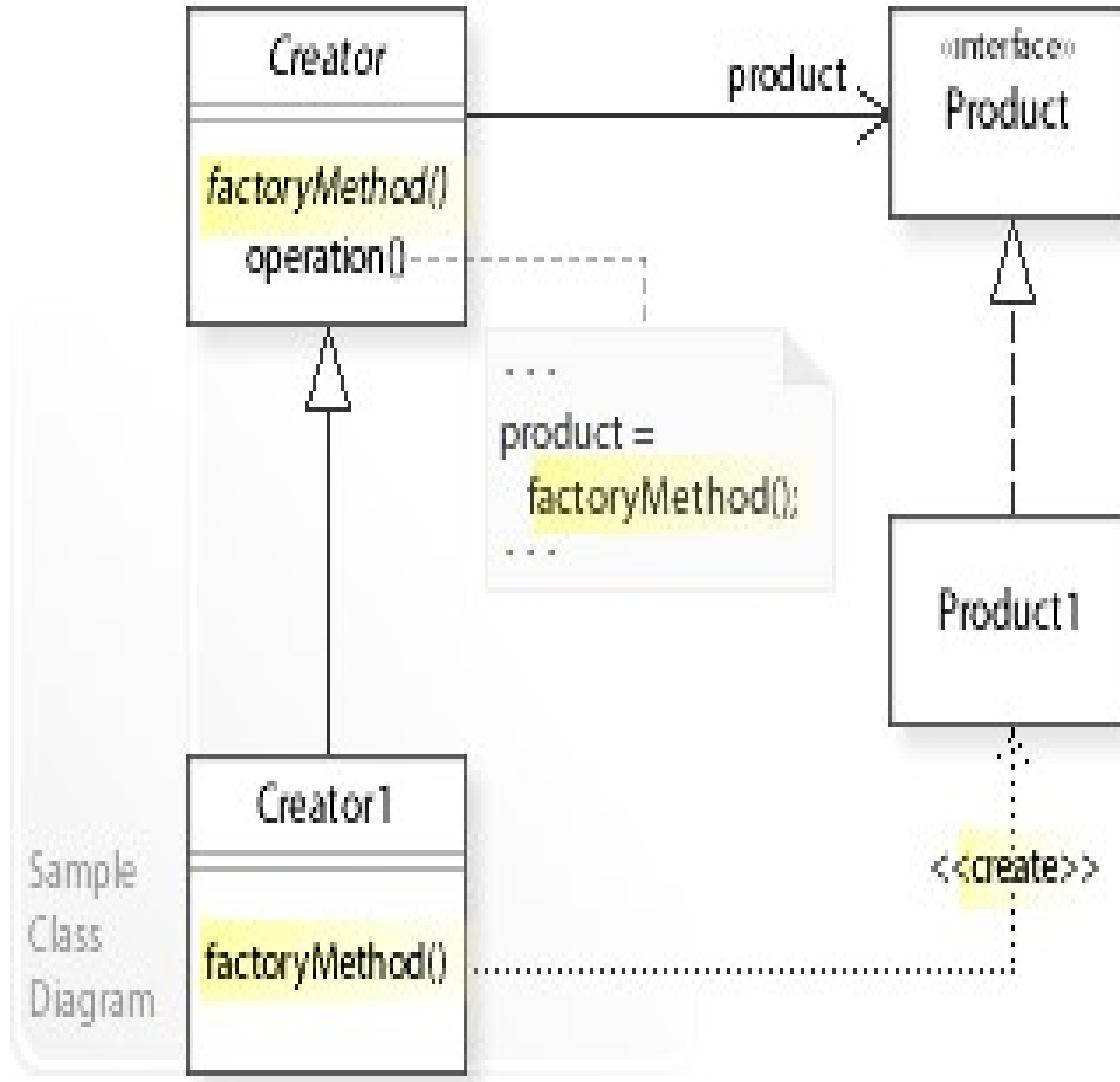


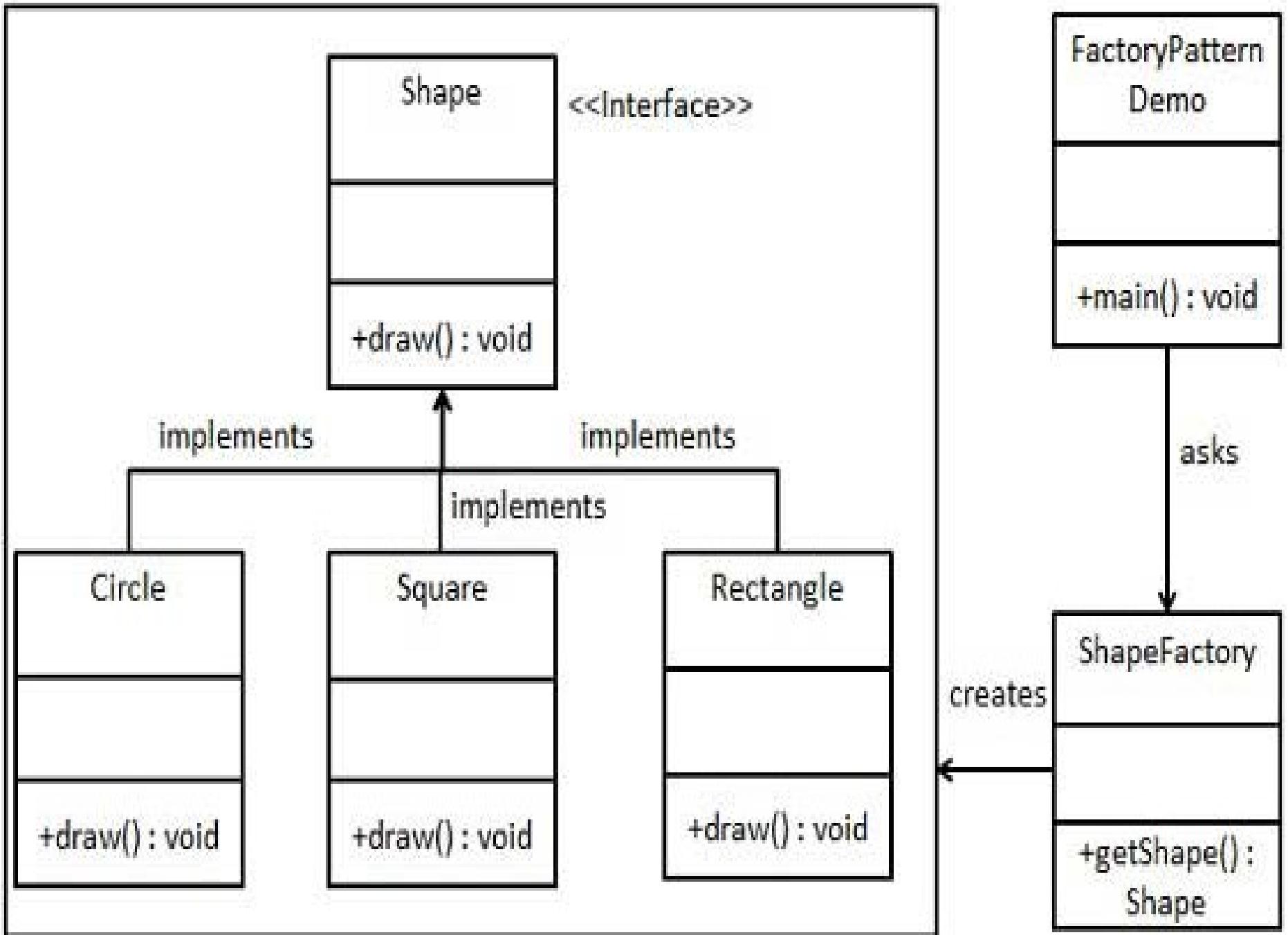




# Example

- *FactoryPatternDemo*, our demo class will use *ShapeFactory* to get a *Shape* object. It will pass information (*CIRCLE* / *RECTANGLE* / *SQUARE*) to *ShapeFactory* to get the type of object it needs.





## Step 1

Create an interface.

*Shape.java*

```
public interface Shape {  
    void draw();  
}
```

## Step 2

Create concrete classes implementing the same interface.

*Rectangle.java*

```
public class Rectangle implements Shape {  
  
    @Override  
    public void draw() {  
        System.out.println("Inside Rectangle::draw() method.");  
    }  
}
```

## *Square.java*

```
public class Square implements Shape {  
  
    @Override  
    public void draw() {  
        System.out.println("Inside Square::draw() method.");  
    }  
}
```

## *Circle.java*

```
public class Circle implements Shape {  
  
    @Override  
    public void draw() {  
        System.out.println("Inside Circle::draw() method.");  
    }  
}
```

## *ShapeFactory.java*

```
public class ShapeFactory {  
  
    //use getShape method to get object of type shape  
    public Shape getShape(String shapeType){  
        if(shapeType == null){  
            return null;  
        }  
        if(shapeType.equalsIgnoreCase("CIRCLE")){  
            return new Circle();  
  
        } else if(shapeType.equalsIgnoreCase("RECTANGLE")){  
            return new Rectangle();  
  
        } else if(shapeType.equalsIgnoreCase("SQUARE")){  
            return new Square();  
        }  
  
        return null;  
    }  
}
```

```
public class FactoryPatternDemo {  
  
    public static void main(String[] args) {  
        ShapeFactory shapeFactory = new ShapeFactory();  
  
        //get an object of Circle and call its draw method.  
        Shape shape1 = shapeFactory.getShape("CIRCLE");  
  
        //call draw method of Circle  
        shape1.draw();  
  
        //get an object of Rectangle and call its draw method.  
        Shape shape2 = shapeFactory.getShape("RECTANGLE");  
  
        //call draw method of Rectangle  
        shape2.draw();  
  
        //get an object of Square and call its draw method.  
        Shape shape3 = shapeFactory.getShape("SQUARE");  
  
        //call draw method of square  
        shape3.draw();  
    }  
}
```

# References

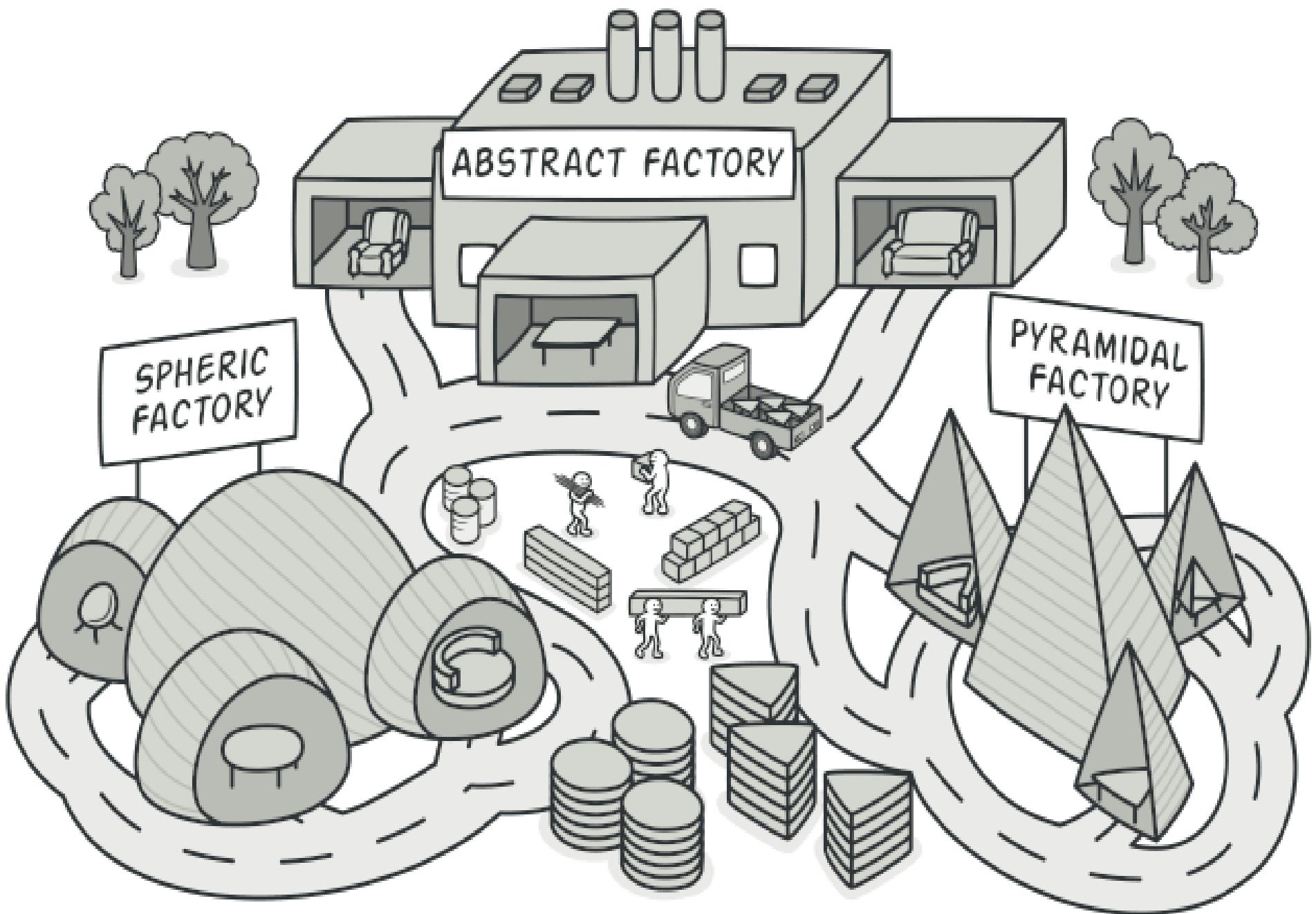
- <https://refactoring.guru/design-patterns/factory-method>
- [https://en.wikipedia.org/wiki/Factory method pattern](https://en.wikipedia.org/wiki/Factory_method_pattern)
- [https://www.tutorialspoint.com/design pattern/factory pattern.htm](https://www.tutorialspoint.com/design_pattern/factory_pattern.htm)

# **Abstract Factory**

Lecture 18

# Intent

- **Abstract Factory** is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.

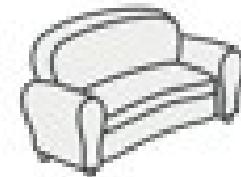


# Problem

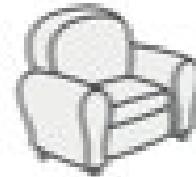
- Imagine that you're creating a furniture shop simulator. Your code consists of classes that represent:
- A family of related products, say: Chair + Sofa + CoffeeTable.
- Several variants of this family. For example, products Chair + Sofa + CoffeeTable are available in these variants:
  - Modern, Victorian, ArtDeco.

## Coffee Table

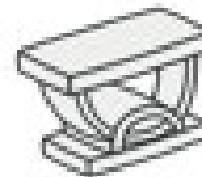
Sofa



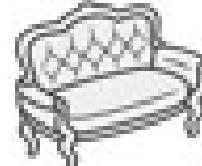
Chair



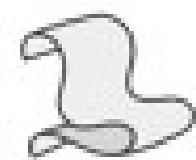
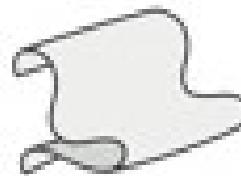
## Art Deco



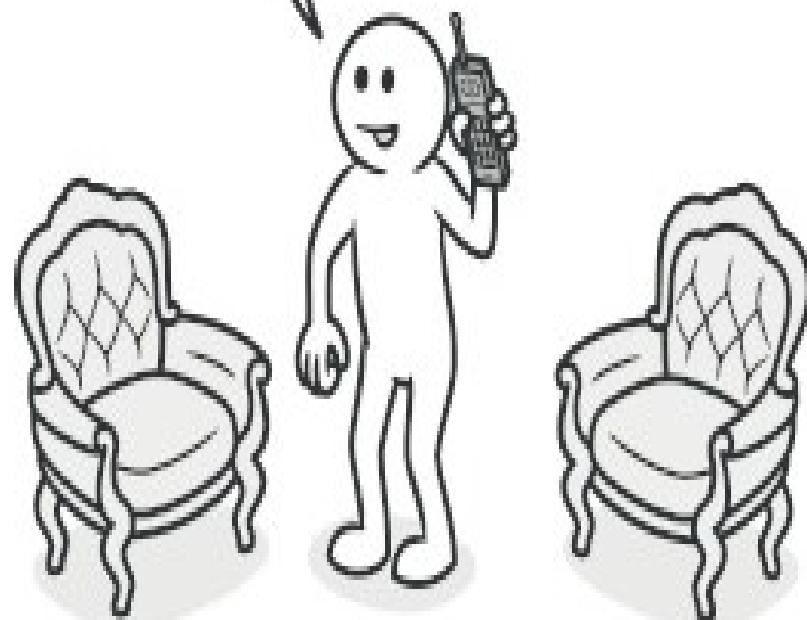
## Victorian



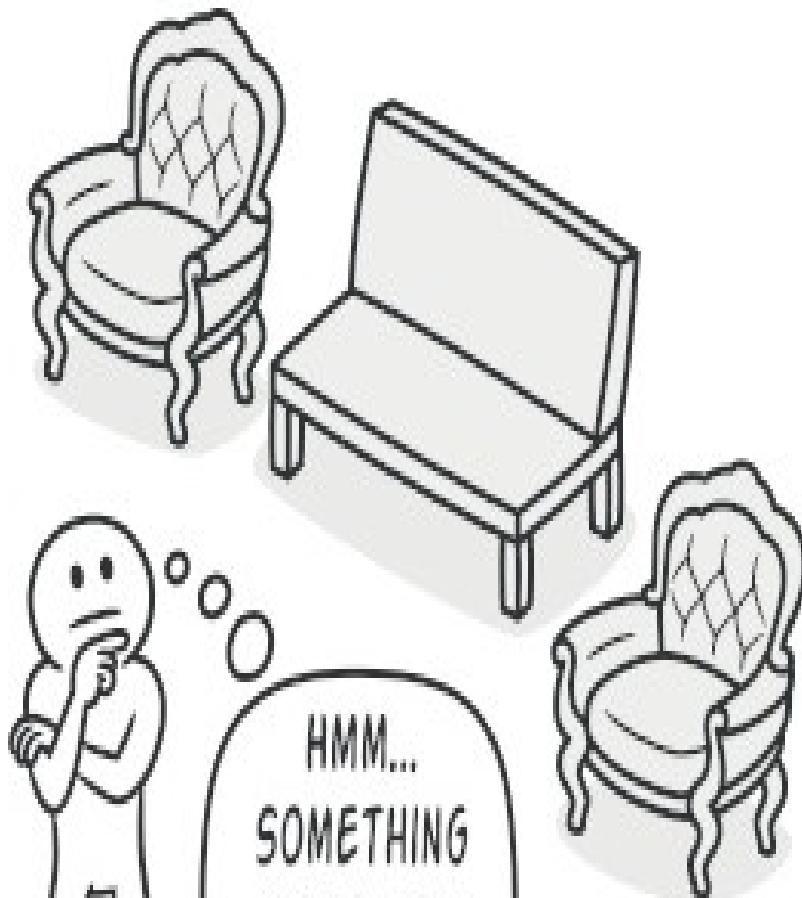
## Modern



LISTEN, I ORDERED SOME CHAIRS LAST WEEK, BUT I GUESS I NEED A SOFA TOO...



HMM... SOMETHING DOES NOT LOOK RIGHT.

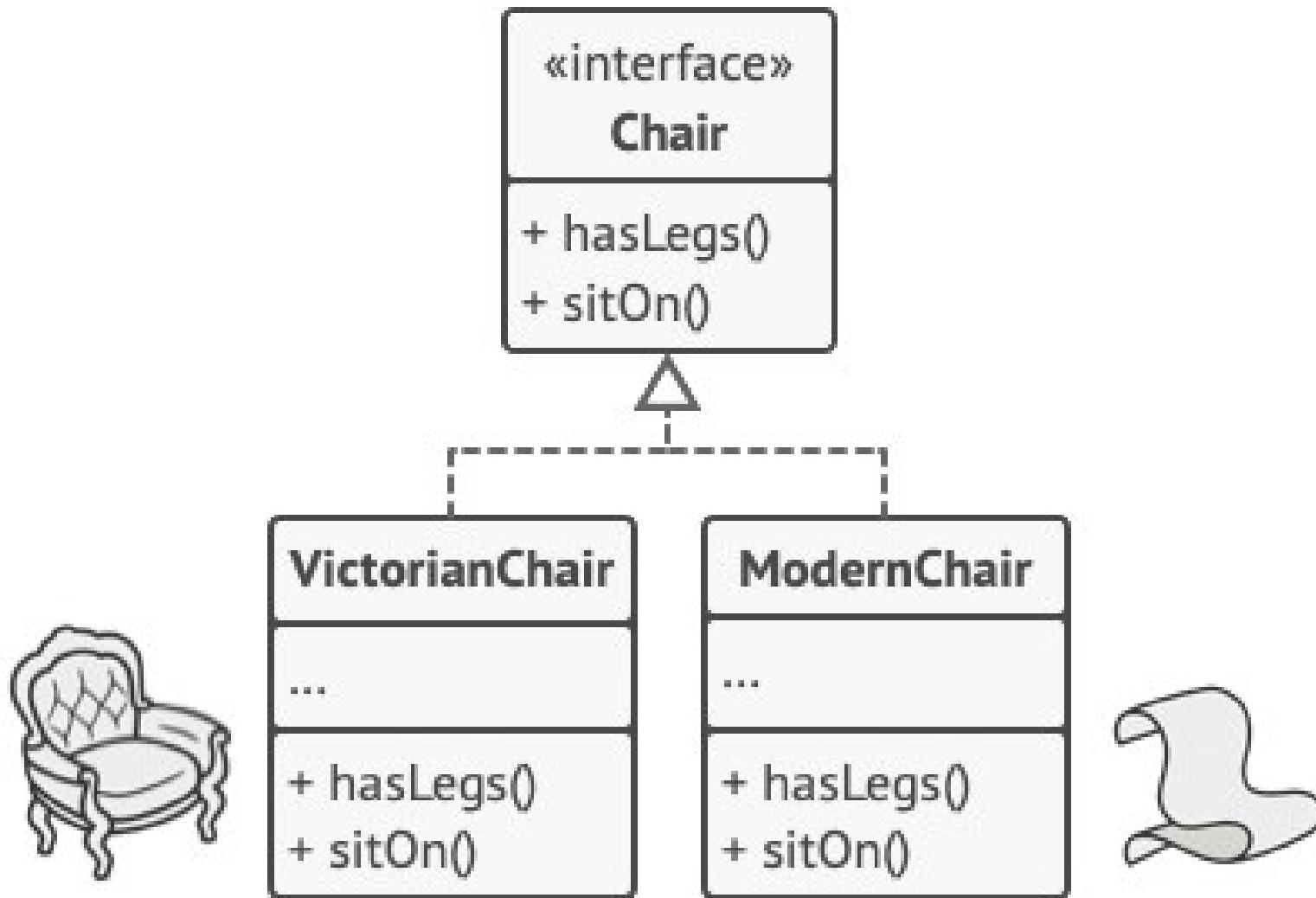


# Problem

- You need a way to create individual furniture objects so that they match other objects of the same family. Customers get quite mad when they receive non-matching furniture.
- Also, you don't want to change existing code when adding new products or families of products to the program. Furniture vendors update their catalogs very often, and you wouldn't want to change the core code each time it happens.

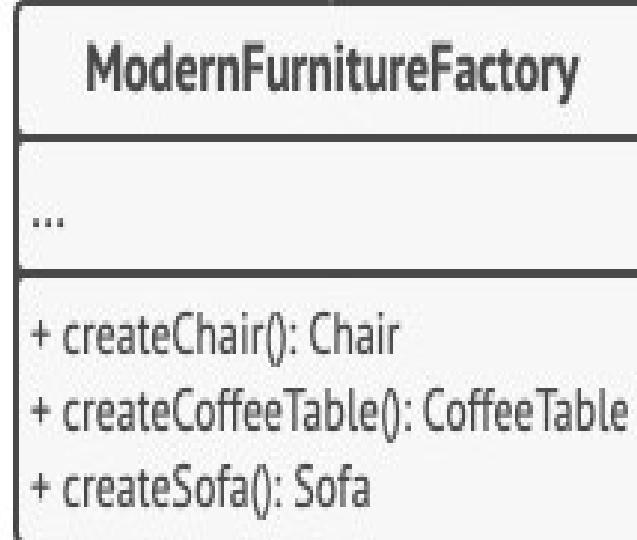
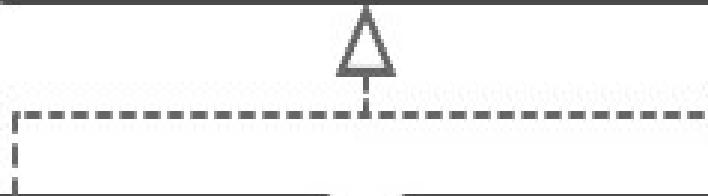
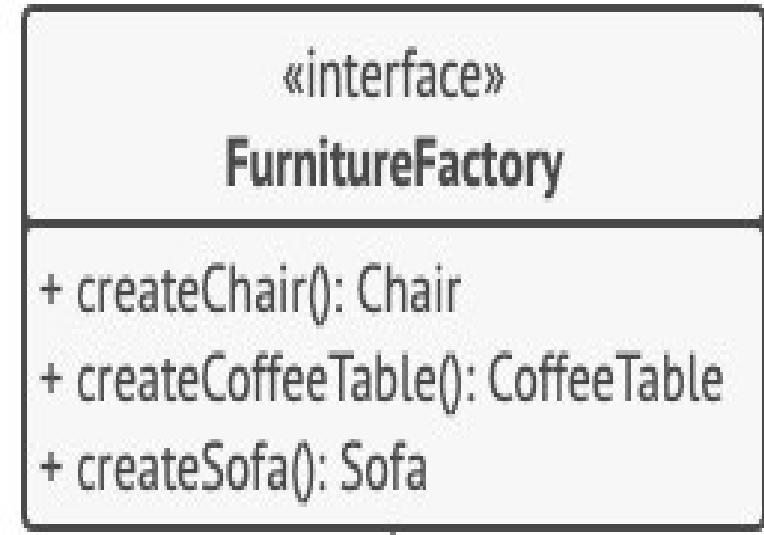
# Solution

- The first thing the Abstract Factory pattern suggests is to explicitly declare interfaces for each distinct product of the product family (e.g., chair, sofa or coffee table).
- Then you can make all variants of products follow those interfaces.
- For example, all chair variants can implement the Chair interface; all coffee table variants can implement the CoffeeTable interface, and so on.



# Solution

- The next move is to declare the Abstract Factory—an interface with a list of creation methods for all products that are part of the product family (for example, `createChair`, `createSofa` and `createCoffeeTable`).
- These methods must return abstract product types represented by the interfaces we extracted previously: `Chair`, `Sofa`, `CoffeeTable` and so on.

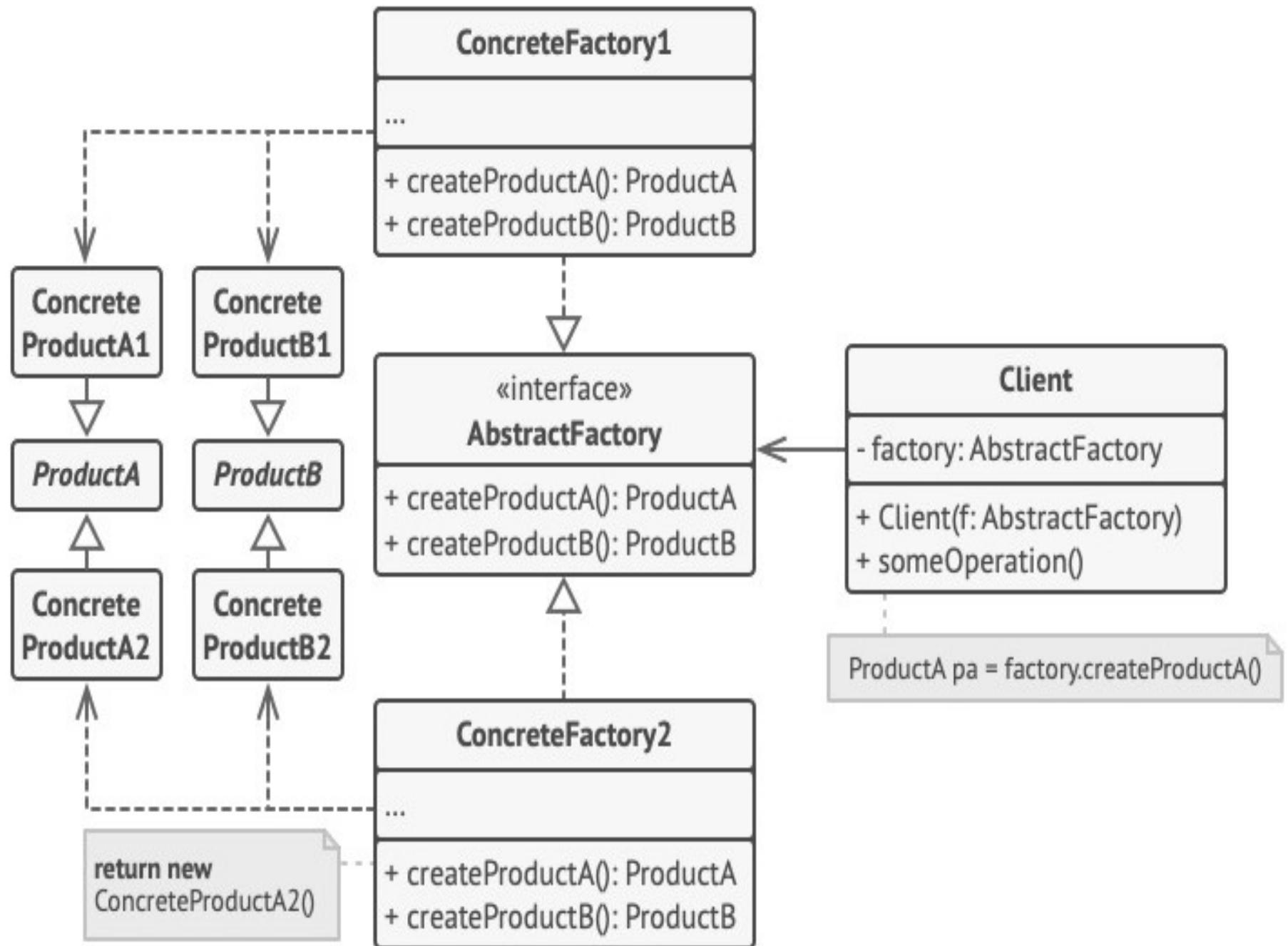


# Explanation

- Now, how about the product variants? For each variant of a product family, we create a separate factory class based on the `AbstractFactory` interface.
- A factory is a class that returns products of a particular kind.
- For example, the `ModernFurnitureFactory` can only create `ModernChair`, `ModernSofa` and `ModernCoffeeTable` objects.

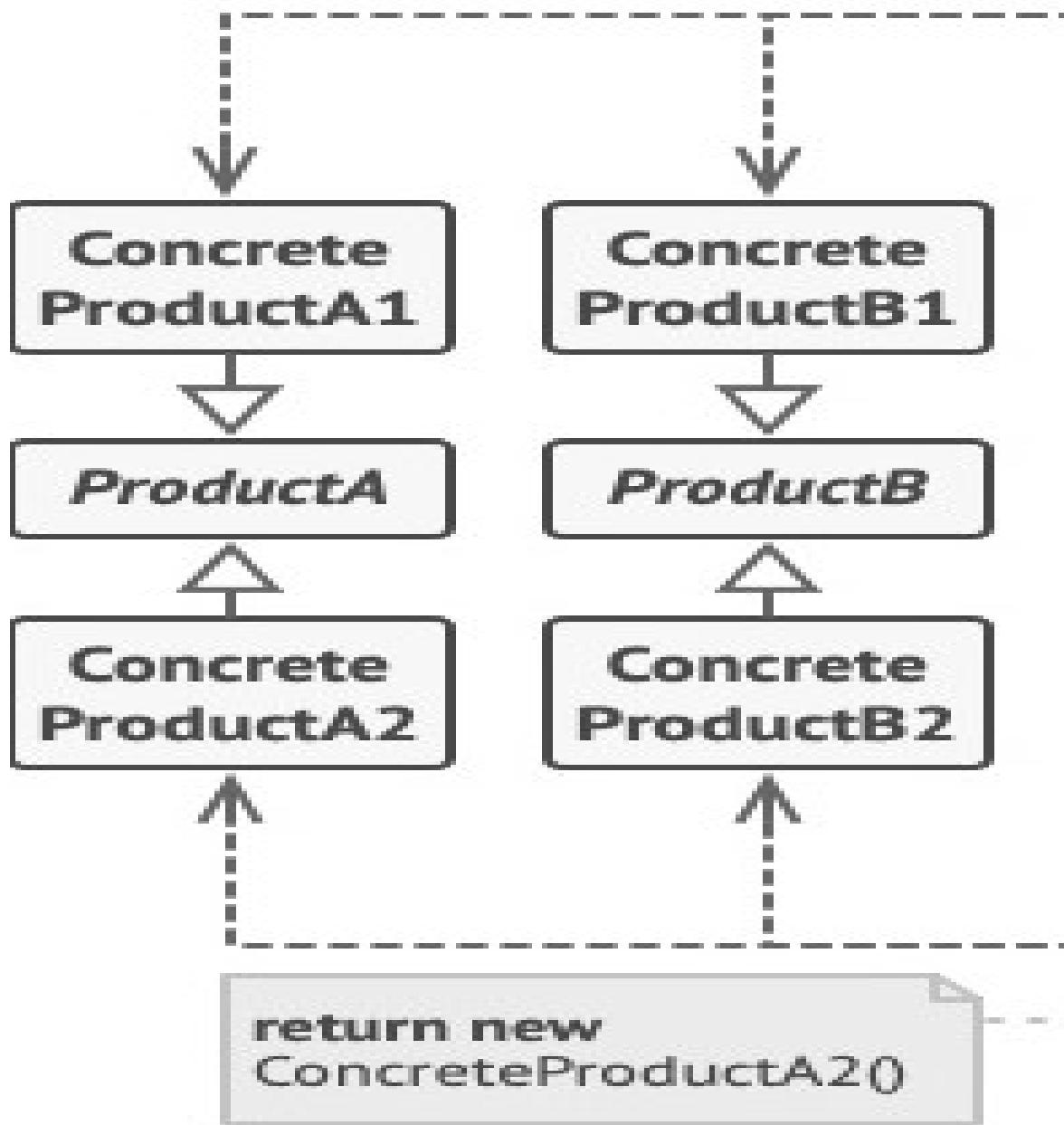
# Explanation

- The client code has to work with both factories and products via their respective abstract interfaces.
- This lets you change the type of a factory that you pass to the client code, as well as the product variant that the client code receives, without breaking the actual client code.



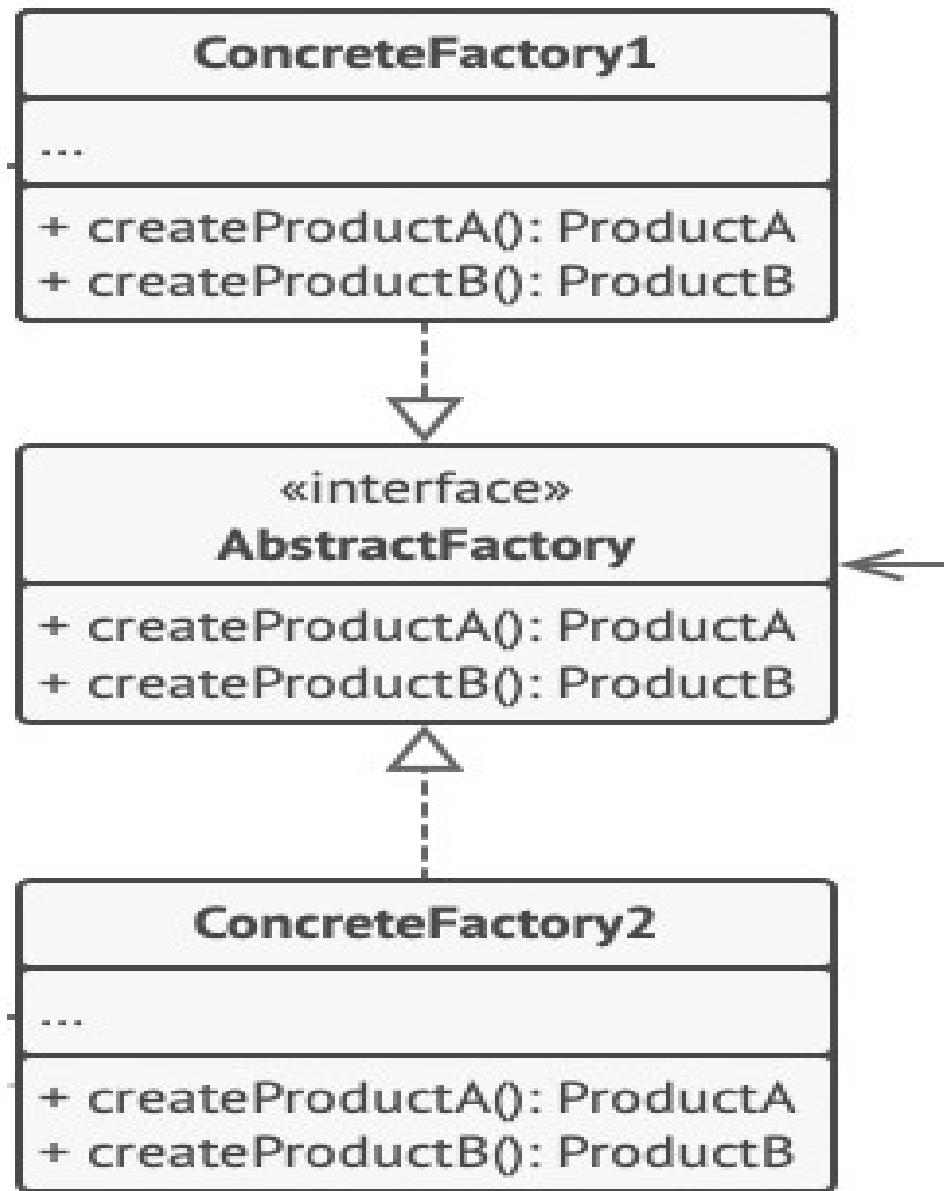
# Structure

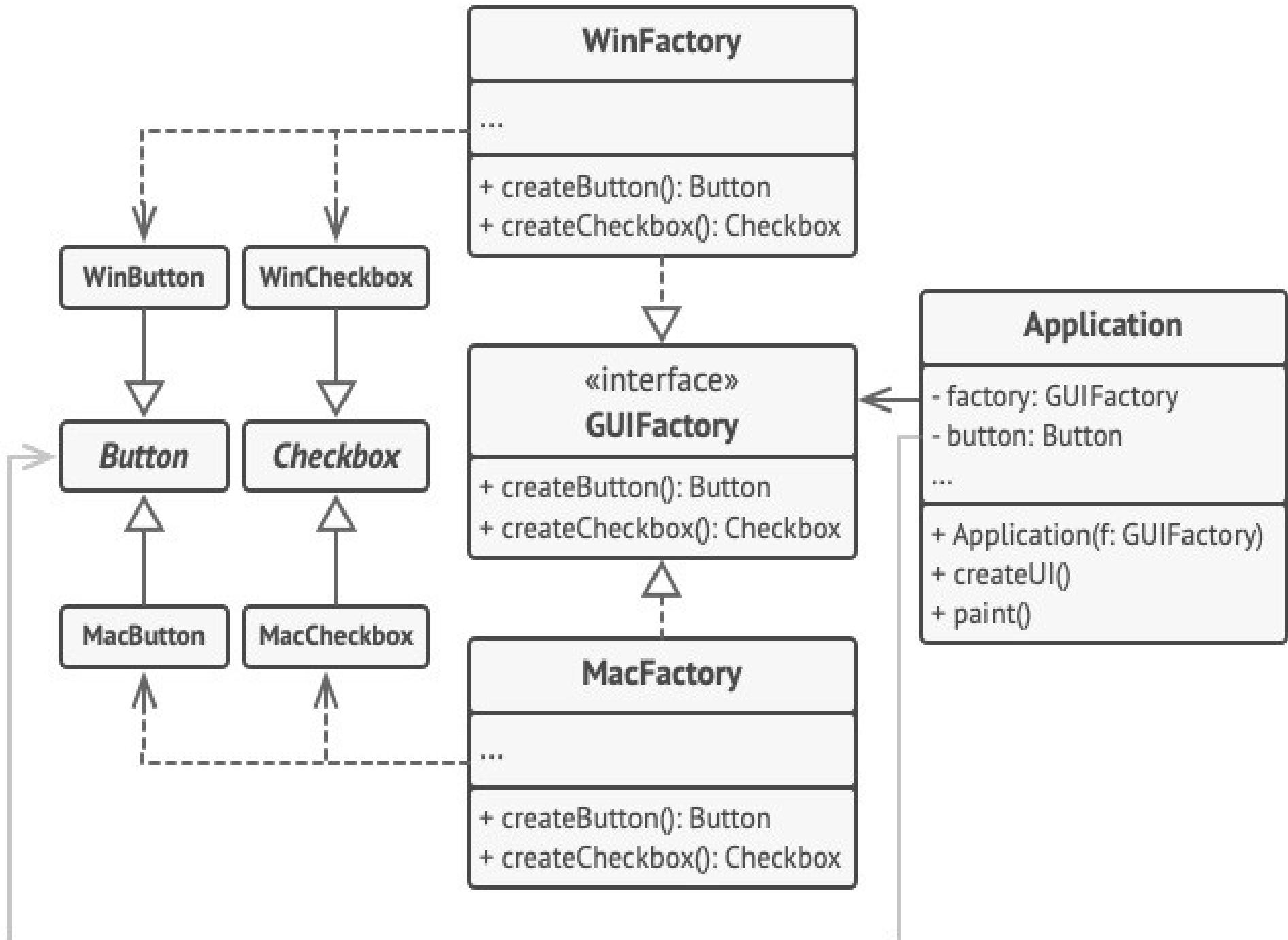
- Step 1
  - **Abstract Products** declare interfaces for a set of distinct but related products which make up a product family.
- Step 2
  - **Concrete Products** are various implementations of abstract products, grouped by variants. Each abstract product (chair/sofa) must be implemented in all given variants (Victorian/Modern).



# Structure

- Step 3
  - The **Abstract Factory** interface declares a set of methods for creating each of the abstract products.
- Step 4
  - **Concrete Factories** implement creation methods of the abstract factory. Each concrete factory corresponds to a specific variant of products and creates only those product variants.





# Pseudocode

```
interface GUIFactory is
    method createButton():Button
    method createCheckbox():Checkbox

class WinFactory implements GUIFactory is
    method createButton():Button is
        return new WinButton()
    method createCheckbox():Checkbox is
        return new WinCheckbox()

class MacFactory implements GUIFactory is
    method createButton():Button is
        return new MacButton()
    method createCheckbox():Checkbox is
        return new MacCheckbox()
```

```
// interface.  
  
interface Button is  
    method paint()  
  
// Concrete products are created by corresponding concrete  
// factories.  
  
class WinButton implements Button is  
    method paint() is  
        // Render a button in Windows style.  
  
class MacButton implements Button is  
    method paint() is  
        // Render a button in macOS style.
```

```
interface Checkbox is
    method paint()

class WinCheckbox implements Checkbox is
    method paint() is
        // Render a checkbox in Windows style.

class MacCheckbox implements Checkbox is
    method paint() is
        // Render a checkbox in macOS style.
```

```
class Application is
    private field factory: GUIFactory
    private field button: Button
    constructor Application(factory: GUIFactory) is
        this.factory = factory
    method createUI() is
        this.button = factory.createButton()
    method paint() is
        button.paint()
```

```
class ApplicationConfigurator is
    method main() is
        config = readApplicationConfigFile()

        if (config.OS == "Windows") then
            factory = new WinFactory()
        else if (config.OS == "Mac") then
            factory = new MacFactory()
        else
            throw new Exception("Error! Unknown operating system.")

        Application app = new Application(factory)
```

# References

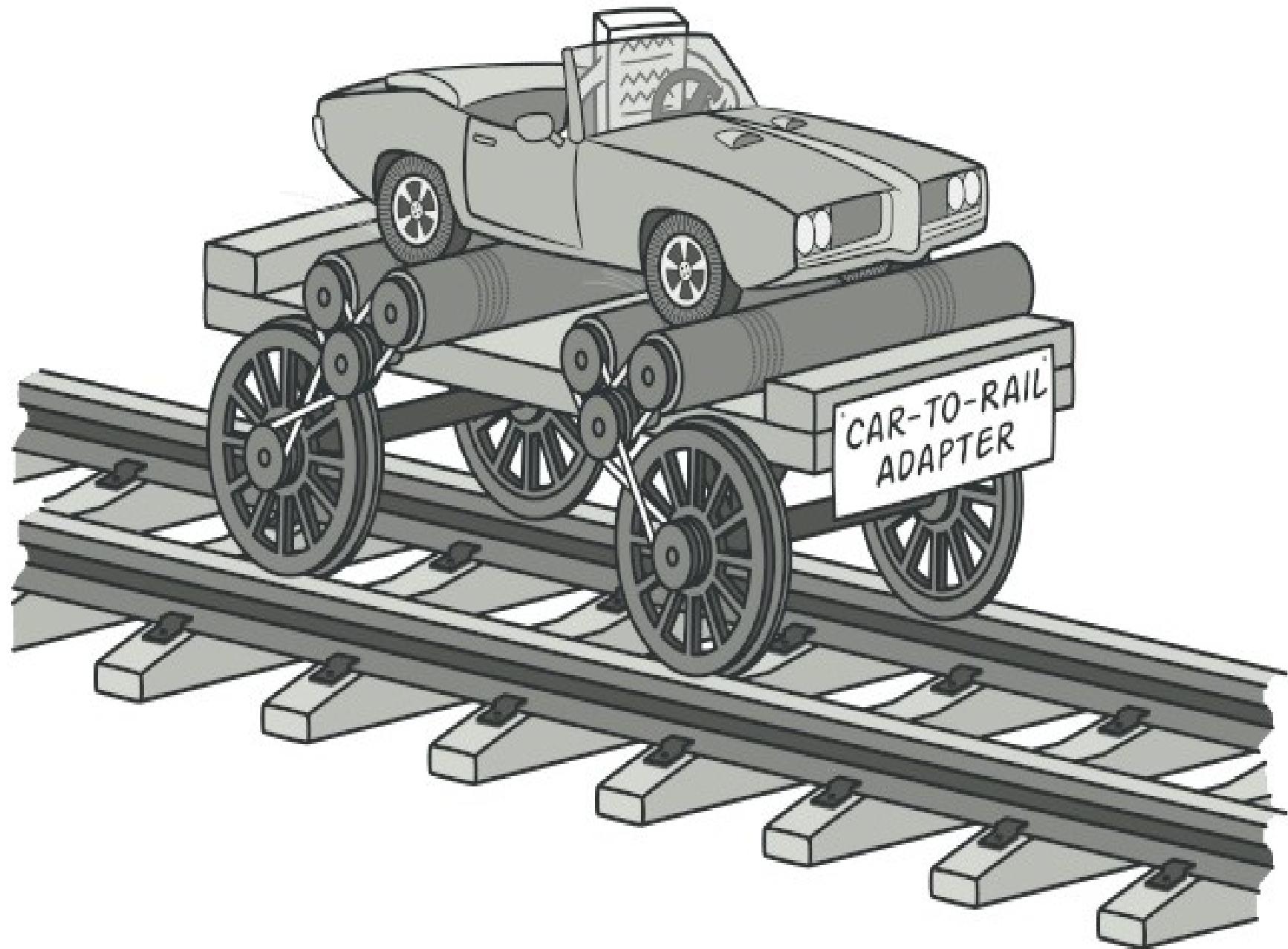
- <https://refactoring.guru/design-patterns/abstract-factory>

# Adaptor

Lecture 19

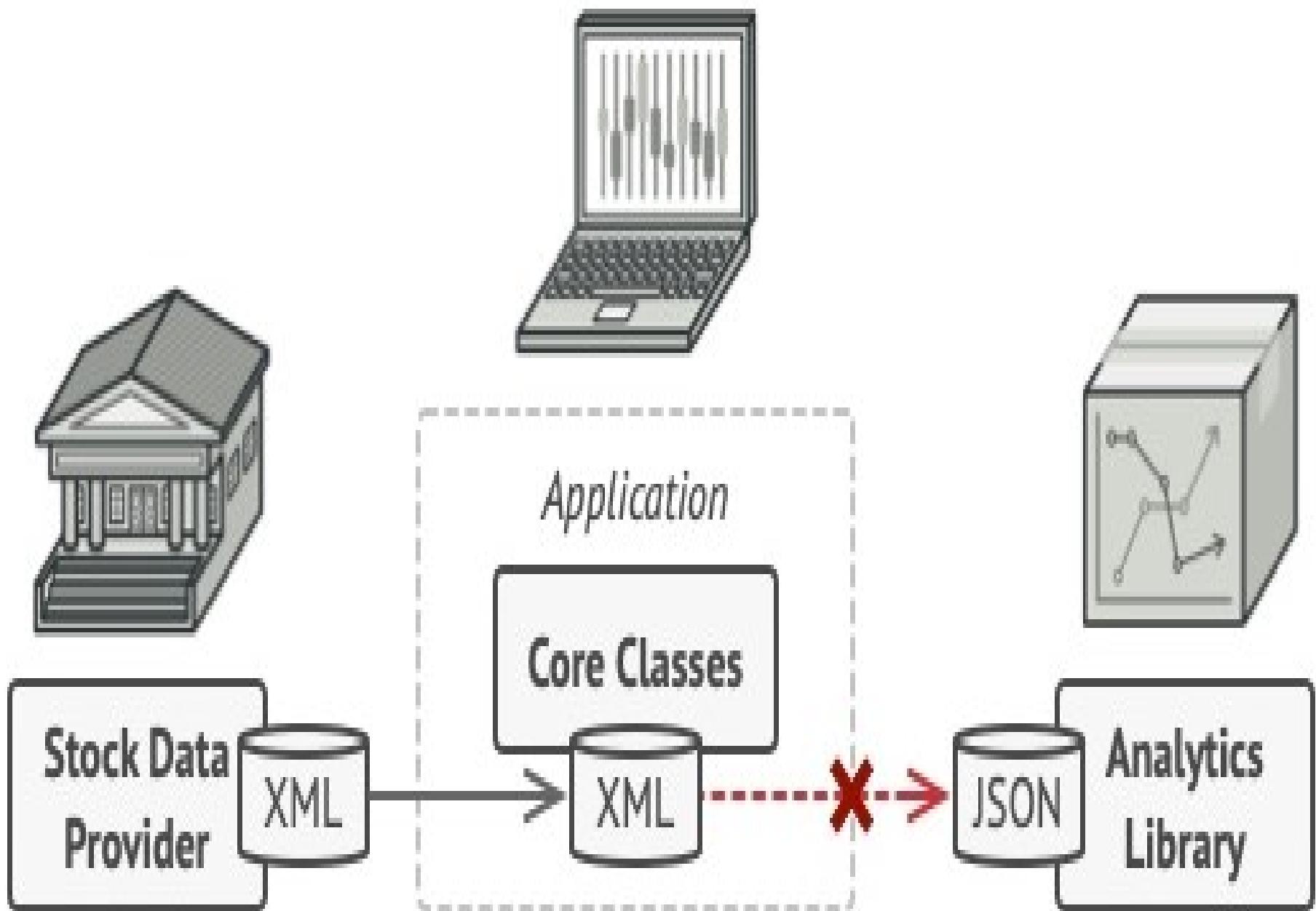
# Intent

- **Adapter** is a structural design pattern that allows objects with incompatible interfaces to collaborate.



# Problem

- Imagine that you're creating a stock market monitoring app. The app downloads the stock data from multiple sources in XML format and then displays nice-looking charts and diagrams for the user.
- At some point, you decide to improve the app by integrating a smart 3rd-party analytics library. But there's a catch: the analytics library only works with data in JSON format.



# Problem

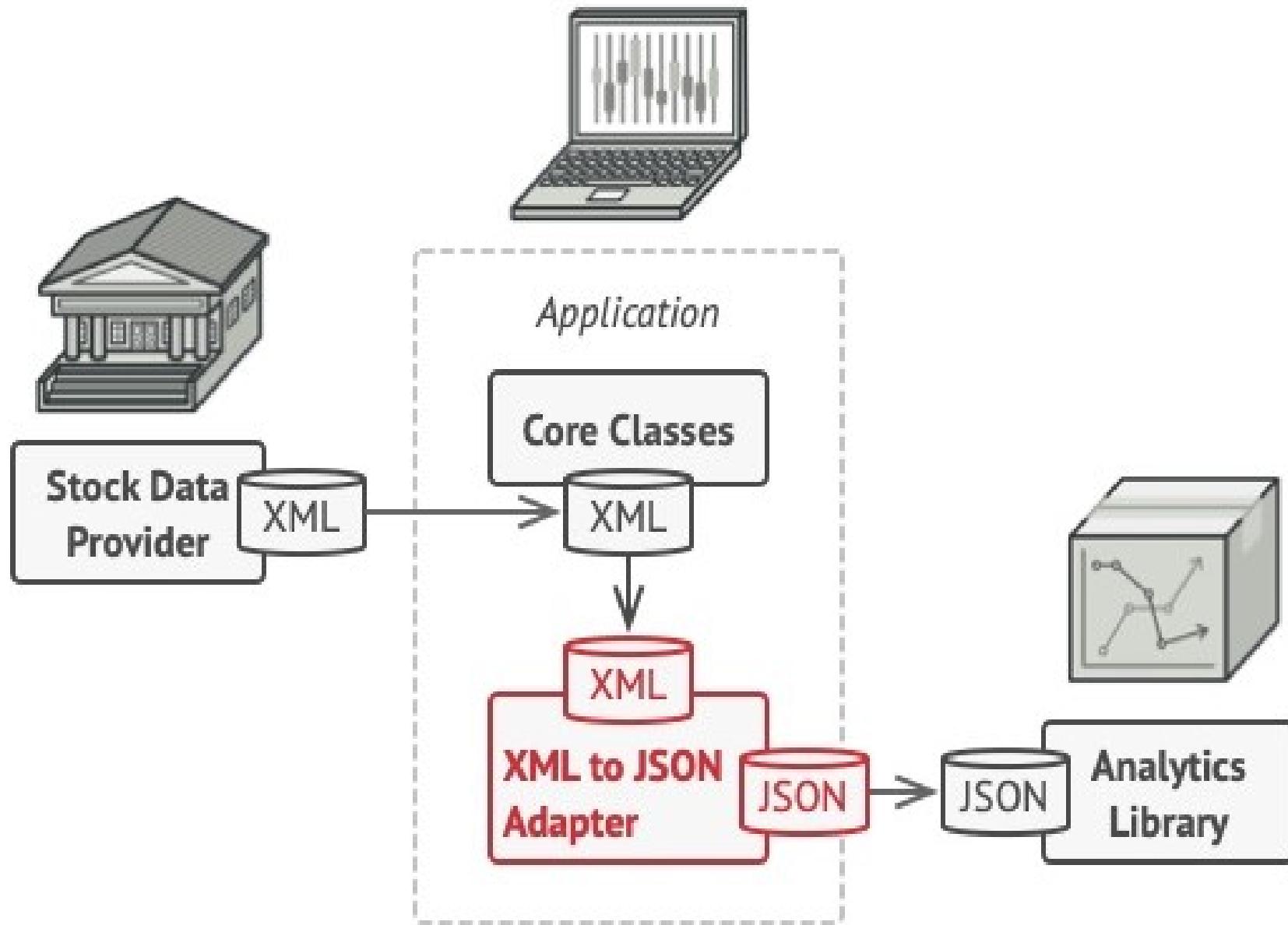
- You could change the library to work with XML.
- However, this might break some existing code that relies on the library.
- And worse, you might not have access to the library's source code in the first place, making this approach impossible.

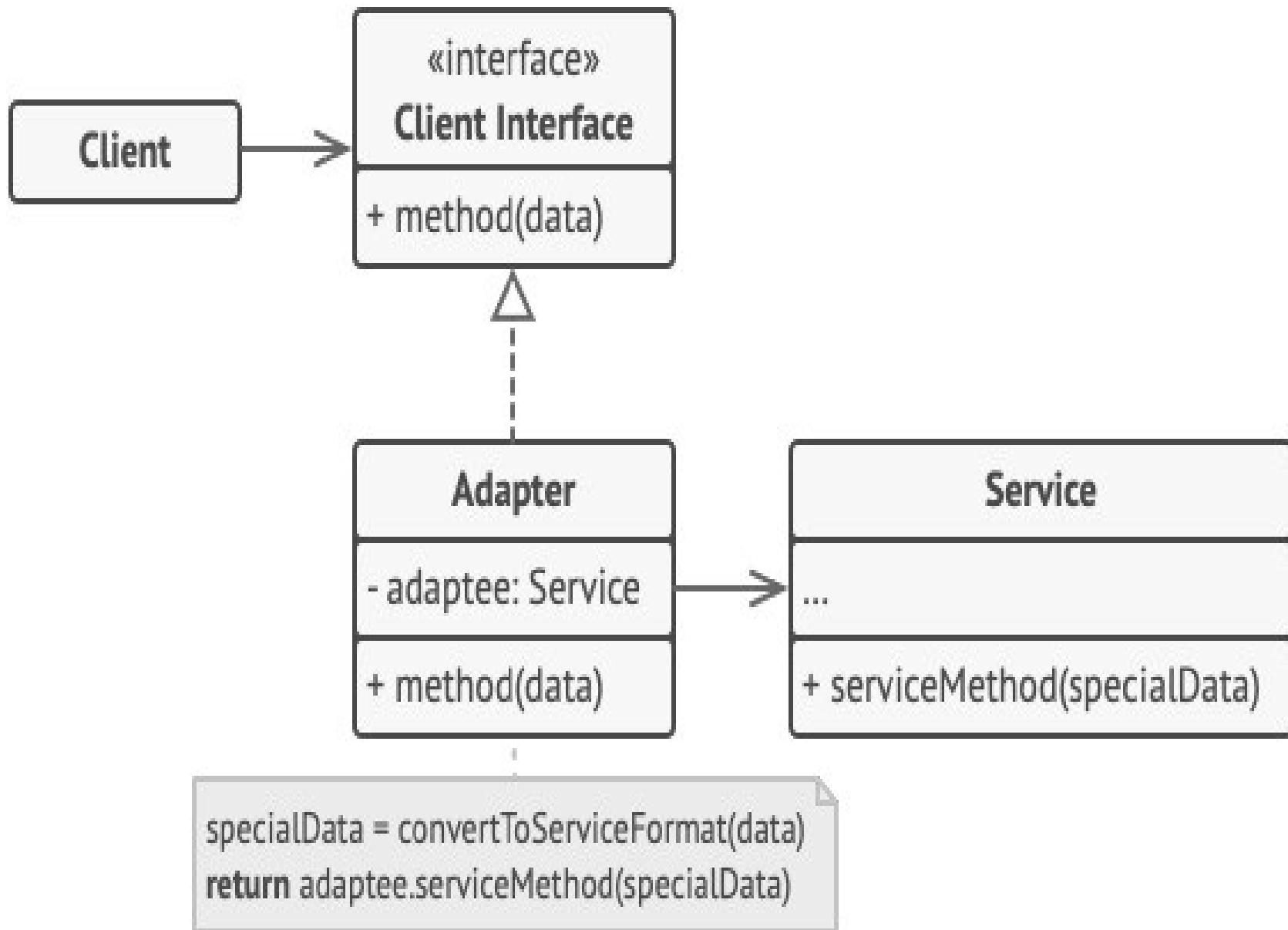
# Solution

- You can create an *adapter*. This is a special object that converts the interface of one object so that another object can understand it.

# Solution

- Adapters can not only convert data into various formats but can also help objects with different interfaces collaborate. Here's how it works:
  - The adapter gets an interface, compatible with one of the existing objects.
  - Using this interface, the existing object can safely call the adapter's methods.
  - Upon receiving a call, the adapter passes the request to the second object, but in a format and order that the second object expects.





# Structure

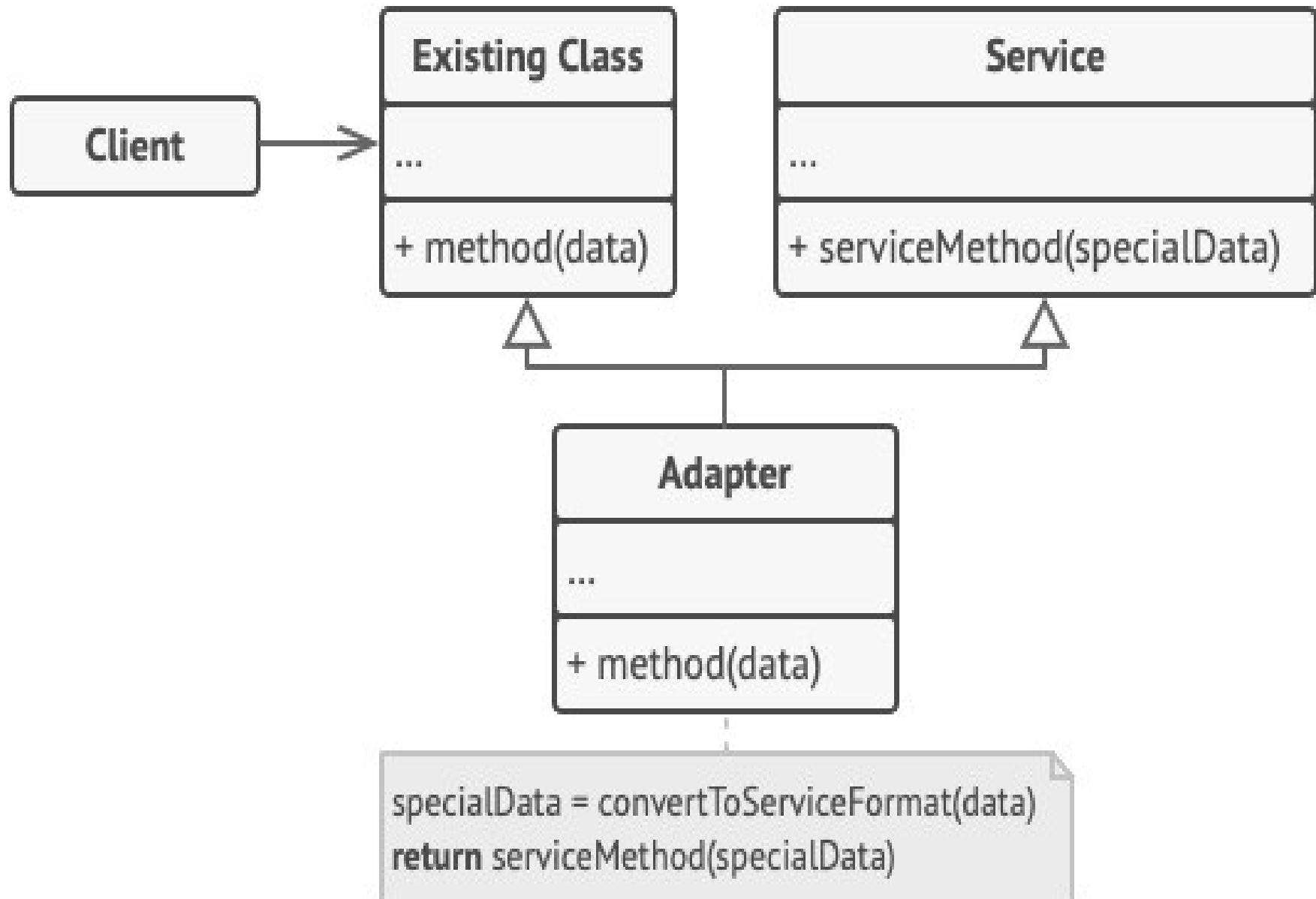
- The **Client** is a class that contains the existing business logic of the program.
- The **Client Interface** describes a protocol that other classes must follow to be able to collaborate with the client code.
- The **Service** is some useful class (usually 3rd-party or legacy). The client can't use this class directly because it has an incompatible interface.

# Explanation

- The **Adapter** is a class that's able to work with both the client and the service: it implements the client interface, while wrapping the service object. The adapter receives calls from the client via the adapter interface and translates them into calls to the wrapped service object in a format it can understand.

# Explanation

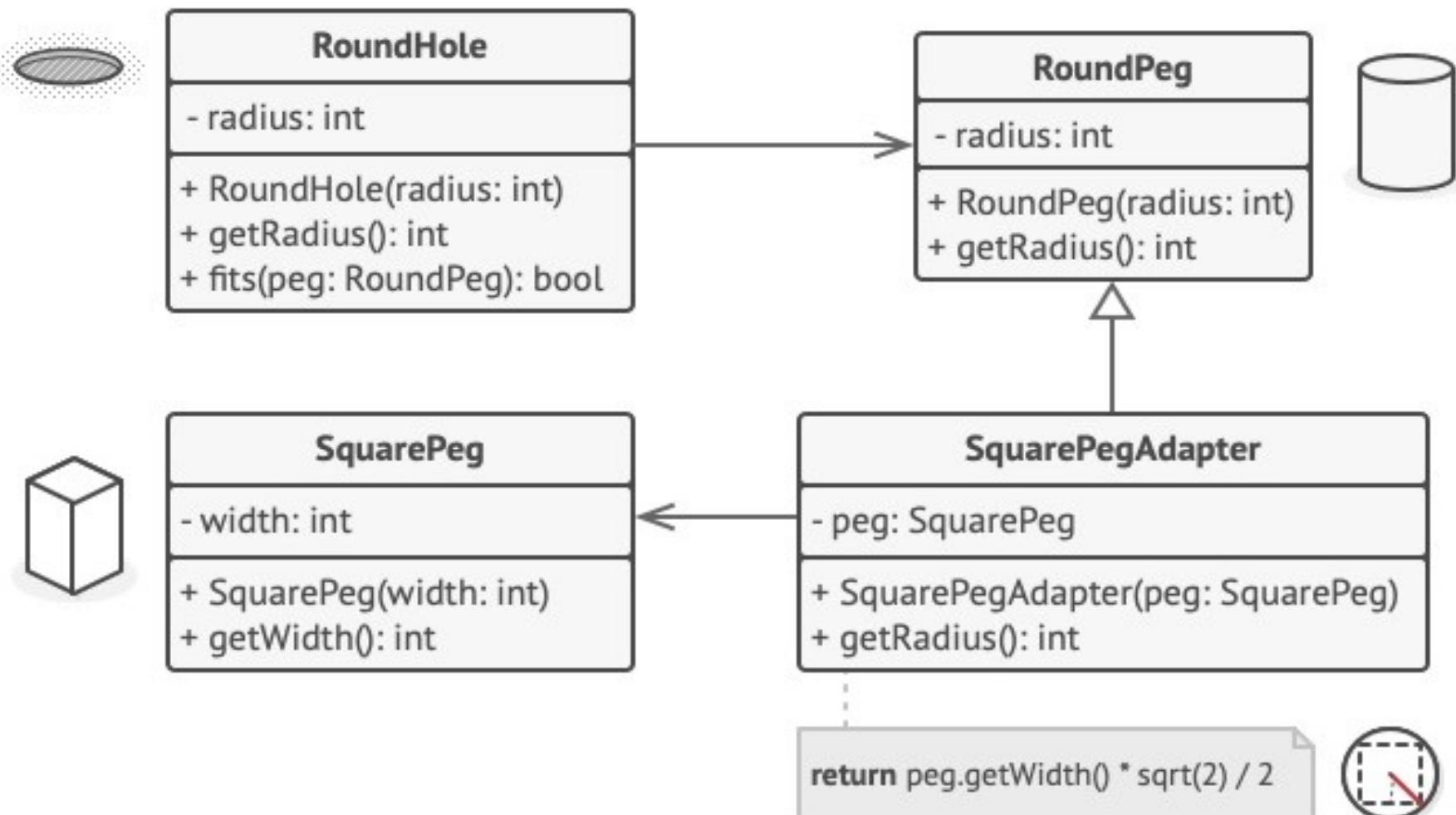
- The client code doesn't get coupled to the concrete adapter class as long as it works with the adapter via the client interface. Thanks to this, you can introduce new types of adapters into the program without breaking the existing client code. This can be useful when the interface of the service class gets changed or replaced: you can just create a new adapter class without changing the client code.



# Example

- This example of the **Adapter** pattern is based on the classic conflict between square pegs and round holes.
- The Adapter pretends to be a round peg, with a radius equal to a half of the square's diameter (in other words, the radius of the smallest circle that can accommodate the square peg).





```
// Say you have two classes with compatible interfaces:  
// RoundHole and RoundPeg.  
class RoundHole is  
    constructor RoundHole(radius) { ... }  
  
    method getRadius() is  
        // Return the radius of the hole.  
  
    method fits(peg: RoundPeg) is  
        return this.getRadius() >= peg.getRadius()  
  
class RoundPeg is  
    constructor RoundPeg(radius) { ... }  
  
    method getRadius() is  
        // Return the radius of the peg.
```

```
// But there's an incompatible class: SquarePeg.  
class SquarePeg is  
    constructor SquarePeg(width) { ... }  
  
    method getWidth() is  
        // Return the square peg width.  
  
// An adapter class lets you fit square pegs into round holes.  
// It extends the RoundPeg class to let the adapter objects act  
// as round pegs.  
class SquarePegAdapter extends RoundPeg is  
    // In reality, the adapter contains an instance of the  
    // SquarePeg class.  
    private field peg: SquarePeg  
  
constructor SquarePegAdapter(peg: SquarePeg) is  
    this.peg = peg  
  
method getRadius() is  
    // The adapter pretends that it's a round peg with a  
    // radius that could fit the square peg that the adapter  
    // actually wraps.  
    return peg.getWidth() * Math.sqrt(2) / 2
```

```
// Somewhere in client code.

hole = new RoundHole(5)
rpeg = new RoundPeg(5)
hole.fits(rpeg) // true

small_sqpeg = new SquarePeg(5)
large_sqpeg = new SquarePeg(10)
hole.fits(small_sqpeg) // this won't compile (incompatible types)

small_sqpeg_adapter = new SquarePegAdapter(small_sqpeg)
large_sqpeg_adapter = new SquarePegAdapter(large_sqpeg)
hole.fits(small_sqpeg_adapter) // true
hole.fits(large_sqpeg_adapter) // false
```

# How to Implement

1. Make sure that you have at least two classes with incompatible interfaces:
  - A useful *service* class, which you can't change (often 3rd-party, legacy or with lots of existing dependencies).
  - One or several *client* classes that would benefit from using the service class.
2. Declare the client interface and describe how clients communicate with the service.
3. Create the adapter class and make it follow the client interface. Leave all the methods empty for now.

4. Add a field to the adapter class to store a reference to the service object. The common practice is to initialize this field via the constructor, but sometimes it's more convenient to pass it to the adapter when calling its methods.
  5. One by one, implement all methods of the client interface in the adapter class. The adapter should delegate most of the real work to the service object, handling only the interface or data format conversion.
  6. Clients should use the adapter via the client interface. This will let you change or extend the adapters without affecting the client code.
-

# Pros and Cons

- ✓ *Single Responsibility Principle.* You can separate the interface or data conversion code from the primary business logic of the program.
- ✓ *Open/Closed Principle.* You can introduce new types of adapters into the program without breaking the existing client code, as long as they work with the adapters through the client interface.
- ✗ The overall complexity of the code increases because you need to introduce a set of new interfaces and classes. Sometimes it's simpler just to change the service class so that it matches the rest of your code.

# Reference

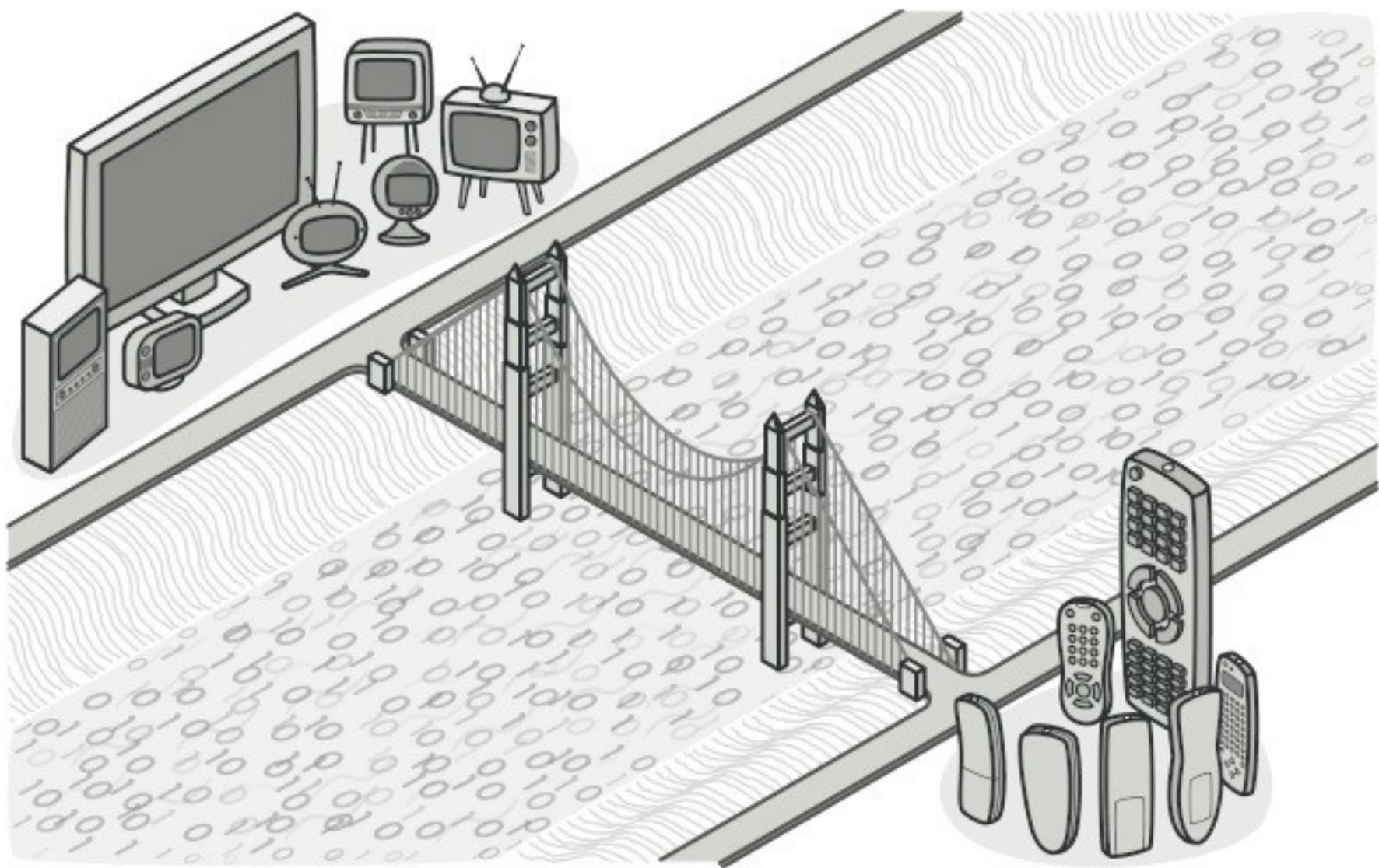
- <https://refactoring.guru/design-patterns/adapter>

# Bridge

Lecture 20

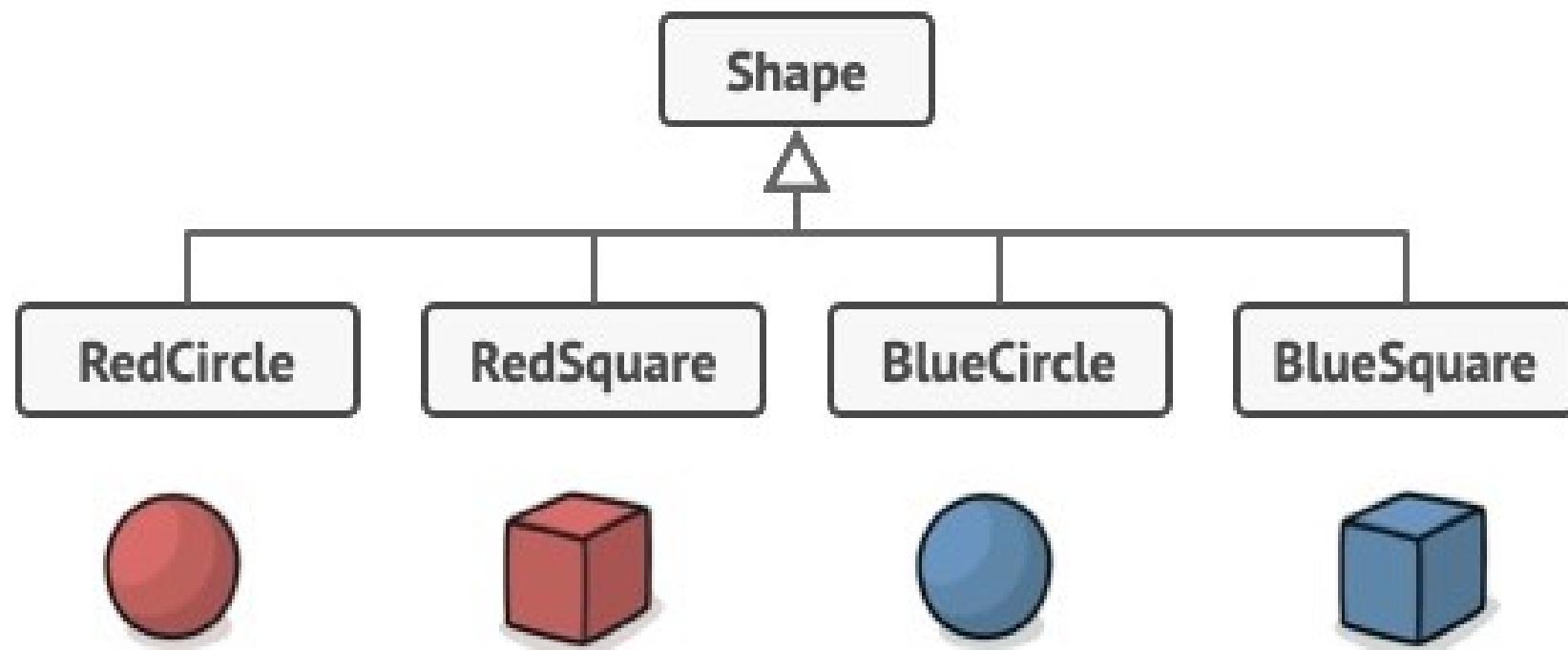
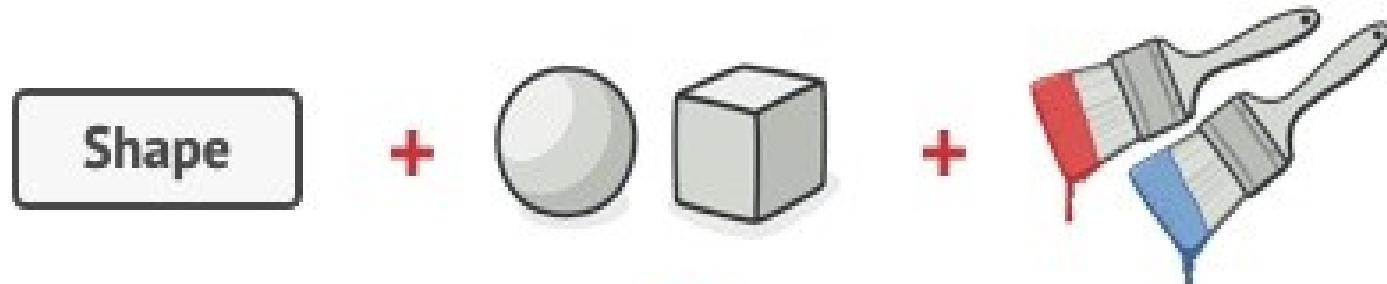
# Intent

- **Bridge** is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.



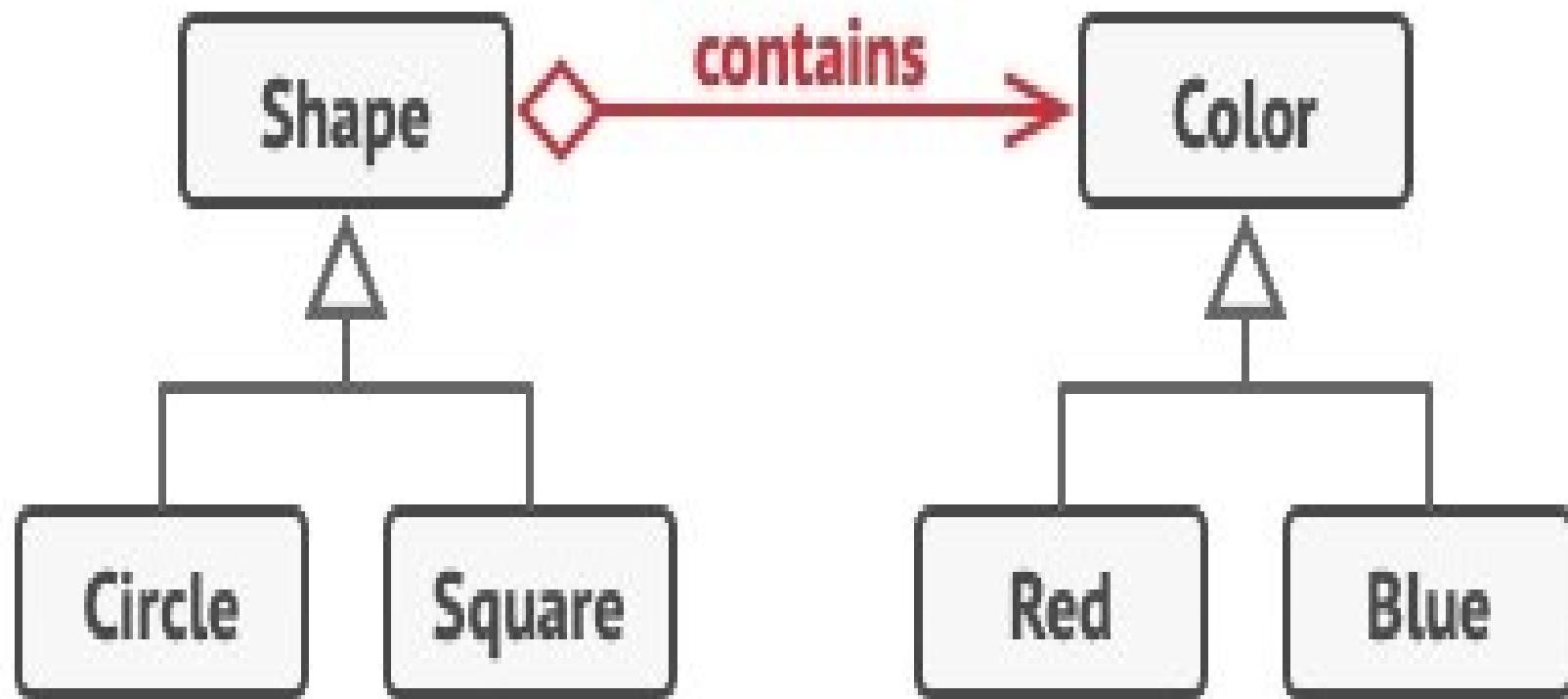
# Problem

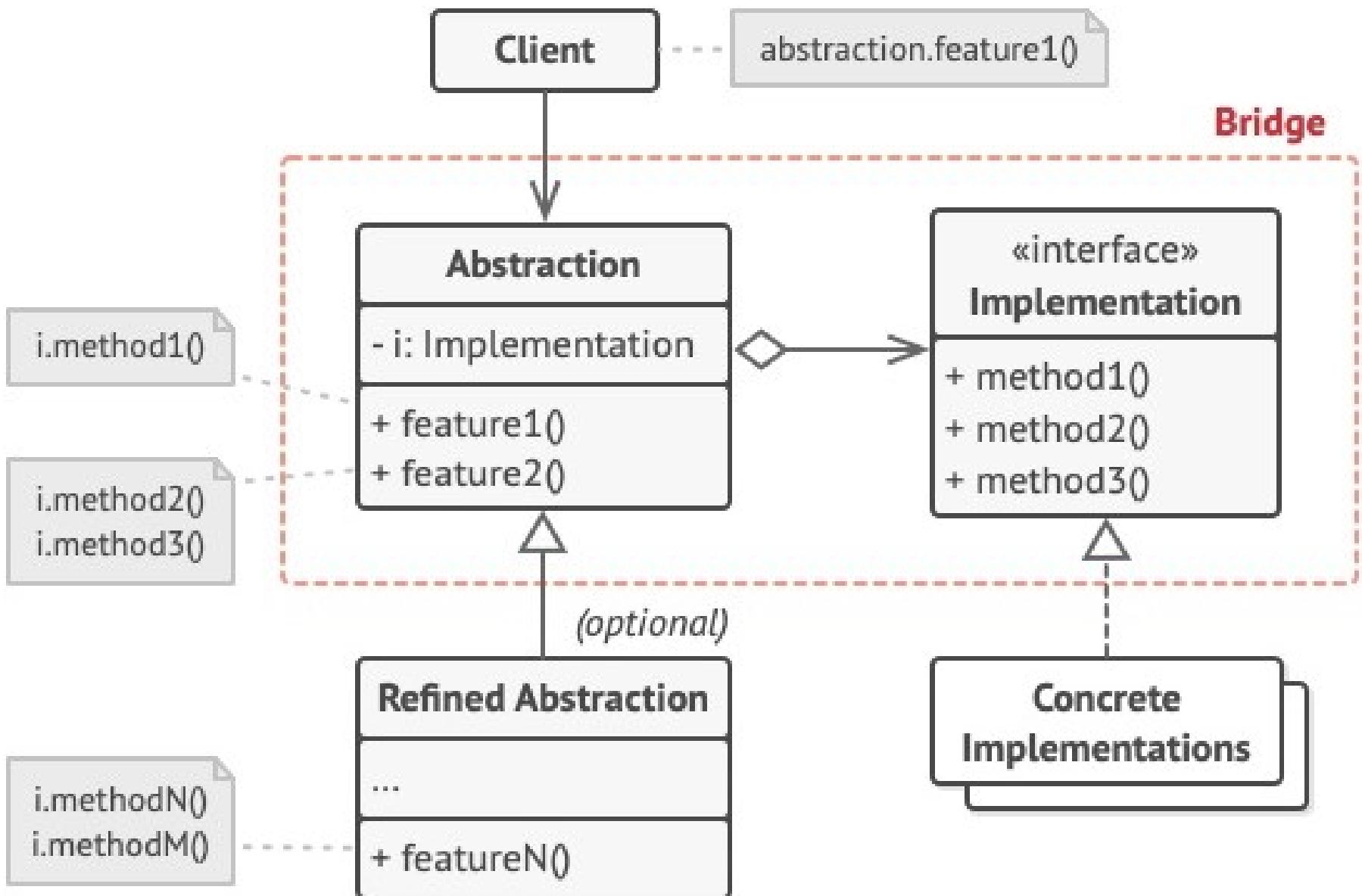
- Say you have a geometric Shape class with a pair of subclasses: Circle and Square.
- You want to extend this class hierarchy to incorporate colors, so you plan to create Red and Blue shape subclasses.
- However, since you already have two subclasses, you'll need to create four class combinations
- Adding new shape types and colors to the hierarchy will grow it exponentially.



# Solution

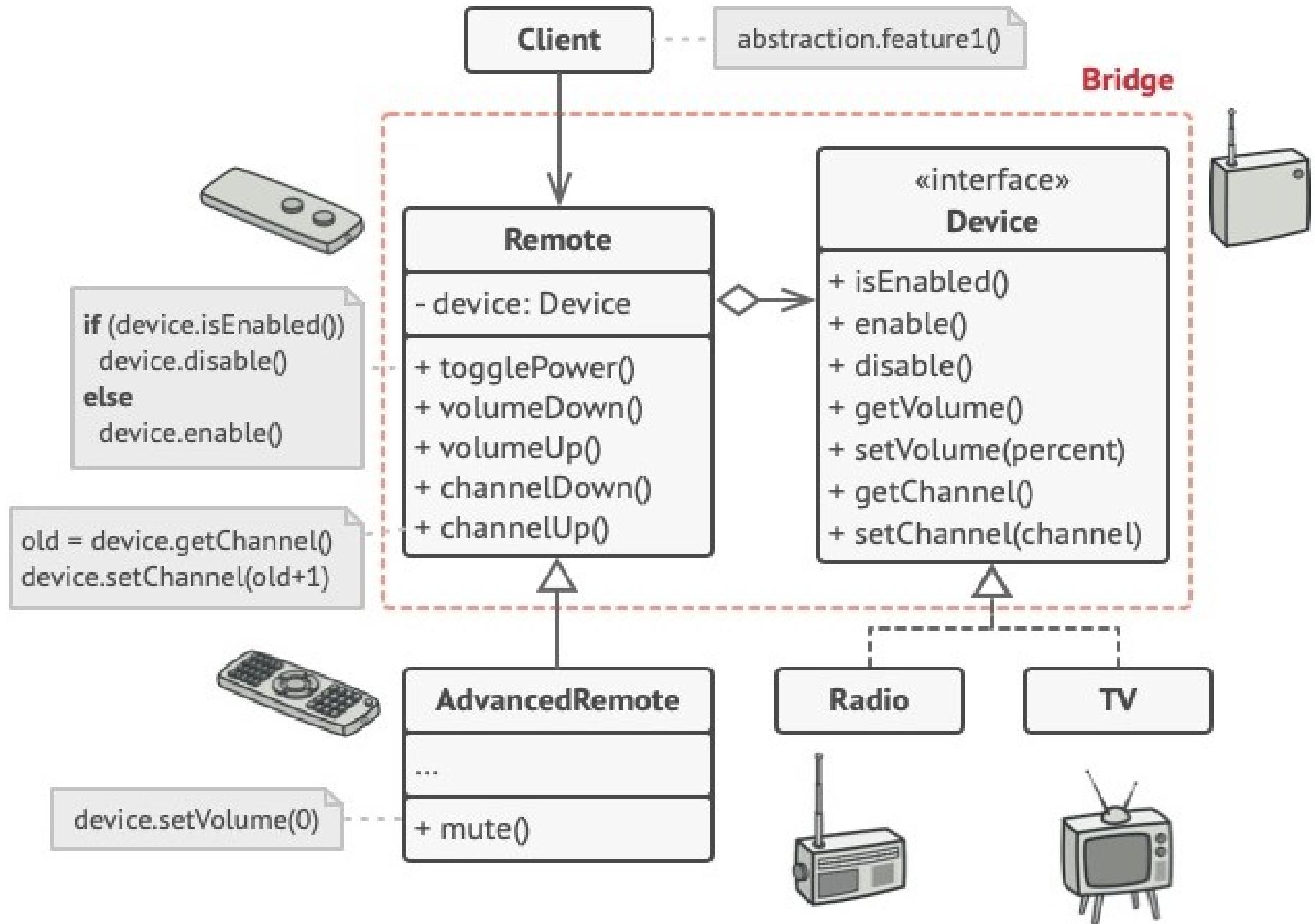
- This problem occurs because we're trying to extend the shape classes in two independent dimensions: by form and by color. That's a very common issue with class inheritance.
- The Bridge pattern attempts to solve this problem by switching from inheritance to the object composition.
- What this means is that you extract one of the dimensions into a separate class hierarchy, so that the original classes will reference an object of the new hierarchy, instead of having all of its state and behaviors within one class.





# Example

- This example illustrates how the **Bridge** pattern can help divide the monolithic code of an app that manages devices and their remote controls. The Device classes act as the implementation, whereas the Remotes act as the abstraction.



```
class RemoteControl is
    protected field device: Device
    constructor RemoteControl(device: Device) is
        this.device = device
    method togglePower() is
        if (device.isEnabled()) then
            device.disable()
        else
            device.enable()
    method volumeDown() is
        device.setVolume(device.getVolume() - 10)
    method volumeUp() is
        device.setVolume(device.getVolume() + 10)
    method channelDown() is
        device.setChannel(device.getChannel() - 1)
    method channelUp() is
        device.setChannel(device.getChannel() + 1)

// You can extend classes from the abstraction hierarchy
// independently from device classes.
class AdvancedRemoteControl extends RemoteControl is
    method mute() is
        device.setVolume(0)
```

```
// All devices follow the same interface.  
class Tv implements Device is  
    // ...  
  
class Radio implements Device is  
    // ...  
  
// Somewhere in client code.  
tv = new Tv()  
remote = new RemoteControl(tv)  
remote.togglePower()  
  
radio = new Radio()  
remote = new AdvancedRemoteControl(radio)
```

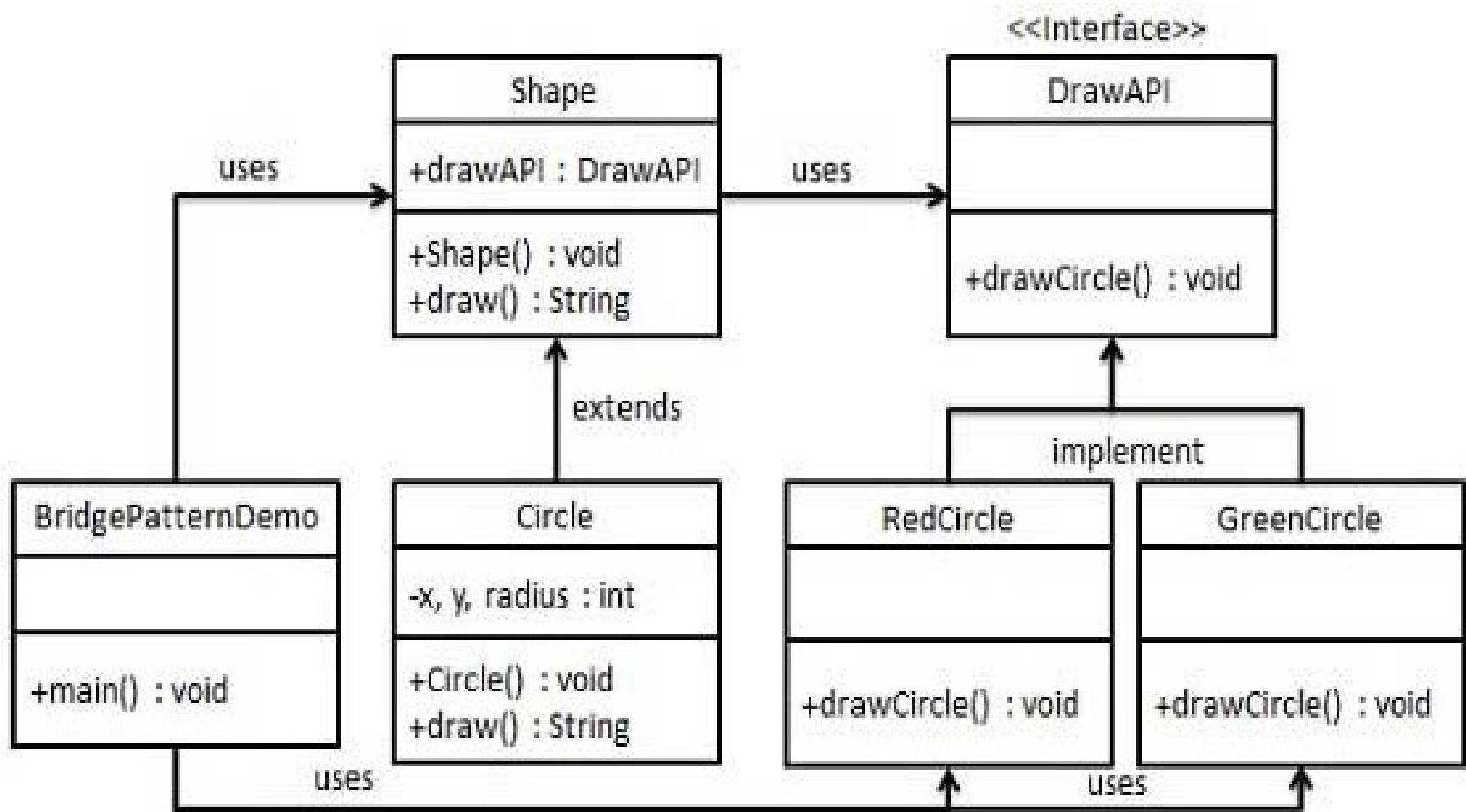
# Advantages

- You can create platform-independent classes and apps.
- The client code works with high-level abstractions. It isn't exposed to the platform details.
- *Open/Closed Principle*. You can introduce new abstractions and implementations independently from each other.
- *Single Responsibility Principle*. You can focus on high-level logic in the abstraction and on platform details in the implementation.

# Disadvantage

- You might make the code more complicated by applying the pattern to a highly cohesive class.

# Example



## Step 1

Create bridge implementer interface.

*DrawAPI.java*

```
public interface DrawAPI {  
    public void drawCircle(int radius, int x, int y);  
}
```

## Step 2

Create concrete bridge implementer classes implementing the *DrawAPI* interface.

### *RedCircle.java*

```
public class RedCircle implements DrawAPI {
    @Override
    public void drawCircle(int radius, int x, int y) {
        System.out.println("Drawing Circle[ color: red, radius: " + radius + ", x:
    }
}
```

### *GreenCircle.java*

```
public class GreenCircle implements DrawAPI {
    @Override
    public void drawCircle(int radius, int x, int y) {
        System.out.println("Drawing Circle[ color: green, radius: " + radius + ", x:
    }
}
```

## Step 3

Create an abstract class *Shape* using the *DrawAPI* interface.

*Shape.java*

```
public abstract class Shape {  
    protected DrawAPI drawAPI;  
  
    protected Shape(DrawAPI drawAPI){  
        this.drawAPI = drawAPI;  
    }  
    public abstract void draw();  
}
```

Class A

## Step 4

Create concrete class implementing the *Shape* interface.

*Circle.java*

```
public class Circle extends Shape {  
    private int x, y, radius;  
  
    public Circle(int x, int y, int radius, DrawAPI drawAPI) {  
        super(drawAPI);  
        this.x = x;  
        this.y = y;  
        this.radius = radius;  
    }  
  
    public void draw() {  
        drawAPI.drawCircle(radius,x,y);  
    }  
}
```

## Step 5

Use the *Shape* and *DrawAPI* classes to draw different colored circles.

*BridgePatternDemo.java*

```
public class BridgePatternDemo {  
    public static void main(String[] args) {  
        Shape redCircle = new Circle(100,100, 10, new RedCircle());  
        Shape greenCircle = new Circle(100,100, 10, new GreenCircle());  
  
        redCircle.draw();  
        greenCircle.draw();  
    }  
}
```

# Reference

- [https://www.tutorialspoint.com/design\\_pattern/bridge\\_pattern.htm](https://www.tutorialspoint.com/design_pattern/bridge_pattern.htm)
- <https://refactoring.guru/design-patterns/bridge>

# **Architectural design**

Lecture 21

# Agenda

- understand why the architectural design of software is important;
- understand the decisions that have to be made about the system architecture during the architectural design process;
- some architectural styles covering the overall system organisation, modular decomposition and control;
- understand how reference architectures are used to communicate architectural concepts and to assess system architectures.

# Architectural Design

- Large systems are always decomposed into sub-systems that provide some related set of services.
- Design process of identifying these sub-systems and establishing a framework for sub-system control and communication is called architectural design.
- The output of this design process is a description of the software architecture.

# Why explicitly designing a software architecture ?

- *Stakeholder communication*
- *System analysis*
- *Large-scale reuse*

# *Stakeholder communication*

- The architecture is a high-level presentation of the system that may be used as a focus for discussion by a range of different stakeholders.

# *System analysis*

- Making the system architecture explicit at an early stage in the system development requires some analysis.
- Architectural design decisions have a profound effect on whether the system can meet critical requirements such as performance, reliability and maintainability.

## *Large-scale reuse*

- A system architecture model is a compact, manageable description of how a system is organised and how the components interoperate.
- The system architecture is often the same for systems with similar requirements and so can support large-scale software reuse.

# Helps consider key design aspects

- Software architecture can serve as a design plan that is used to negotiate system requirements and as a means of structuring discussions with clients, developers and managers.
- essential tool for complexity management. It hides details and allows the designers to focus on the key system abstractions.

# Architectural Decisions

- The particular style and structure chosen for an application may therefore depend on the non-functional system requirements:
  1. Performance
  2. Security
  3. Safety
  4. Availability
  5. Maintainability

# Performance

- If performance is a critical requirement, the architecture should be designed to localise critical operations within a small number of subsystems, with as little communication as possible between these sub-systems.
- This may mean using relatively large-grain rather than fine-grain components to reduce component communications.

# Security

- If security is a critical requirement, a layered structure for the architecture should be used, with the most critical assets protected in the innermost layers and with a high level of security validation applied to these layers.

# Safety

- If safety is a critical requirement, the architecture should be designed so that safety-related operations are all located in either a single sub-system or in a small number of sub-systems.
- This reduces the costs and problems of safety validation and makes it possible to provide related protection systems.

# Availability

- If availability is a critical requirement, the architecture should be designed to include redundant components and so that it is possible to replace and update components without stopping the system.

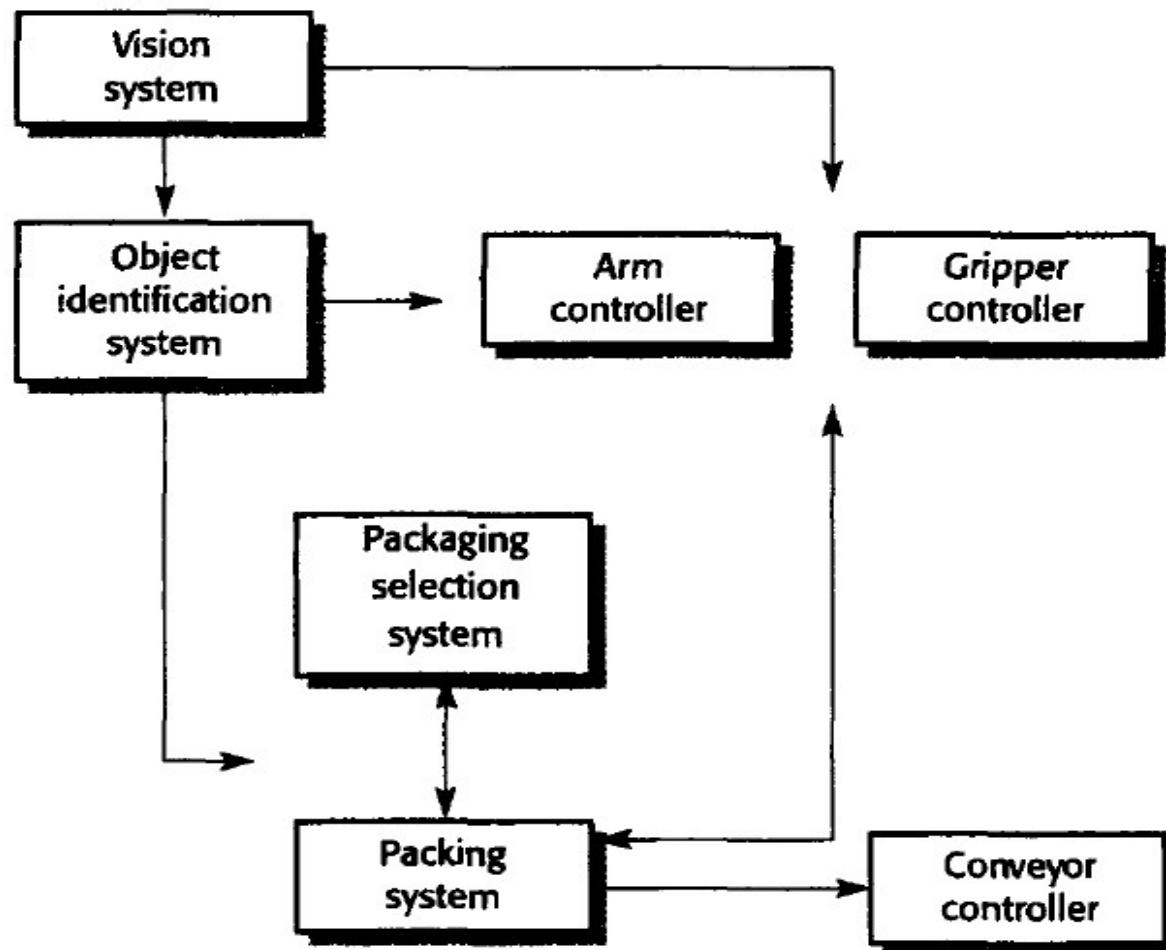
# Maintainability

- If maintainability is a critical requirement, the system architecture should be designed using fine-grain, self-contained components that may readily be changed.
- Producers of data should be separated from consumers and shared data structures should be avoided.

# Conflict of interest

- Potential conflict between some of these architectures.
  - For example, using large-grain components improves performance, and using fine-grain components improves maintainability. If both of these are important system requirements, then some compromise solution must be found

# Architecture for a packing robot system



# Architectural design decisions

- Is there a generic application architecture that can act as a template for the system that is being designed?
- How will the system be distributed across a number of processors?
- What architectural style or styles are appropriate for the system?
- What will be the fundamental approach used to structure the system?

## Contd..

- How will the structural units in the system be decomposed into modules?
- What strategy will be used to control the operation of the units in the system?
- How will the architectural design be evaluated?
- How should the architecture of the system be documented?

# Design Document

- The product of the architectural design process is an architectural design document.
  - A *static structural model that shows the subsystems or components that are to be developed as separate units.*
  - A *dynamic process model that shows how the system is organised into processes at run-time.* This may be different from the static model.

# Design Document

- An *interface model that defines the services offered by each sub-system through its public interface.*
- *Relationship models that shows relationships, such as data flow, between the sub-systems.*
- *A distribution model that shows how sub-systems may be distributed across computers.*

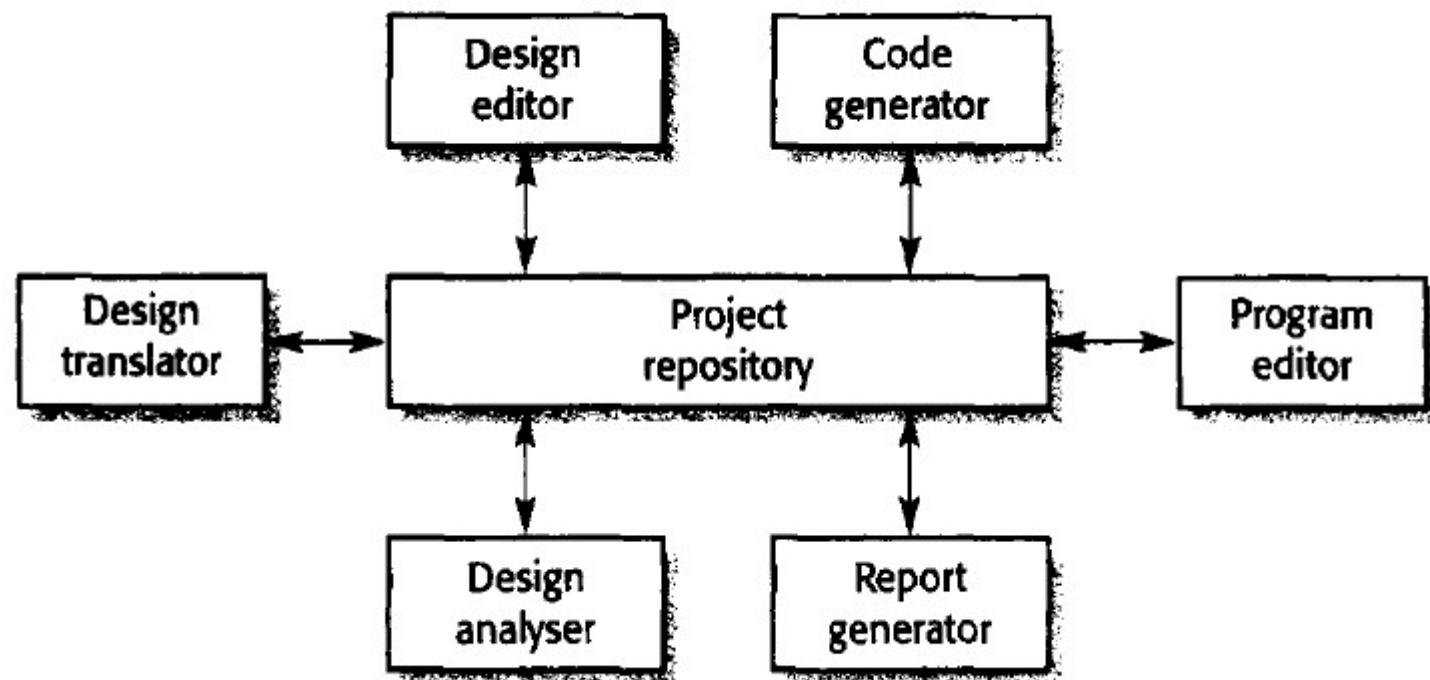
# Architectural Styles

- Repository Model
- Client Server model
- Layered Model

# The repository model

1. All shared data is held in a central database that can be accessed by all sub-systems. A system model based on a shared database is sometimes called a *repository model*.
2. Each sub-system maintains its own database. Data is interchanged with other sub-systems by passing messages to them.

# Architecture CASE Toolset



# Advantages /Disadvantages of a shared repository

- It is an efficient way to share large amounts of data. There is no need to transmit data explicitly from one sub-system to another.
- However, sub-systems must agree on the repository data model. Inevitably, this is a compromise between the specific needs of each tool. Performance may be adversely affected by this compromise. It may be difficult or impossible to integrate new sub-systems if their data models do not fit the agreed schema.

# Advantages /Disadvantages of a shared repository

- Sub-systems that produce data need not be concerned with how that data is used by other sub-systems.
- However, evolution may be difficult as a large volume of information is generated according to an agreed data model. Translating this to a new model will certainly be expensive; it may be difficult or even impossible.

# Advantages /Disadvantages of a shared repository

- Activities such as backup, security, access control and recovery from error are centralized. They are the responsibility of the repository manager. Tools can focus on their principal function rather than be concerned with these issues.
- Different sub-systems may have different requirements for security, recovery and backup policies. The repository model forces the same policy on all sub-systems.

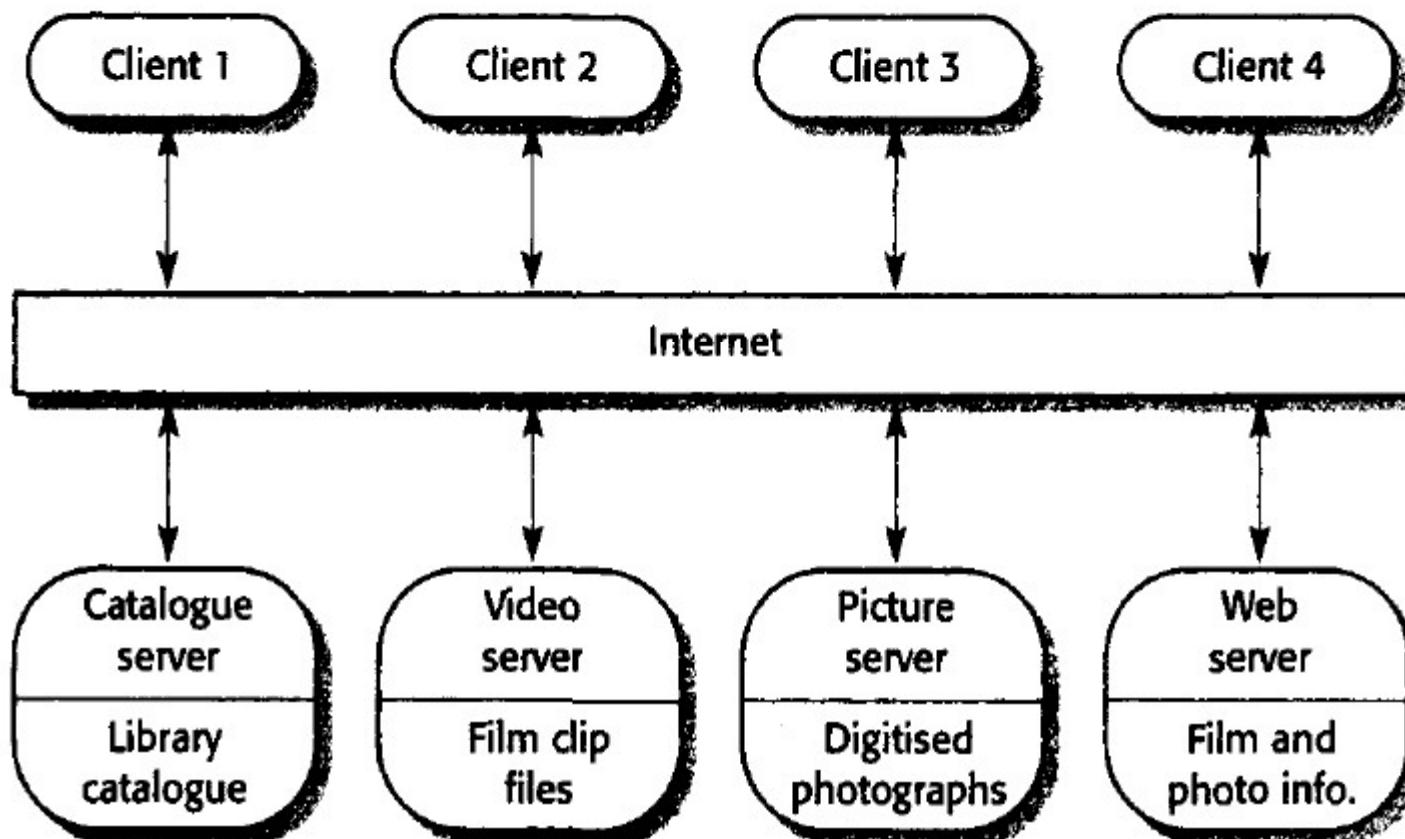
# Advantages /Disadvantages of a shared repository

- The model of sharing is visible through the repository schema. It is straightforward to integrate new tools given that they are compatible with the agreed data model.
- Difficult to distribute the repository over a number of machines. Although it is possible to distribute a logically centralized repository, there may be problems with data redundancy and inconsistency.

# The client-server model

- The client-server architectural model is a system model where the system is organized as 2 set of services and associated servers and clients that access and use the services.

# Film and picture library system



# Advantages

- client-server model is that it is a distributed architecture.
- Effective use can be made of networked systems with many distributed processors.
- It is easy to add a new server and integrate it with the rest of the system or to upgrade servers transparently without affecting other parts of the system.

# The layered model

- The layered model of an architecture (sometimes called an abstract machine model) organizes a system into layers, each of which provide a set of services
- An example of a layered model is the OSI reference model of network protocols

# Layered model of a version management system

Configuration management system layer

Object management system layer

Database system layer

Operating system layer

# Example

- The configuration management system manages versions of objects and provides general configuration management facilities
- it uses an object management system that provides information storage and management services for configuration items or objects.
- This system is built on top of a database system to provide basic data storage and services such as transaction management, rollback and recovery,
- and access control.

# Advantages

- The layered approach supports the incremental development of systems. As a layer is developed, some of the services provided by that layer may be made available to users.
- This architecture is also changeable and portable.
- As layered systems localize machine dependencies in inner layers, this makes it easier to provide multi-platform implementations

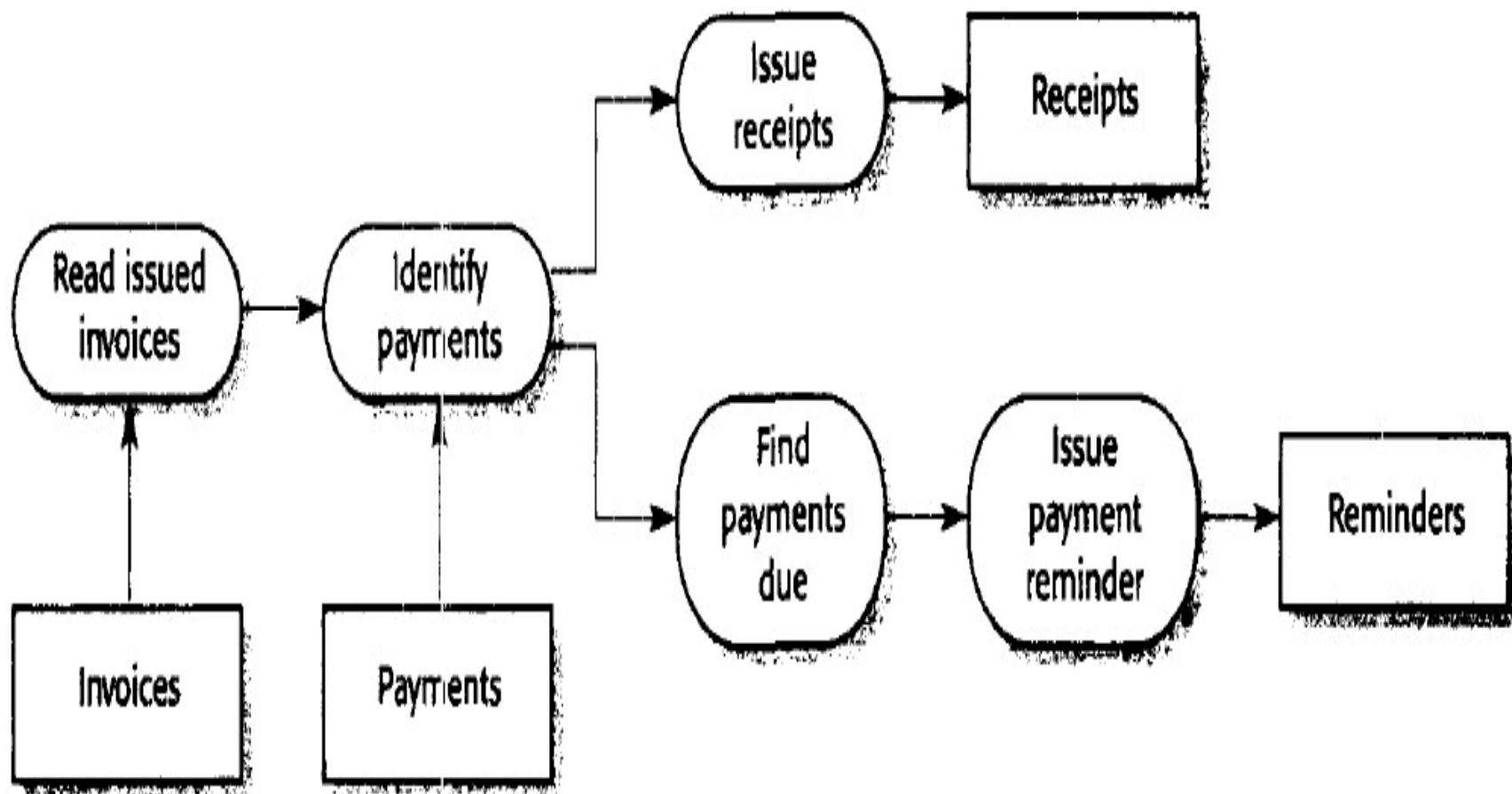
# Disadvantages

- A disadvantage of the layered approach is that structuring systems in this way can be difficult. Inner layers may provide basic facilities, such as file management, that are required at all levels.
- Performance can also be a problem because of the multiple levels of command interpretation that are sometimes required

# Modular decomposition styles

- *Object-oriented decomposition*
- *FUnction-oriented pipelining*

# pipeline model of an invoice processing



# *Advts of Object-oriented decomposition*

- Objects are loosely coupled, the implementation of objects can be modified without affecting other objects.
- Objects are often representations of real-world entities so the structure of the system is readily understandable.
- Object-oriented programming languages have been developed that provide direct implementations of architectural components.

# Advts Function-oriented pipelining

- It supports the reuse of transformations.
- It is intuitive in that many people think of their work in terms of input and output processing.
- Evolving the system by adding new transformations is usually straightforward.
- It is simple to implement either as a concurrent or a sequential system

# Control styles

- Centralised control One sub-system has overall responsibility for control and starts and stops other sub-systems. It may also devolve control to another subsystem but will expect to have this control responsibility returned to it.
- Event-based control Rather than control information being embedded in a subsystem, each sub-system can respond to externally generated events. These events might come from other sub-systems or from the environment of the system.

# Reference

- Ian Sommerville 2000 Software Engineering.  
Chapter 11 8<sup>th</sup> edition

# User interface design

Lecture 22

# Agenda

- understand a number of user interface design principles;
- several interaction styles and appropriateness;
- understand when to use graphical and textual presentation of information;
- know what is involved in the principal activities in the use interface design process;
- understand usability attributes and have been introduced to different approaches to interface evaluation.

# Design issues

- How should the user interact with the computer system?
- How should information from the computer system be presented to the user?

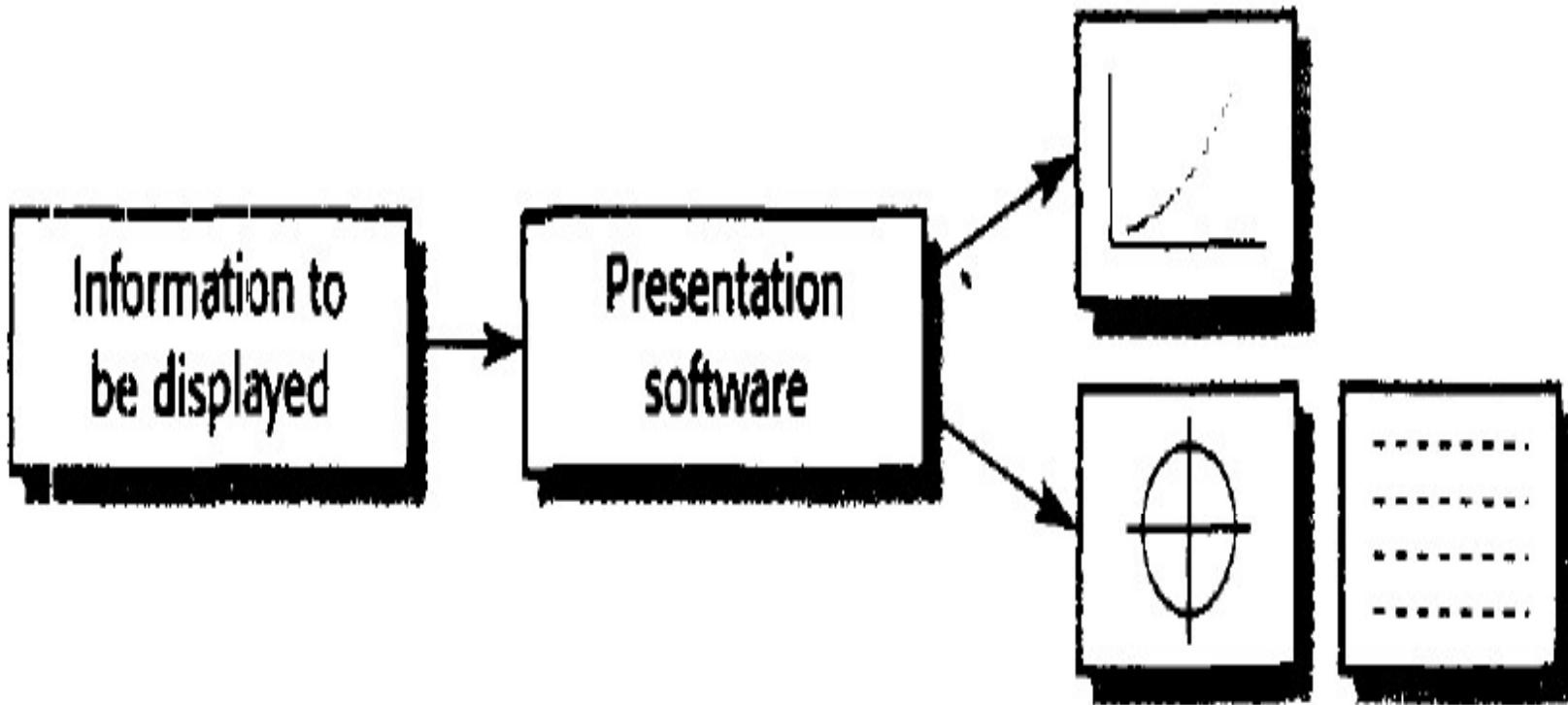
# User interaction styles

- *Direct manipulation*
- *Menu selection*
- *Form fill in*
- *Command language*
- *Natural language*

Interaction style	Main advantages	Main disadvantages	Application examples
Direct manipulation	Fast and intuitive interaction Easy to learn	May be hard to implement Only suitable where there is a visual metaphor for tasks and objects	Video games CAD systems
Menu selection	Avoids user error Little typing required	Slow for experienced users Can become complex if many menu options	Most general-purpose systems
Form fill-in Easy to learn Checkable	Simple data entry	Takes up a lot of screen space Causes problems where user options do not match the form fields	Stock control Personal loan processing
Command language	Powerful and flexible	Hard to learn Poor error management	Operating systems Command and control systems
Natural language	Accessible to casual users Easily extended	Requires more typing Natural language understanding systems are unreliable	Information retrieval systems

# Information presentation

- All interactive systems have to provide some way of presenting information to users. The information presentation may simply be a direct representation of the input information (e.g., text in a word processor) or it may present the information graphically.
- A good design guideline has to keep the software required for information presentation separate from the information itself.
- Separating the presentation system from the data allows us to change the representation on the user screen without having to change the underlying computational system.



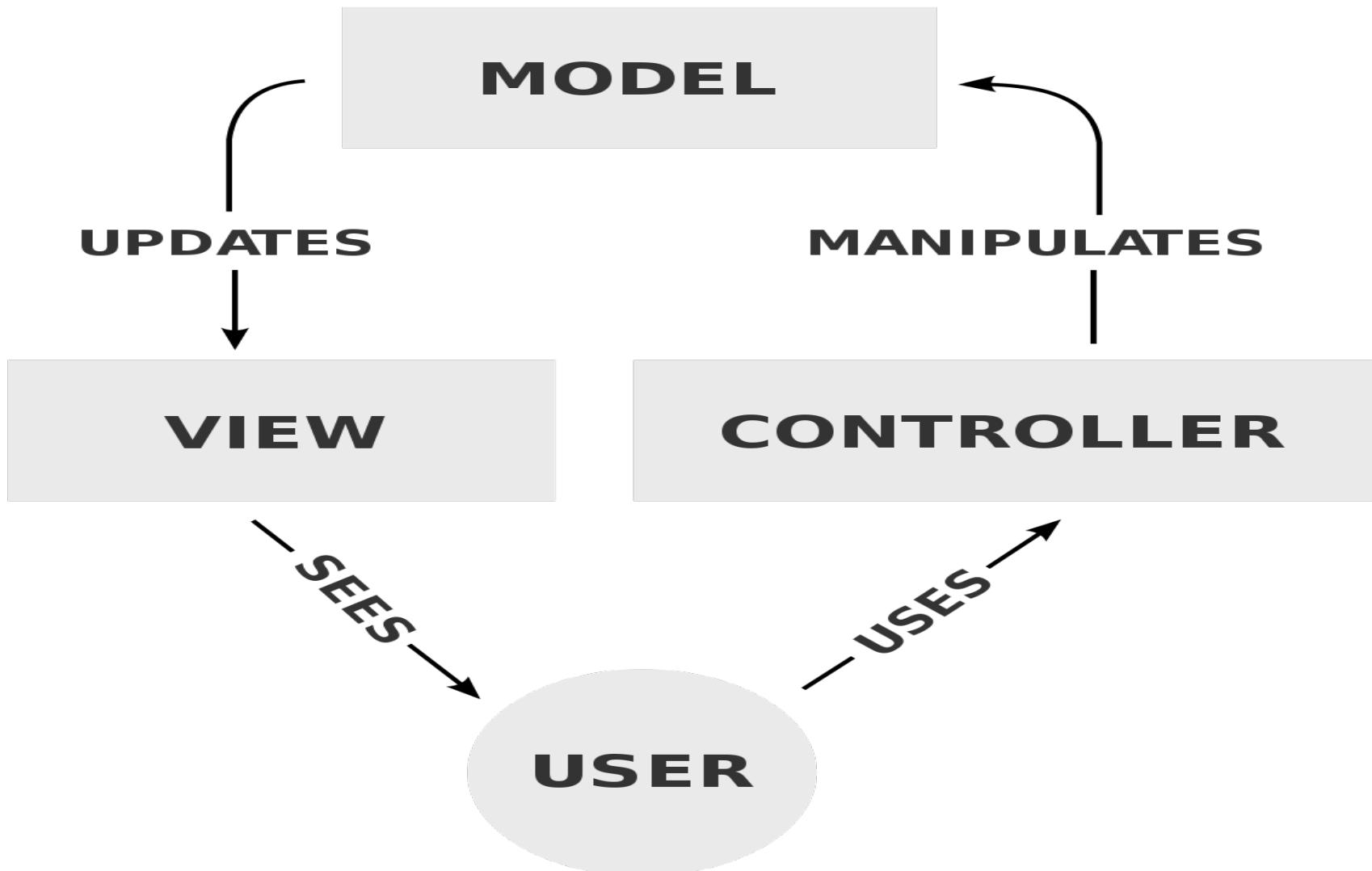
Display

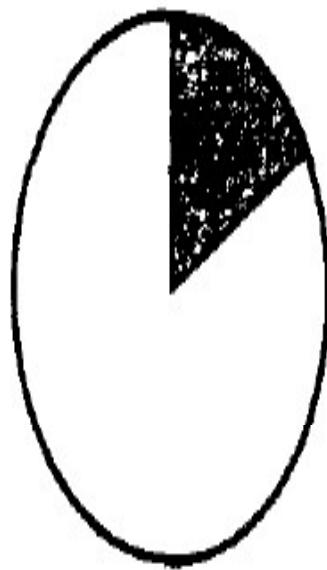
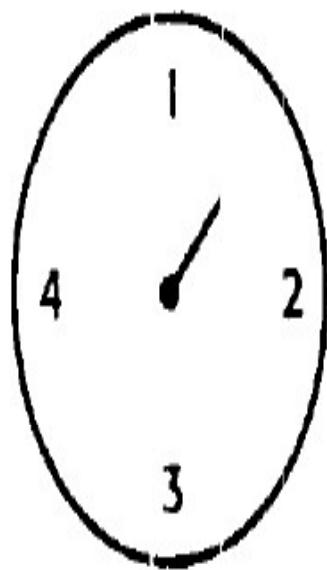
# MVC Approach

- The MVC approach (Figure 16.6), first made widely available in Smalltalk (Goldberg and Robson, 1983), is an effective way to support multiple presentations of data.
- Users can interact with each presentation in a style that is appropriate to the presentation. The data to be displayed is encapsulated in a model object.
- Each model object may have a number of separate view objects associated with it where each view is a different display representation of the model.

# MVC Approach

- Each view has an associated controller object that handles user input and device interaction. Therefore, a model that represents numeric data may have a view that represents the data as a histogram and a view that presents the data as a table.
- The model may be edited by changing the values in the table or by lengthening or shortening the bars in the histogram.





Dial with needle

Pie chart

Thermometer

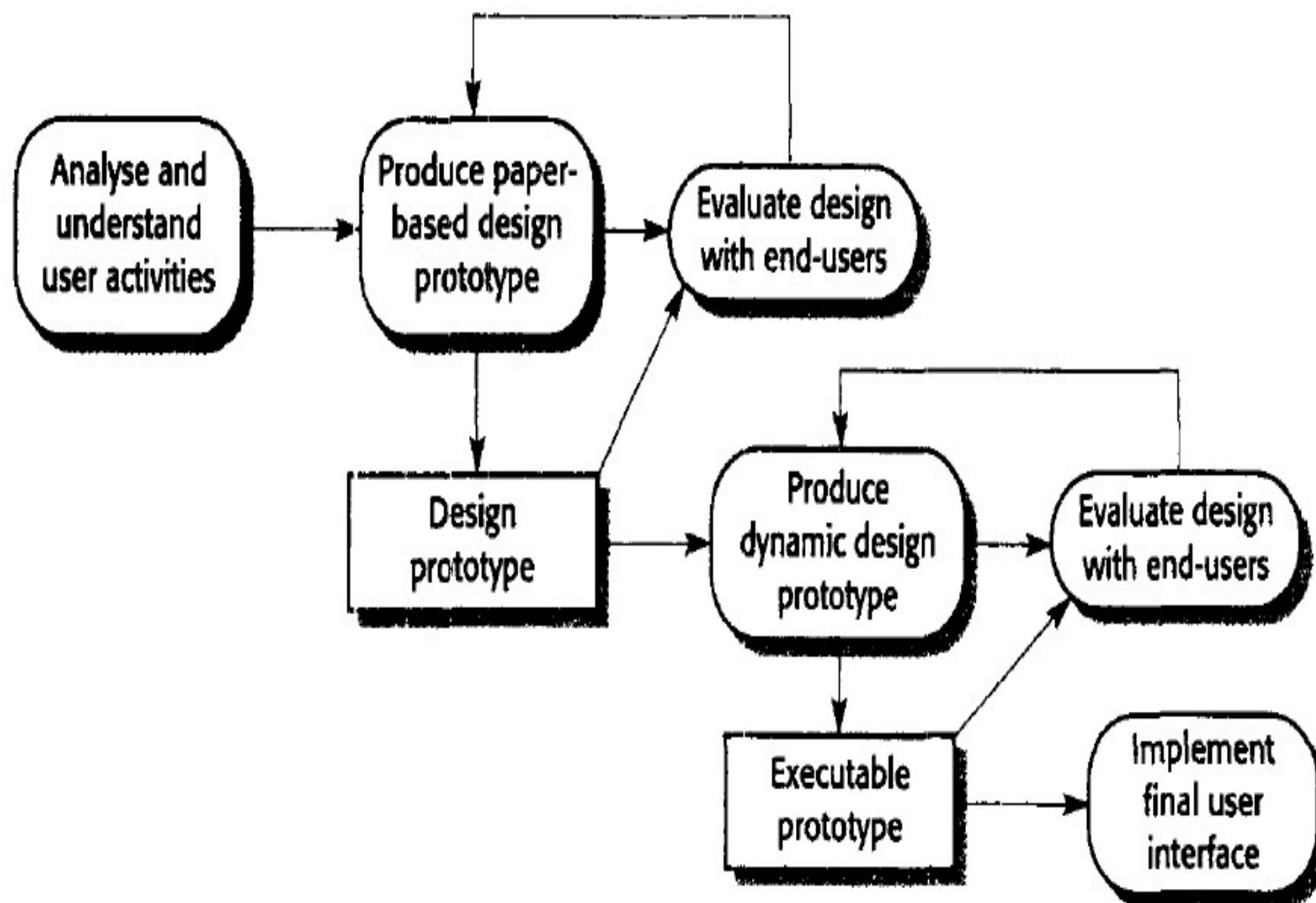
Horizontal bar

Factor	Description
Context	<b>Wherever possible, the messages generated by the system should reflect the current user context. As far as is possible, the system should be aware of what the user is doing and should generate messages that are relevant to their current activity</b>
Experiene	<b>As users become familiar with a system they become irritated by long, 'meaningful' messages. However, beginners find it difficult to understand short, terse statements of a problem. You should provide both types of message and allow the user to control message conciseness.</b>
Skill level	<b>Messages should be tailored to the users' skills as well as their experience. Messages for the different classes of users may be expressed in different ways depending on the terminology that is familiar to the reader.</b>
Style	<b>Messages should be positive rather than negative. They should use the active rather than the passive mode of address. They should never be insulting or try to be funny.</b>
Culture	<b>Wherever possible, the designer of messages should be familiar with the culture of the country where the system is sold. There are distinct cultural differences between Europe, Asia and America. A suitable message for one culture might be unacceptable in another.</b>

# The UI design process

- User interface (UI) design is an iterative process where users interact with designers and interface prototypes to decide on the features, organization and the look and feel of the system user interface.
- Sometimes, the interface is separately prototyped in parallel with other software engineering activities.
- More commonly, especially where iterative development is used, the user interface design proceeds incrementally as the software is developed.
- In both cases, however, before you start programming, you should have developed and, ideally, tested some paper-based designs.

# Design Process



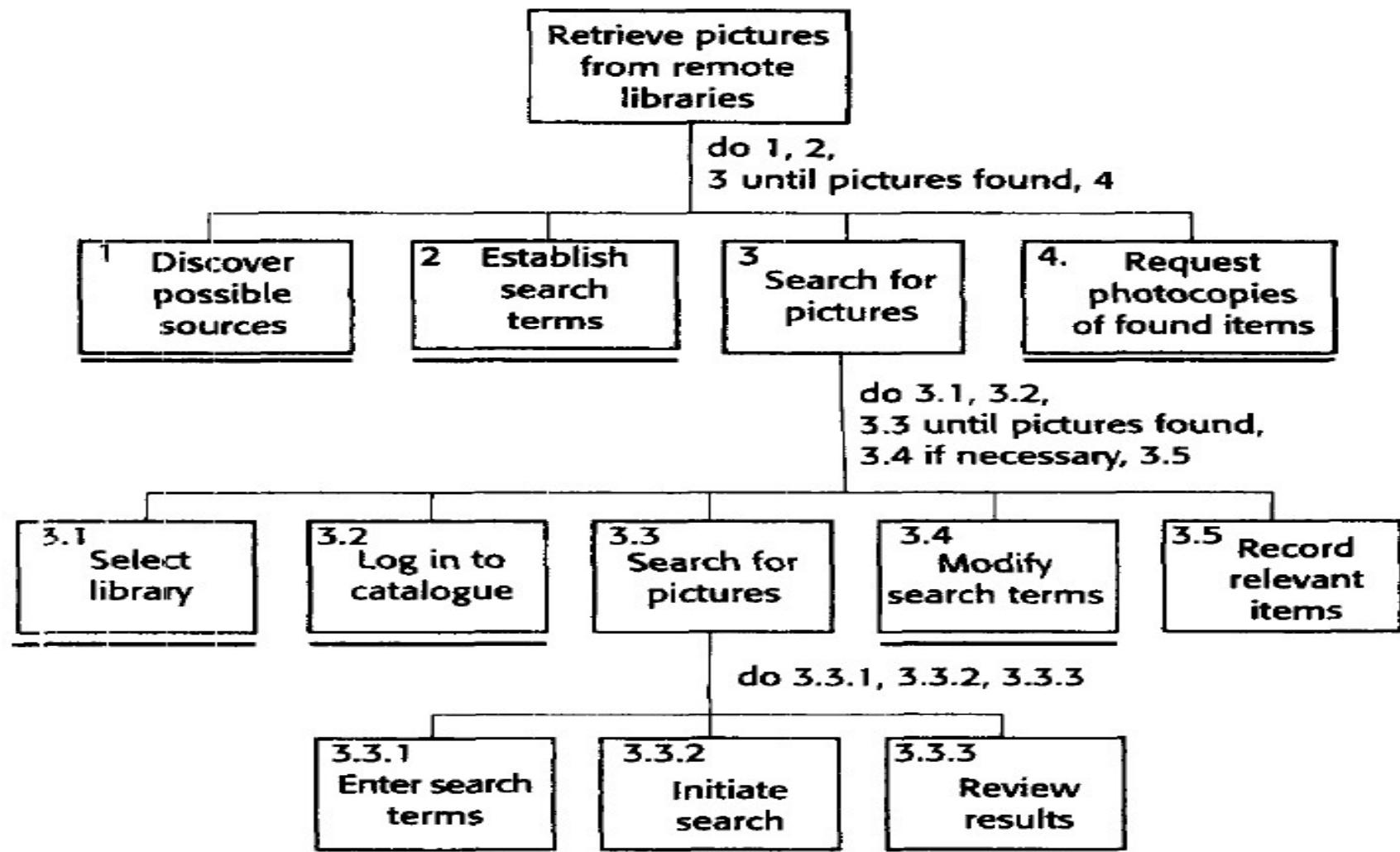
# User analysis

- If you don't understand what users want to do with a system, then you have no realistic prospect of designing an effective user interface.
- To develop this understanding, you may use techniques such as task analysis, ethnographic studies, user interviews and observations or, commonly, a mixture of all of these.

# User analysis problems

- Notations such as UML sequence charts may be able to describe user interactions and are ideal for communicating with software engineers.
- However, other users may think of these charts as too technical and will not try to understand them.
- It is very important to engage users in the design process, you therefore usually have to develop natural language scenarios to describe user activities.

# LIBSYS system Natural Language scenario



# User interface prototyping

- Evolutionary or exploratory prototyping with end-user involvement is the only practical way to design and develop graphical user interfaces for software systems.
- Involving the user in the design and development process is an essential aspect of user-centred design,a design philosophy for interactive systems.

# Prototyping process

- adopt a two-stage prototyping process:
  - develop paper prototypes-mock-ups of screen designs-and walk through these with end-users.
  - develop increasingly sophisticated automated prototypes, then make them available to users for testing and activity simulation.

# Approaches -user interface prototyping:

- There are three approaches that you can use for user interface prototyping:
  - *Script-driven approach*
  - *Visual programming languages*
  - *Internet based prototyping*

## *Script-driven approach*

- In this approach, you create screens with visual elements, such as buttons and menus, and associate a script with these elements.
- When the user interacts with these screens, the script is executed and the next screen is presented, showing them the results of their actions. There is no application logic involved.

# *Visual programming languages*

- Visual programming languages, such as Visual Basic, incorporate a powerful development environment, access to a range of reusable objects and a user-interface development system that allows interfaces to be created quickly, with components and scripts associated with interface objects.

## *Internet based prototyping*

- These solutions, based on web browsers and languages such as Java, offer a ready-made user interface
- add functionality by associatmg segments of Java programs with the information

# Interface evaluation

- Interface evaluation is the process of assessing the usability of an interface and checking that meets user requirements. Therefore, it should be part of the normal verification and validation process for software systems

# Usability attributes

Attribute	Description
Learnability	How long does it take a new user to become productive with the system?
Speed of operation	How well does the system response match the user's work practice?
Robustness	How tolerant is the system of user error?
Recoverability	How good is the system at recovering from user errors?
Adaptability	How closely is the system tied to a single model of work?

# user interface evaluation

- Questionnaires that collect information about what users thought of the interface;
- Observation of users at work with the system and 'thinking aloud' about how they are trying to use the system to accomplish some task;
- Video 'snapshots' of typical system use;
- The inclusion in the software of code which collects information about the most used facilities and the most common errors.

# Reference

- Chapter 16; Software Engineering
- Author Ian Sommerville

# Distributed software engineering

Lecture 23

Based on 17<sup>th</sup> Chapter Software  
Engineering Ian Sommerville 10<sup>th</sup>  
Edition

## Topics covered

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

# Distributed systems

- ✧ Virtually all large computer-based systems are now distributed systems.

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

# Distributed system characteristics

## ✧ Resource sharing

- Sharing of hardware and software resources.

## ✧ Openness

- Use of equipment and software from different vendors.

## ✧ Concurrency

- Concurrent processing to enhance performance.

## ✧ Scalability

- Increased throughput by adding new resources.

## ✧ Fault tolerance

- The ability to continue in operation after a fault has occurred.

# Distributed systems

# Distributed systems issues

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

# Design issues

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

# Transparency

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

# Openness

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

# Scalability

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

# Security

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

# Types of attack

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

# Quality of service

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

# Failure management

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

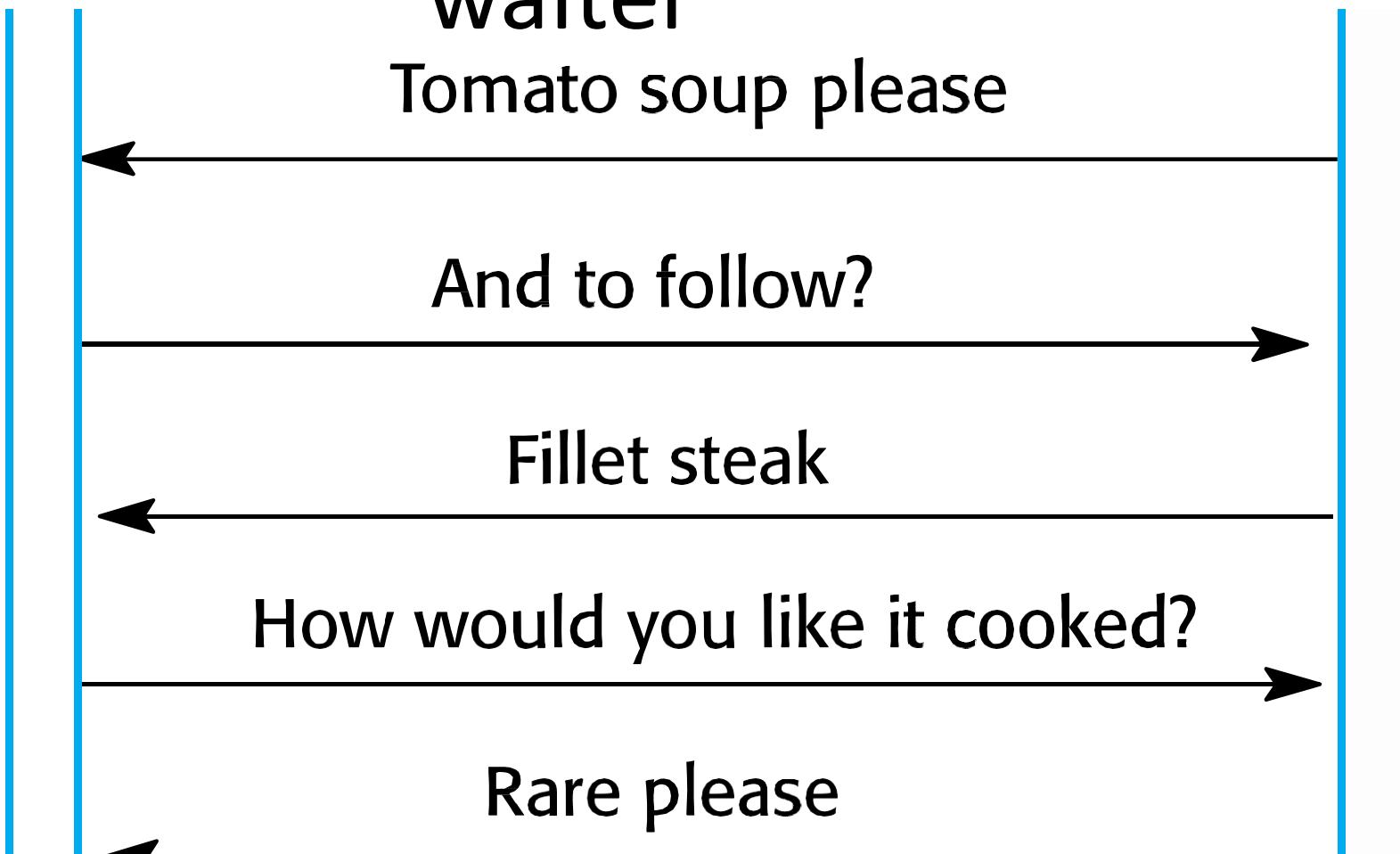
*“You know that you have a distributed system when the crash of a system that you’ve never heard of stops you getting any work done.”*

- ✧ Distributed systems should include mechanisms for discovering if a component of the system has failed, should continue to deliver as many services as possible in spite of that failure and, as far as possible, automatically recover from the failure.

# Models of interaction

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

# Procedural interaction between a diner and a waiter



# Message-based interaction between a waiter and the kitchen

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

# Remote procedure calls

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

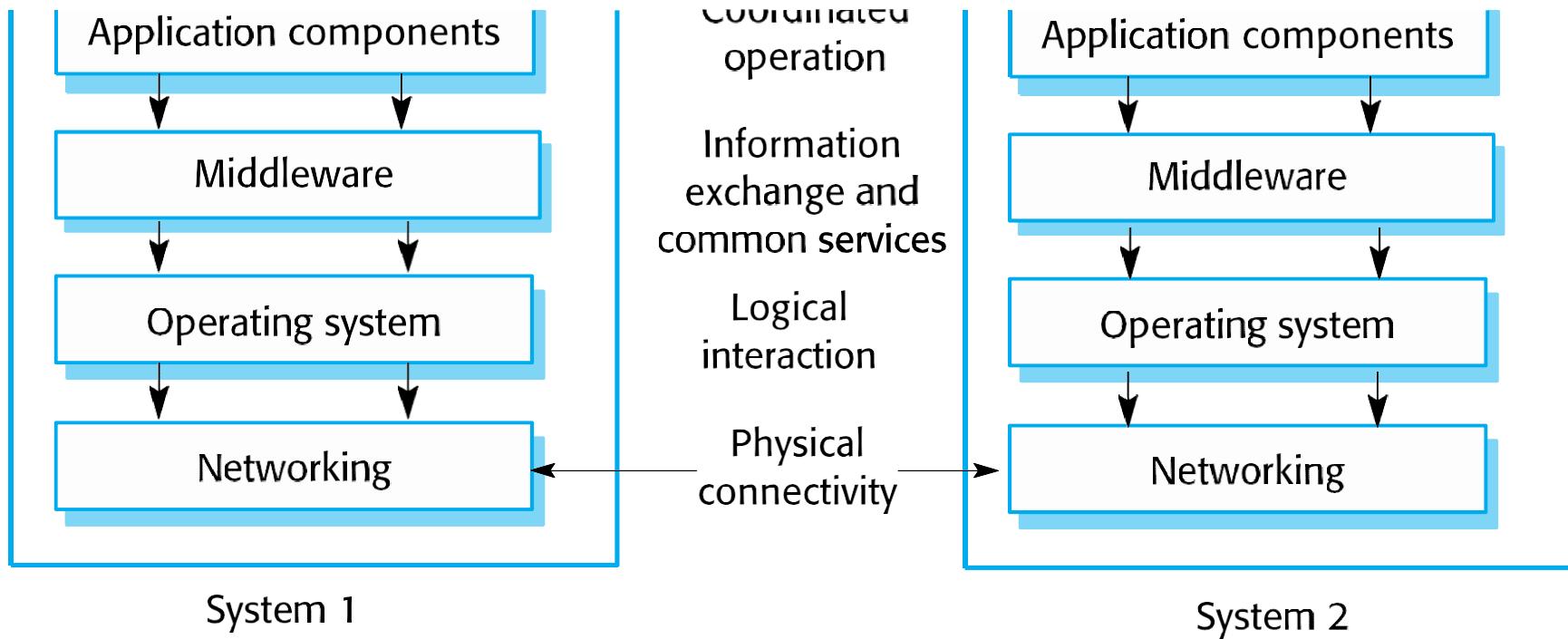
# Message passing

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

## Middleware

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

# Middleware in a distributed system



# Middleware support

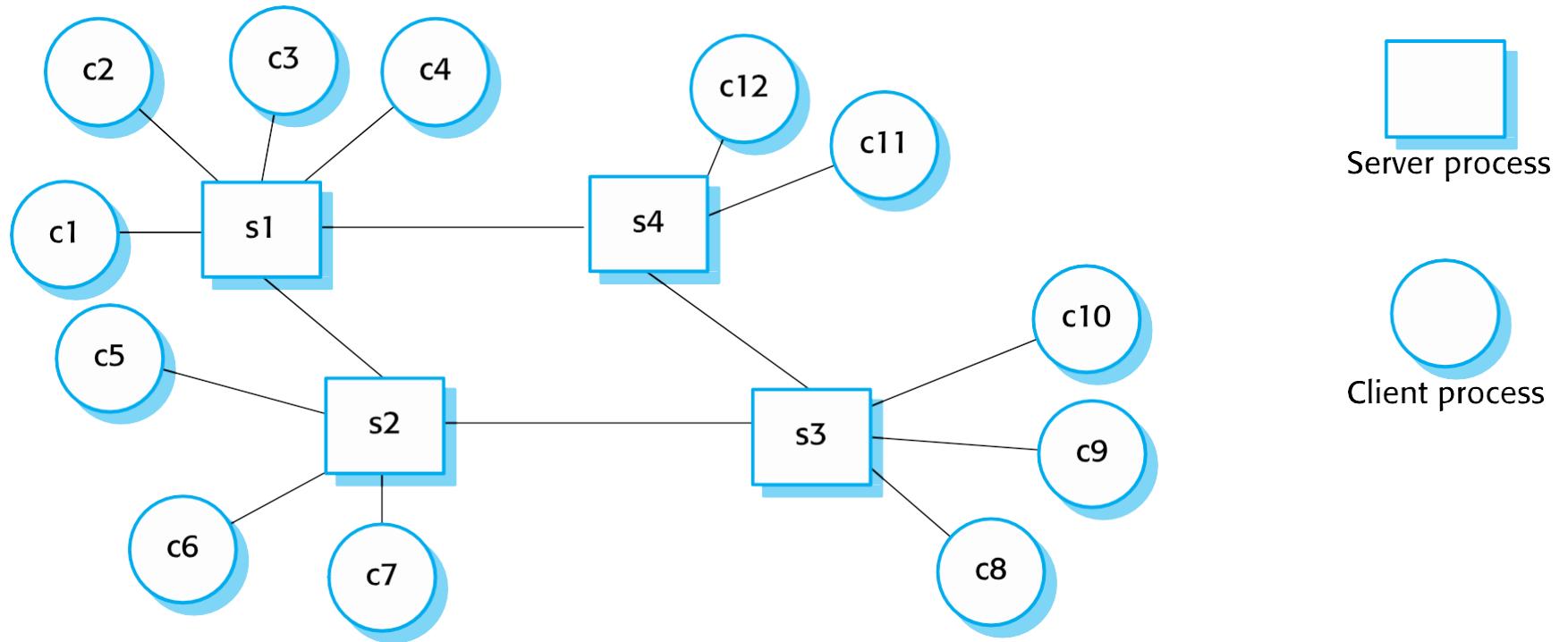
- ✧ Interaction support, where the middleware coordinates interactions between different components in the system
  - The middleware provides location transparency in that it isn't necessary for components to know the physical locations of other components.
- ✧ The provision of common services, where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

# Client-server computing

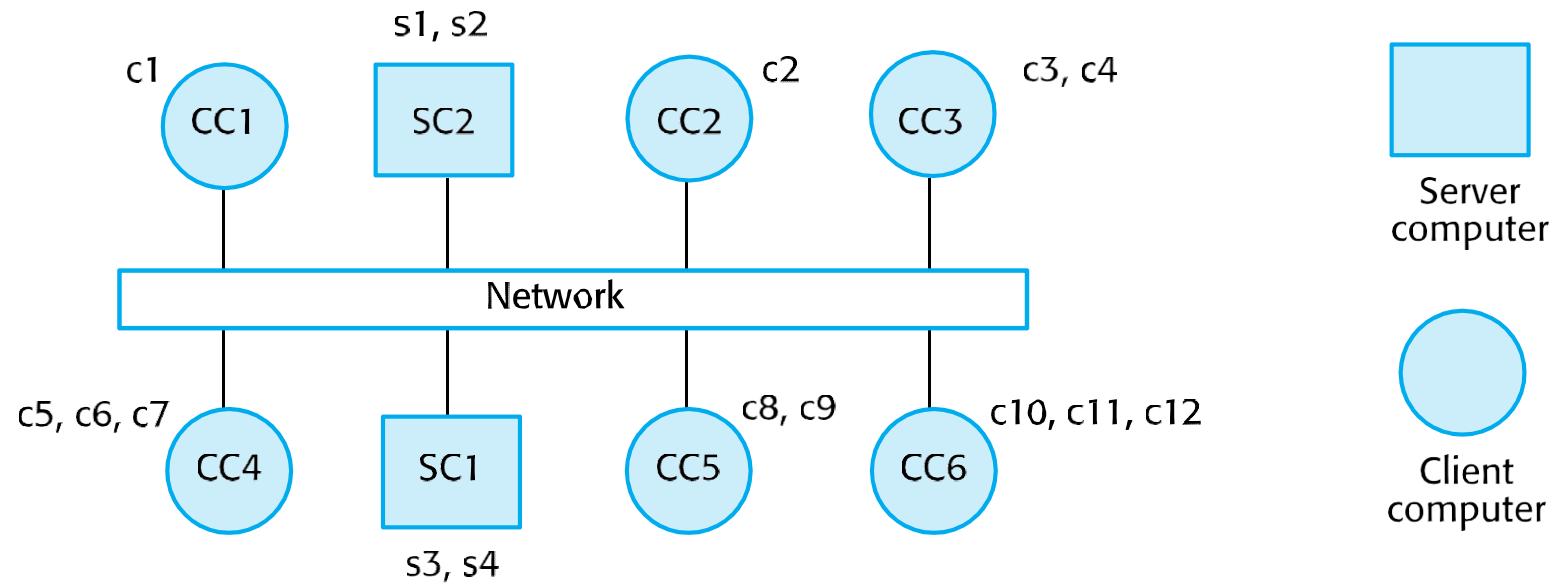
# Client-server computing

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

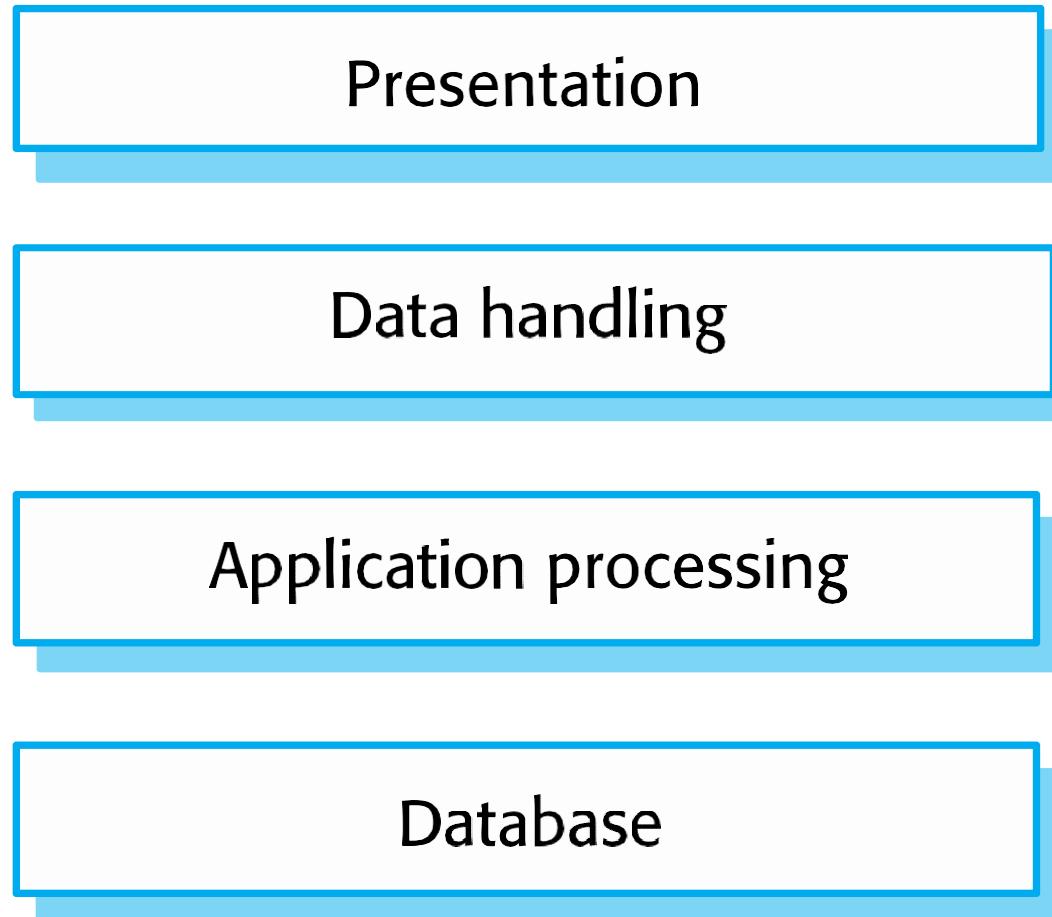
# Client–server interaction



# Mapping of clients and servers to networked computers



# Layered architectural model for client–server applications



# Layers in a client/server system

## ✧ *Presentation*

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

## ✧ *Data handling*

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

## ✧ Application processing layer

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

## ✧ Database

- Stores data and provides transaction management services, etc.

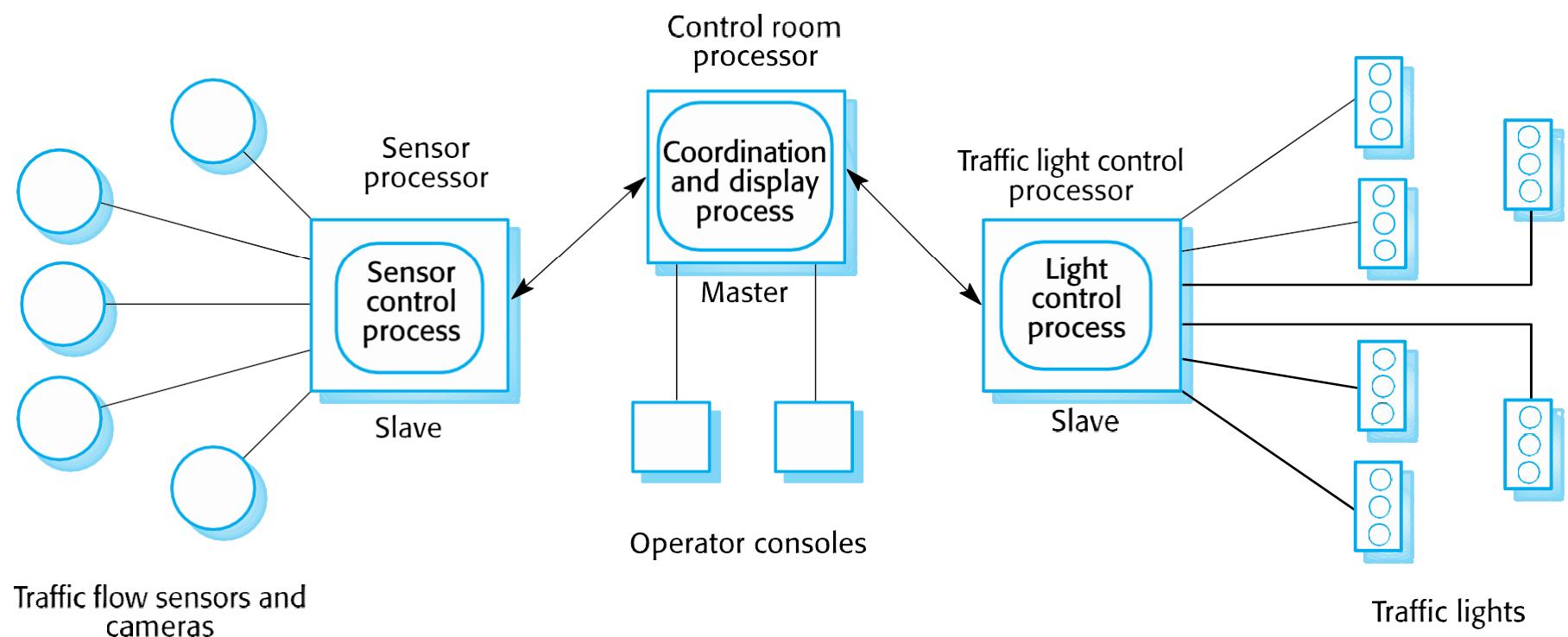
# Architectural patterns

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

## Master-slave architectures

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

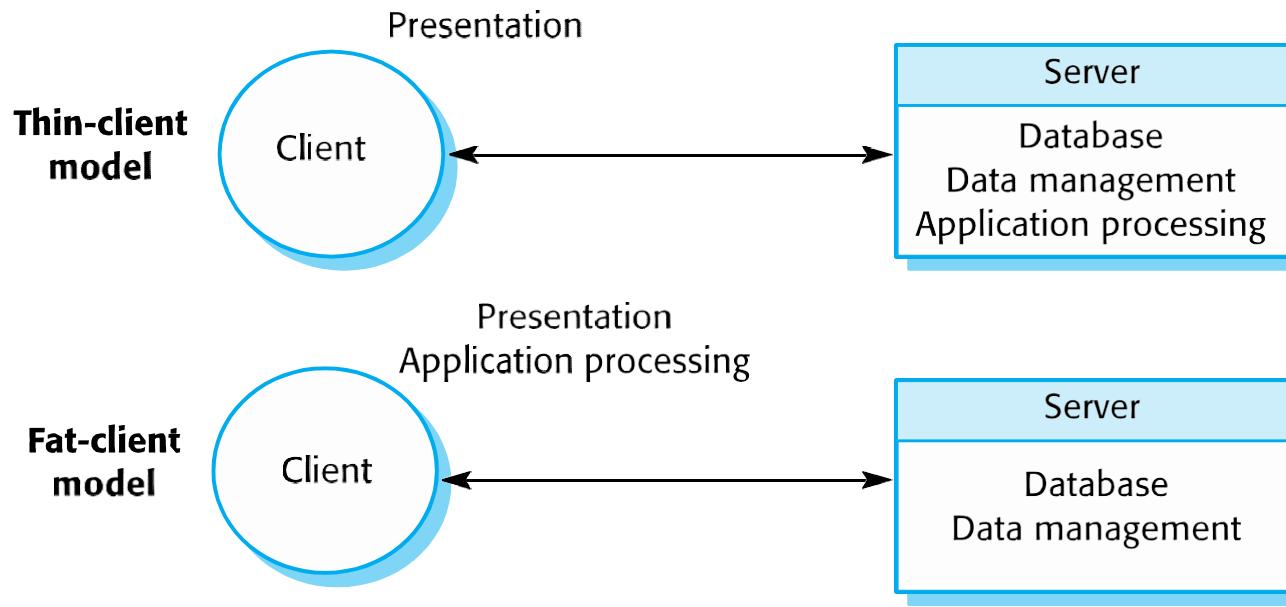
# A traffic management system with a master-slave architecture



# Two-tier client server architectures

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

# Thin- and fat-client architectural models



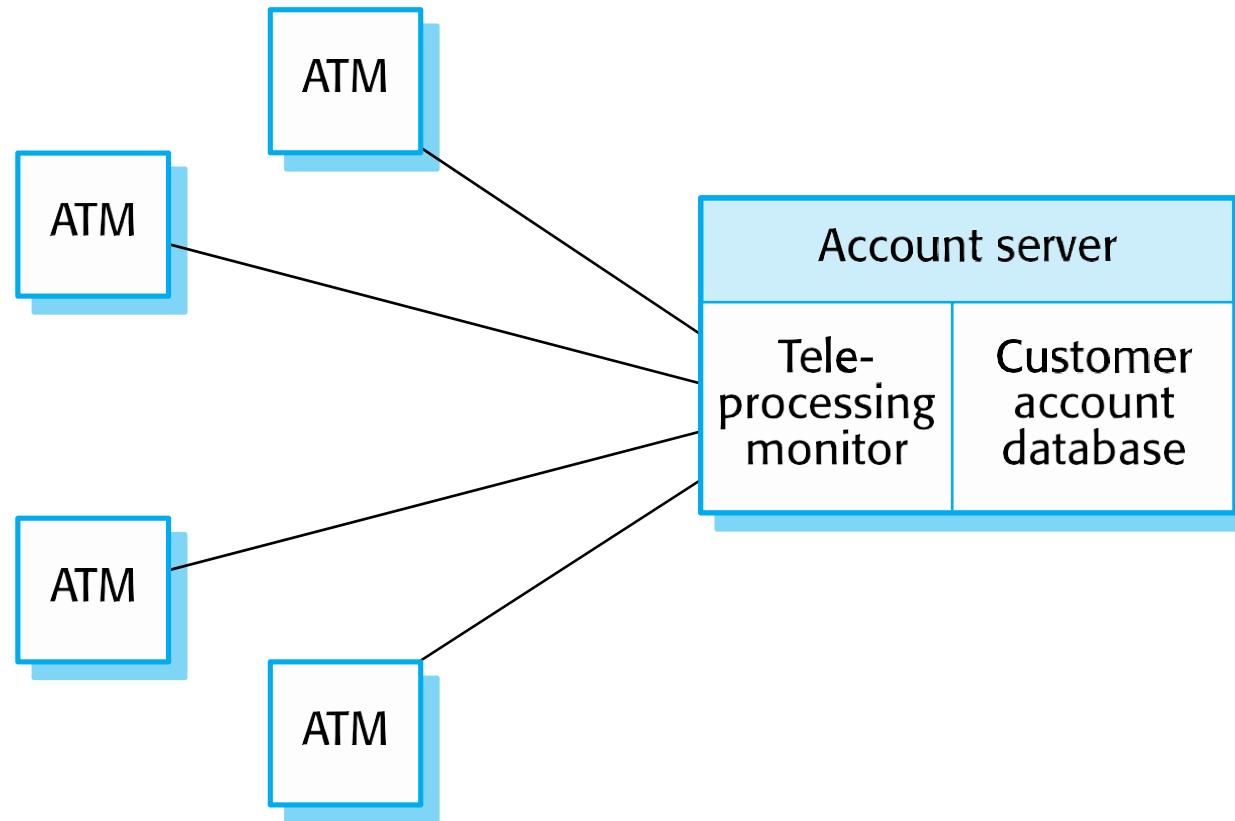
# Thin client model

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

# Fat client model

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

# A fat-client architecture for an ATM system



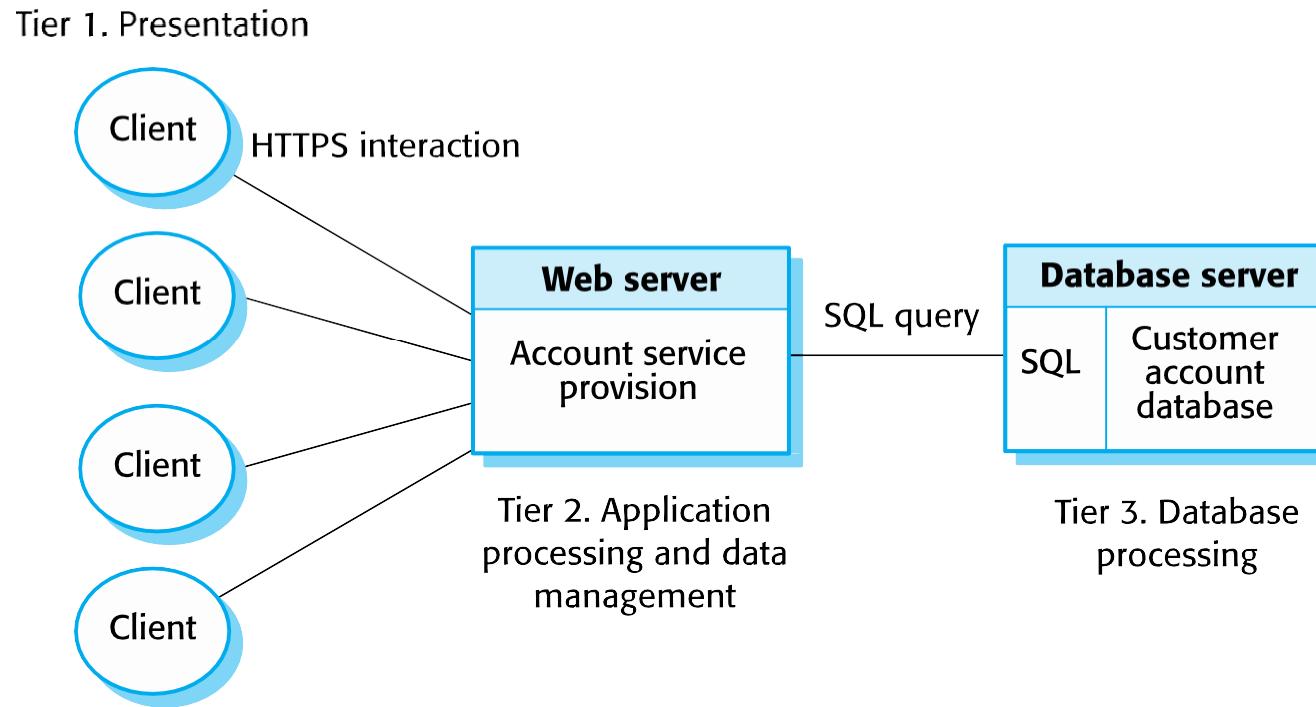
# Thin and fat clients

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

## **Multi-tier client-server architectures**

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

# Three-tier architecture for an Internet banking system



# Use of client–server architectural patterns

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

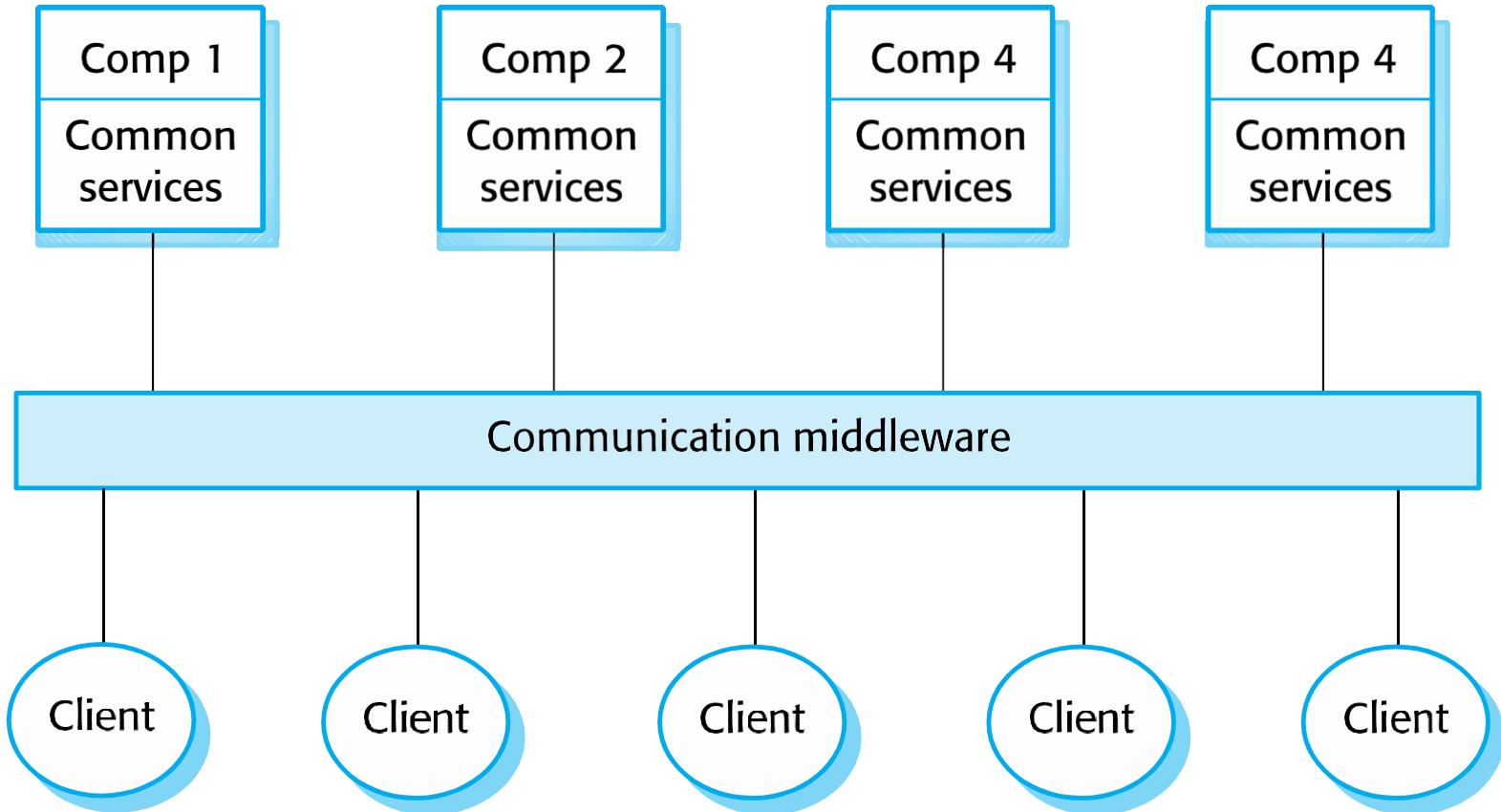
# Use of client–server architectural patterns

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

# Distributed component architectures

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

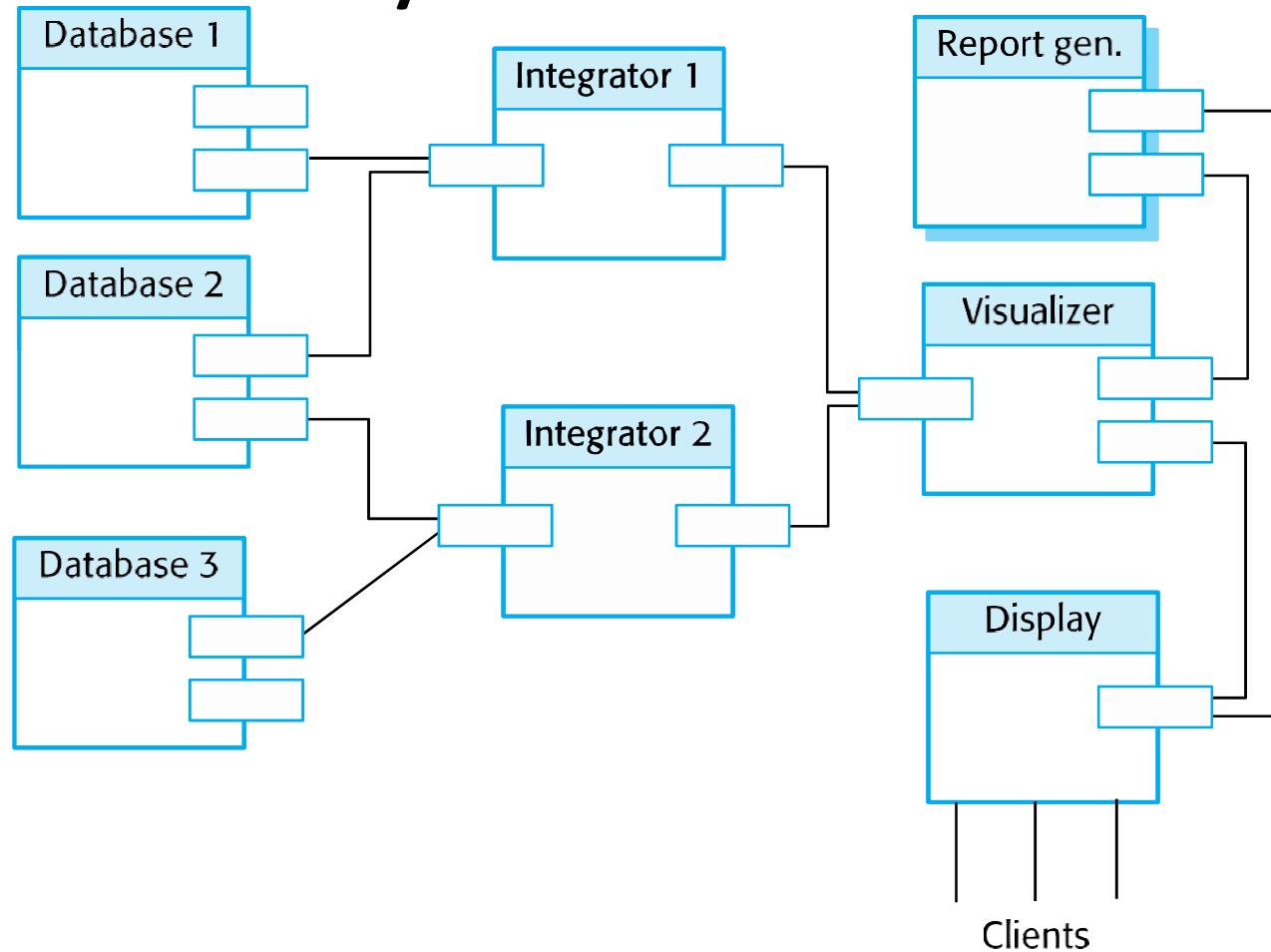
# A distributed component architecture



# Benefits of distributed component architecture

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

# A distributed component architecture for a data mining system



# Disadvantages of distributed component architecture

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

## Peer-to-peer architectures

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

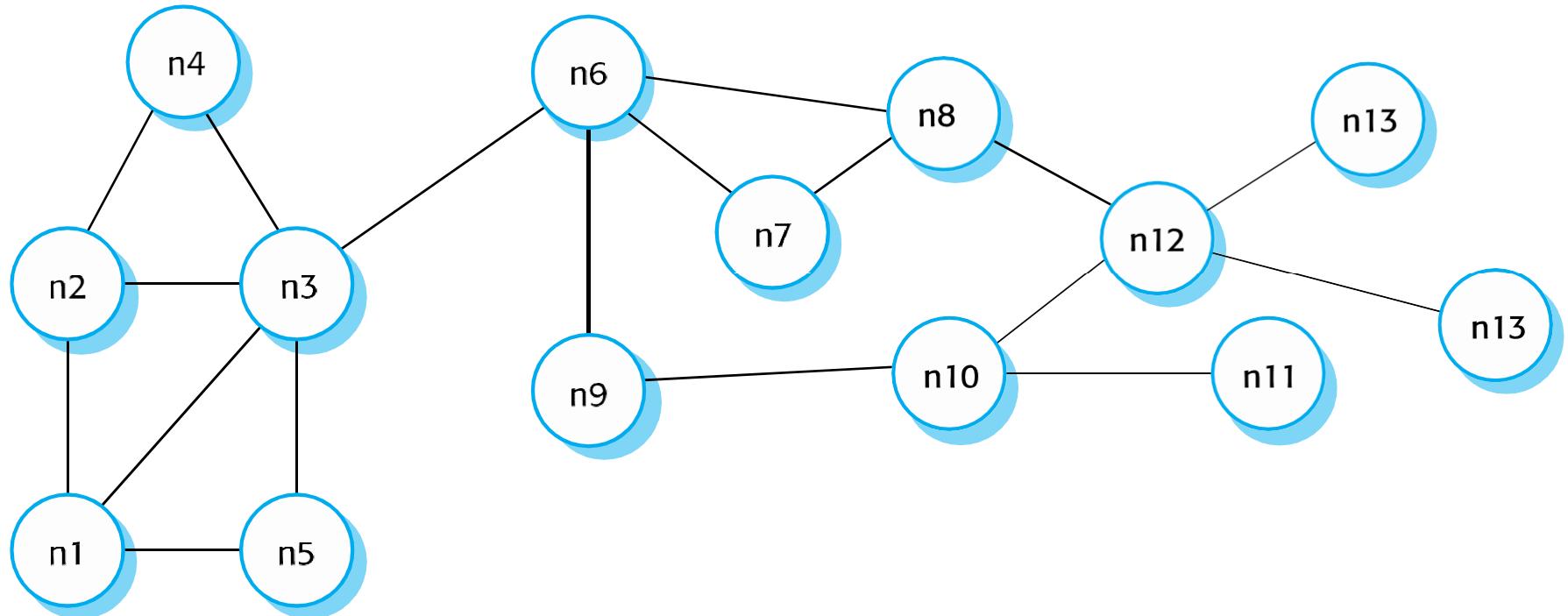
## Peer-to-peer systems

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

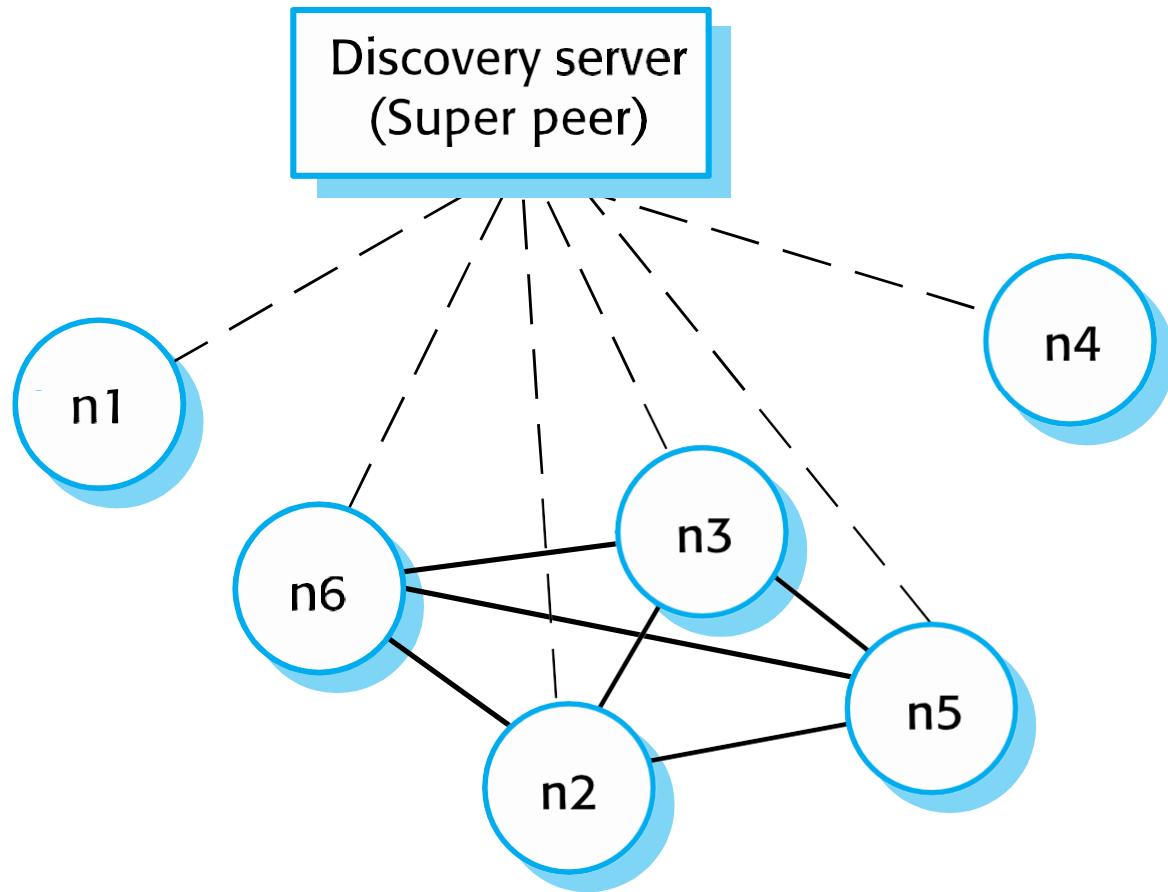
# P2p architectural models

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

# A decentralized p2p architecture



# A semicentralized p2p architecture



# Software as a service

## Use of p2p architecture

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

# Security issues in p2p system

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

# Software as a service

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

## Key elements of SaaS

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

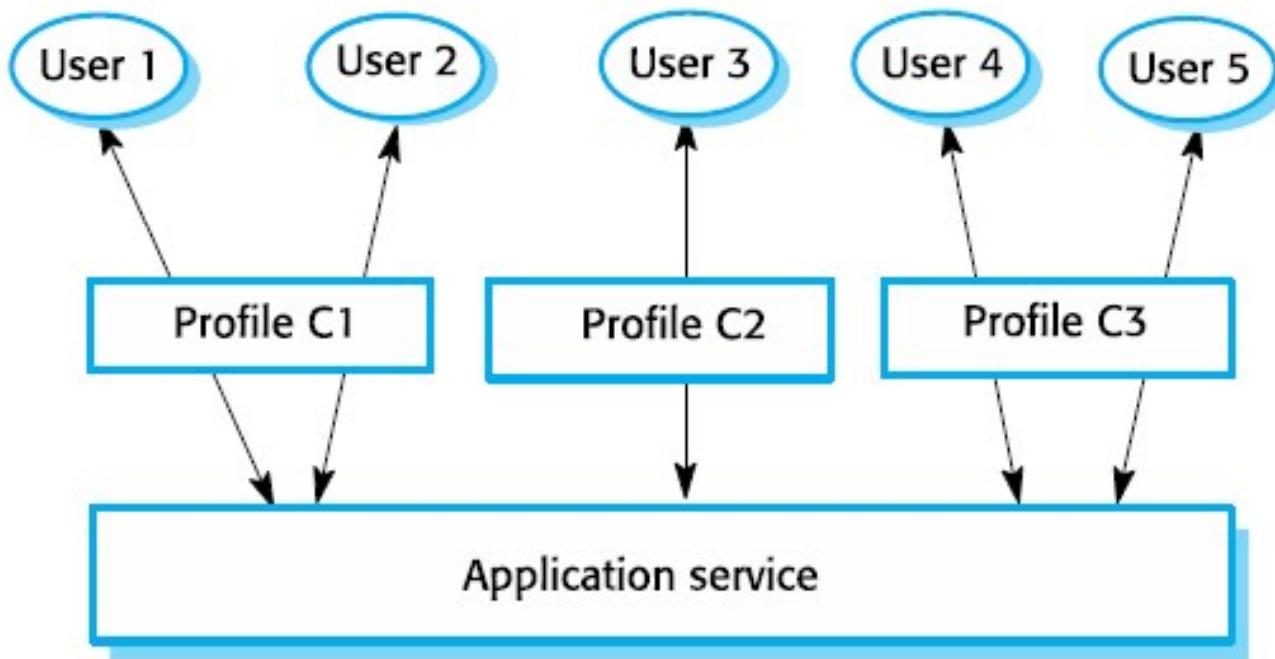
## SaaS and SOA

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

# Implementation factors for SaaS

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

# Configuration of a software system offered as a service



# Service configuration

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

# Multi-tenancy

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

# Scalability

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

# Key points

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

# Key points

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

# Service-oriented Software Engineering

Lecture 24

# Topics Covered

- Service oriented architectures
- RESTful Services
- Service engineering
- Service composition

# Web services

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

# Reusable services

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

## Benefits of service-oriented approach

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

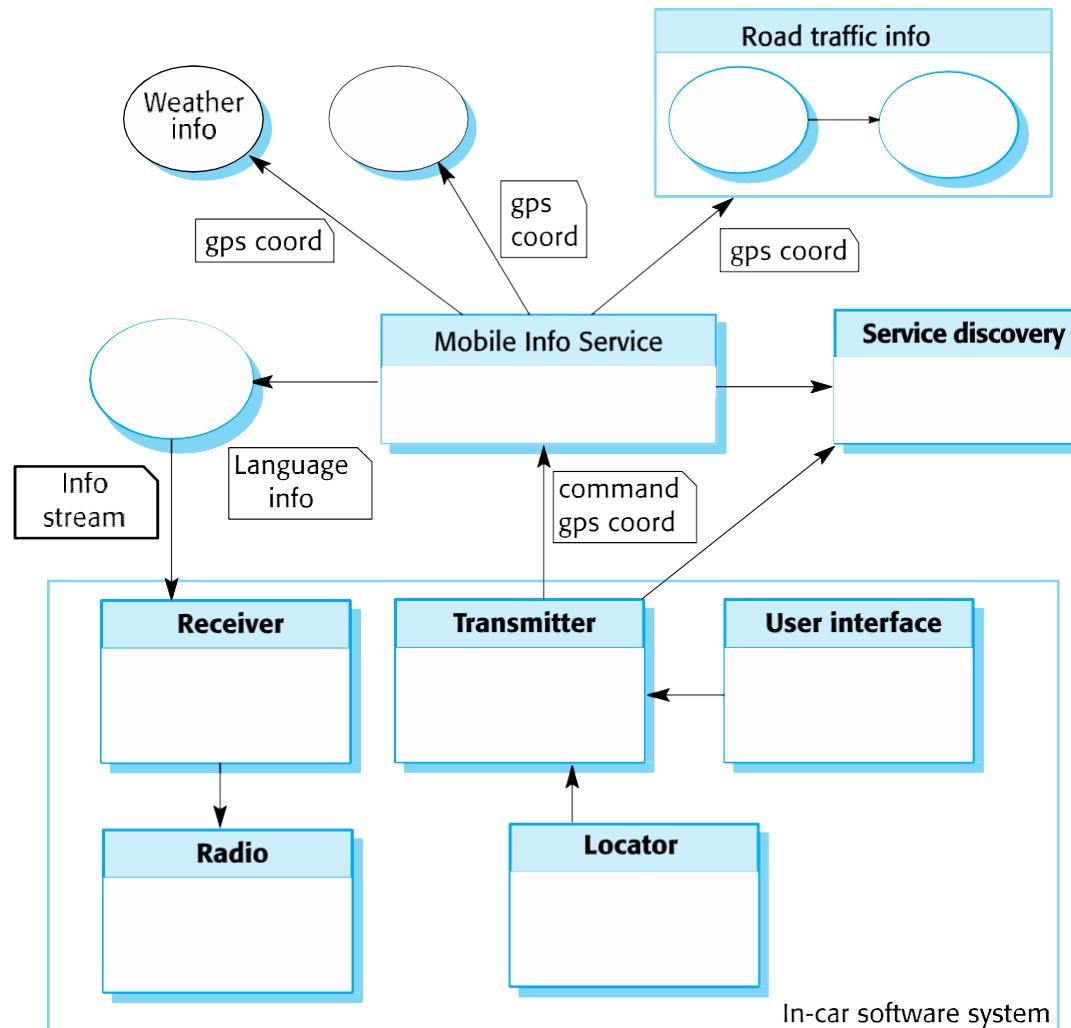
# Benefits of a service-oriented approach

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

## Services scenario

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

# A service-based, in-car information system



# Advantage of SOA for this application

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

# Service-oriented software engineering

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

# Service-oriented architecture

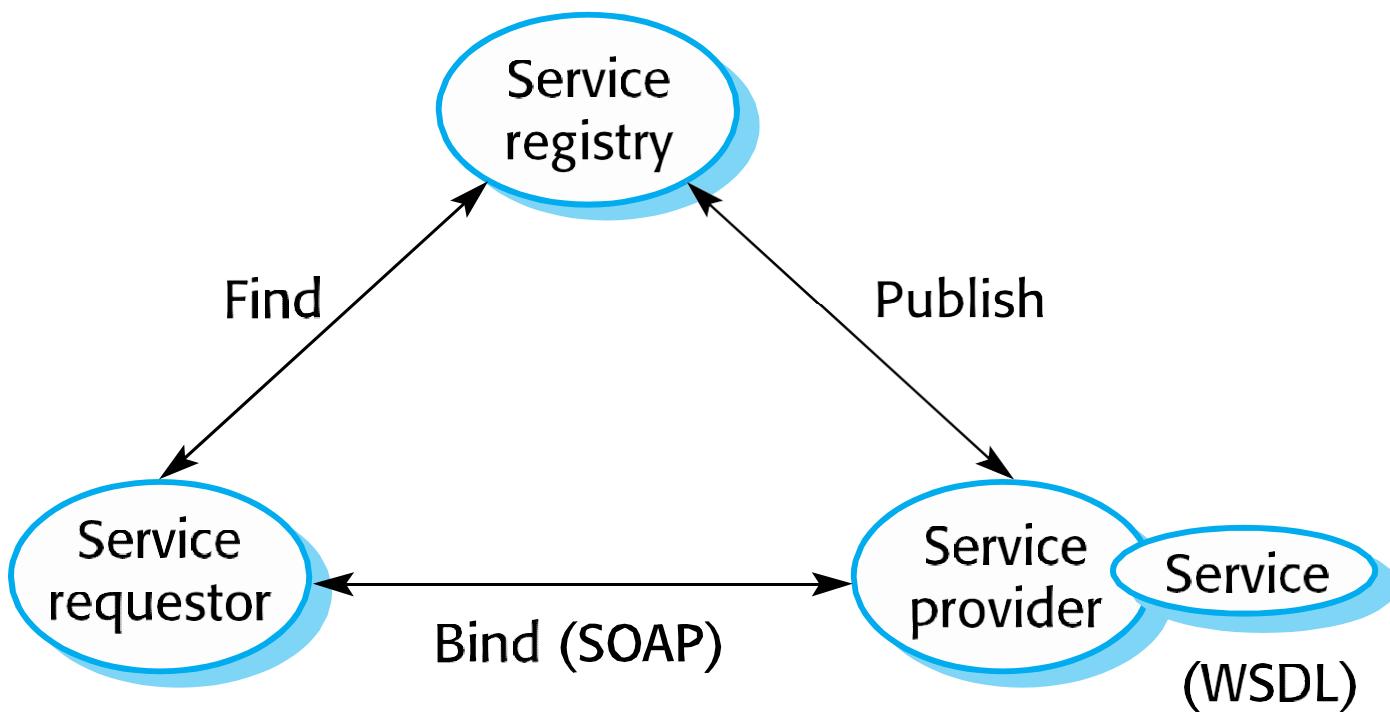
# Service-oriented architectures

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

# Key standards

- SOAP
  - Simple Object Access Protocol A message exchange standard that supports service communication
- WSDL (Web Service Definition Language)
  - This standard allows a service interface and its bindings to be defined
- The Web Services Business Process Execution Language (WS-BPEL), commonly known as BPEL (Business Process Execution Language), executable language for specifying actions within business processes with web services. Processes in BPEL export and import information by using web service interfaces exclusively.
- The Universal Description, Discovery, and Integration (**UDDI**) protocol is an approved OASIS Standard and a key member of the Web services stack. It defines a standard method for publishing and discovering the network-based software components of a service-oriented architecture (SOA)

# Service-oriented architecture



# Benefits of SOA

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

# Web service standards

XML technologies (XML, XSD, XSLT, ....)

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

Process (WS-BPEL)

Service definition (UDDI, WSDL)

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

# Service-oriented software engineering

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

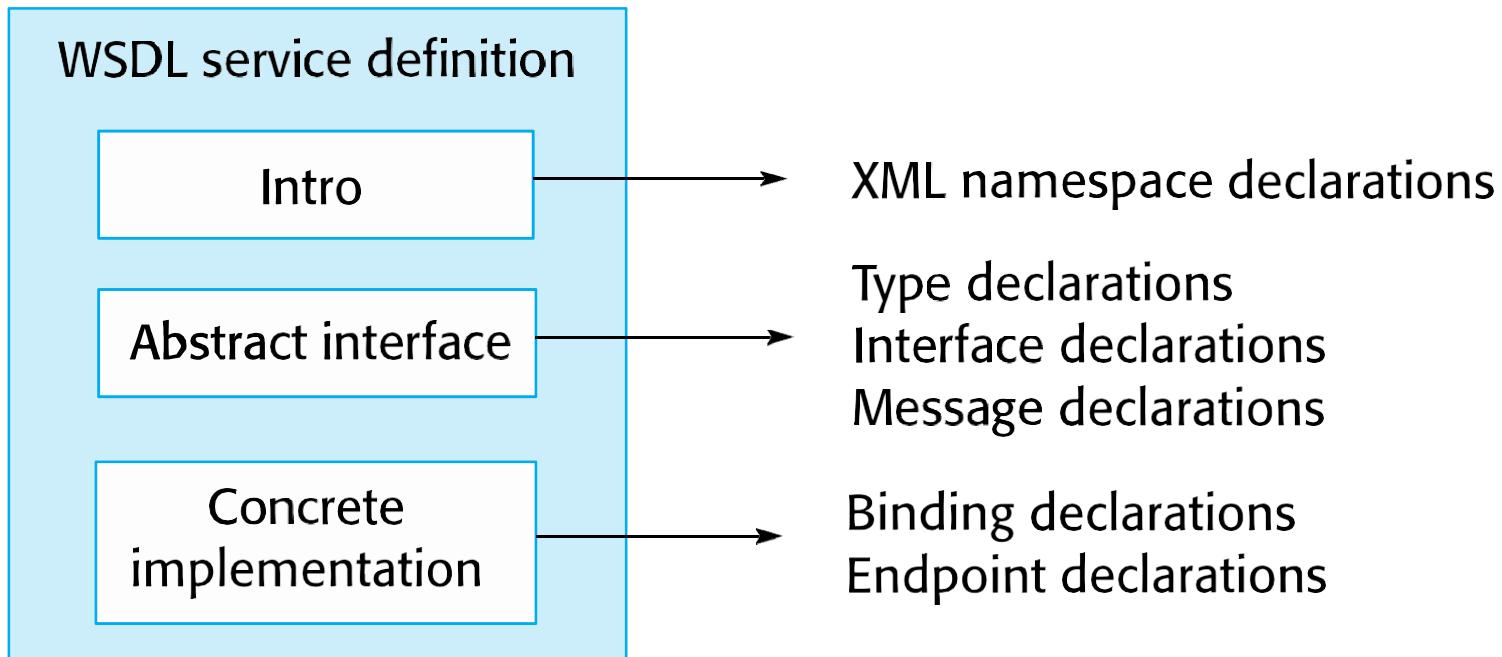
# Services as reusable components

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

# Web service description language

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

# Organization of a WSDL specification



## WSDL specification components

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

# Part of a WSDL description for a web service

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

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

    <xs: complexType name = “pdrec”
      <xs: sequence>
        <xs:element name = “town” type = “xs:string”/>
        <xs:element name = “country” type = “xs:string”/>
        <xs:element name = “day” type = “xs:date” />
      </xs:complexType>
    Definitions of MaxMinType and InDataFault here
  </schema>
</types>
```

# Part of a WSDL description for a web service

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

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

# RESTful web services

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

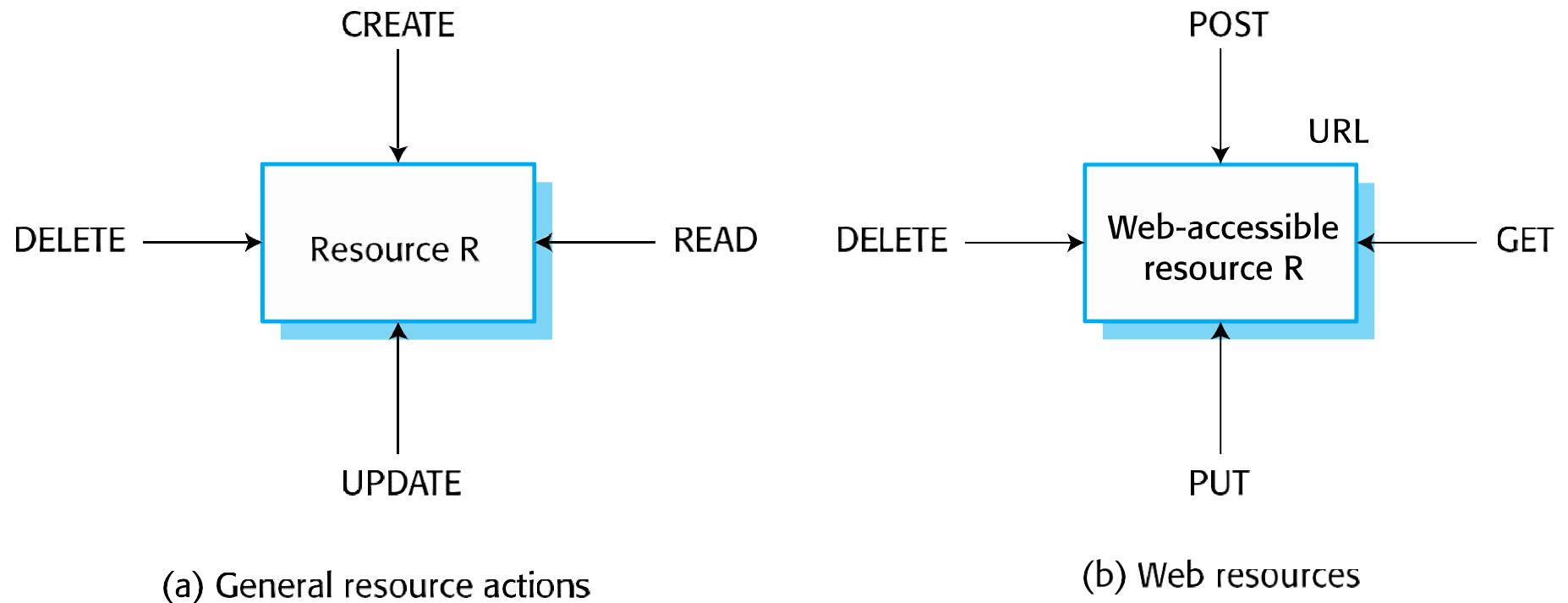
# Resources

- ✧ The fundamental element in a RESTful architecture is a
  - resource.
- ✧ Essentially, a resource is simply a data element such as a catalog, a medical record, or a document, such as this book chapter.
- ✧ In general, resources may have multiple representations
  - i.e. they can exist in different formats.
    - MS WORD
    - PDF
    - Quark XPress

## Resource operations

- ✧ Create – bring the resource into existence.
- ✧ Read – return a representation of the resource.
- ✧ Update – change the value of the resource.
- ✧ Delete – make the resource inaccessible.

# Resources and actions



# Operation functionality

- POST is used to create a resource. It has associated data that defines the resource.
- GET is used to read the value of a resource and return that to the requestor in the specified representation, such as XHTML, that can be rendered in a web browser.
- PUT is used to update the value of a resource.
- DELETE is used to delete the resource

# Resource access

- When a RESTful approach is used, the data is exposed and is accessed using its URL.
- Therefore, the weather data for each place in the database, might be accessed using URLs such as:
- `http://weather-info-example.net/temperatures/boston`  
`http://weather-info-example.net/temperatures/edinburgh`
- Invokes the GET operation and returns a list of maximum and minimum temperatures.
- To request the temperatures for a specific date, a URL query is used:
- `http://weather-info-example.net/temperatures/edinburgh?date=20140226`

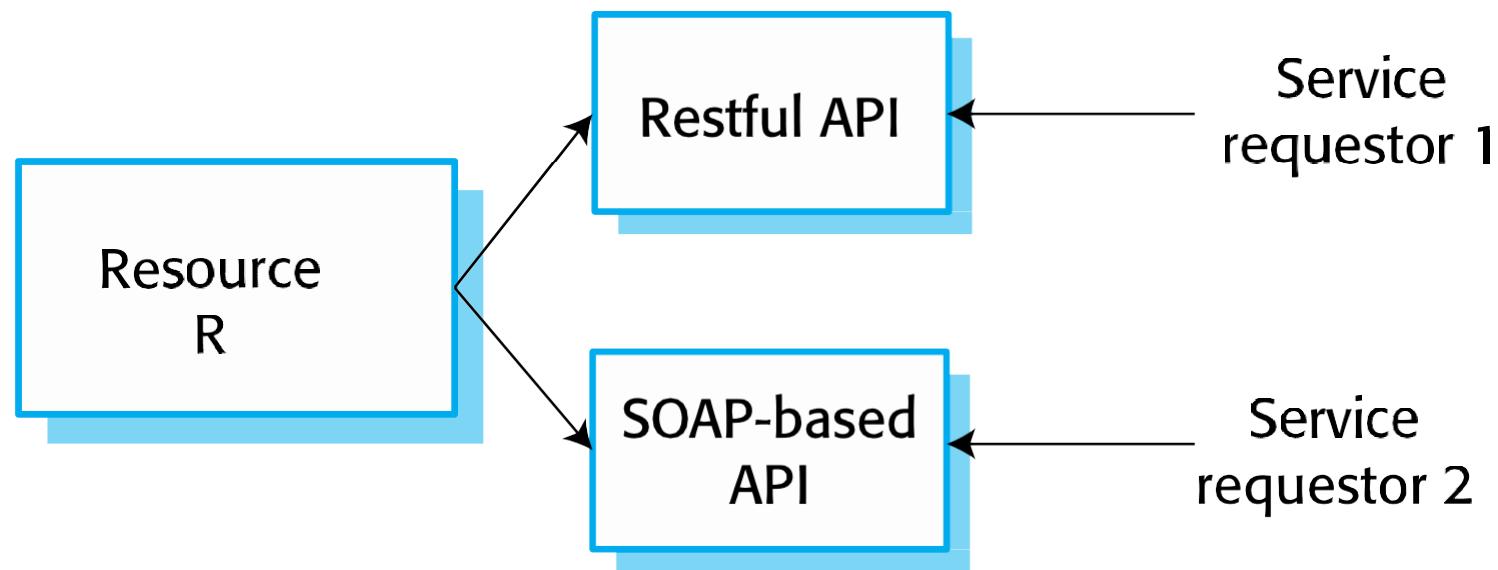
# Query results

- ✧ The response to a GET request in a RESTful service
  - may include URLs.
- ✧ If the response to a request is a set of resources, then the URL of each of these may be included.
  - <http://weather-info-example.net/temperatures/edinburgh-scotland>  
<http://weather-info-example.net/temperatures/edinburgh-australia>
  - <http://weather-info-example.net/temperatures/edinburgh-maryland>
  - [maryland](#)

# Disadvantages of RESTful approach

- When a service has a complex interface and is not a simple resource, it can be difficult to design a set of RESTful services to represent this.
- There are no standards for RESTful interface description so service users must rely on informal documentation to understand the interface.
- When you use RESTful services, you have to implement your own infrastructure for monitoring and managing the quality of service and the service reliability.

# RESTful and SOAP-based APIs

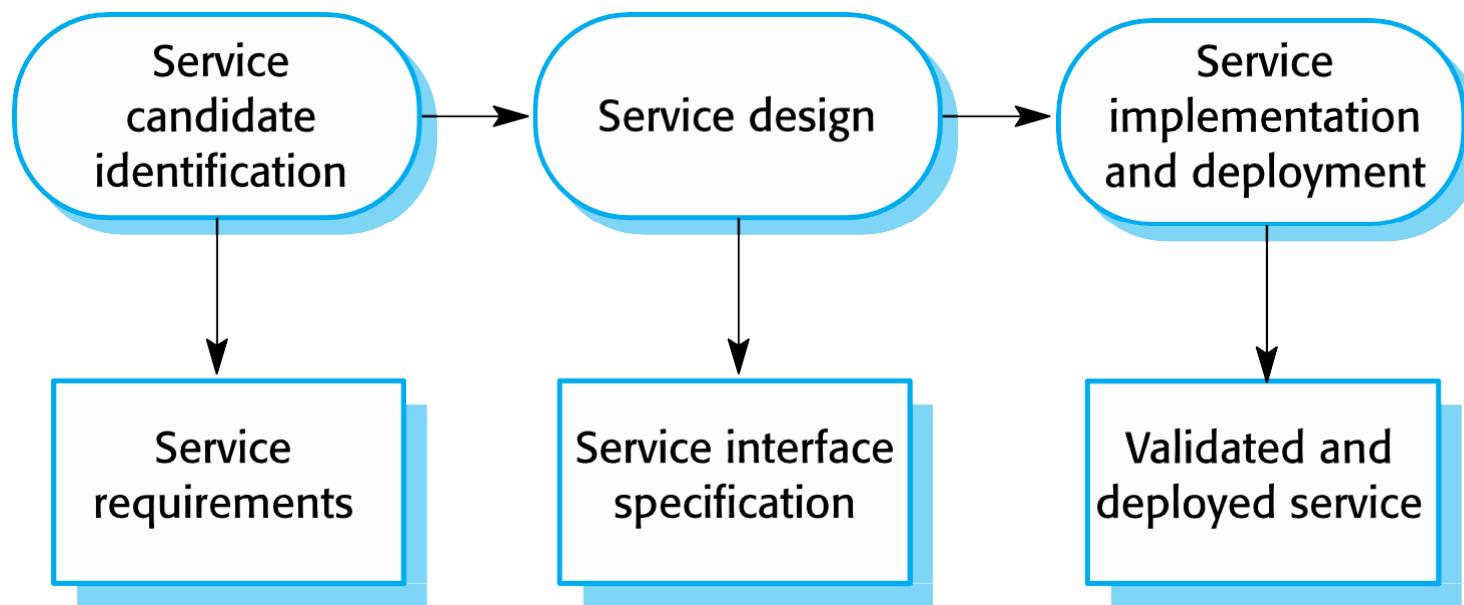


# Service engineering

# Service engineering

- The process of developing services for reuse in service- oriented applications
- The service has to be designed as a reusable abstraction that can be used in different systems.
- Generally useful functionality associated with that abstraction must be designed and the service must be robust and reliable.
- The service must be documented so that it can be discovered and understood by potential users

# The service engineering process



## **Stages of service engineering**

- ✧ Service candidate identification, where you identify possible services that might be implemented and define the service requirements.
- ✧ Service design, where you design the logical service interface and its implementation interfaces (SOAP and/or RESTful)
- ✧ Service implementation and deployment, where you implement and test the service and make it available for use.

# Service candidate identification

- Services should support business processes.
- Service candidate identification involves understanding an organization's business processes to decide which reusable services could support these processes.
- Three fundamental types of service
  - Utility services that implement general functionality used by different business processes.
  - Business services that are associated with a specific business function e.g., in a university, student registration.
  - Coordination services that support composite processes such as ordering.

# Task and entity-oriented services

- Task-oriented services are those associated with some activity.
- Entity-oriented services are like objects. They are associated with a business entity such as a job application form.
- Utility or business services may be entity- or task- oriented, coordination services are always task-oriented

# Service classification

	Utility	Business	Coordination
Task	Currency converter Employee locator	Validate claim form Check credit rating	Process expense claim Pay external supplier
Entity	Document style checker Web form to XML converter	Expenses form Student application form	

# Service identification

- Is the service associated with a single logical entity used in different business processes?
- Is the task one that is carried out by different people in the organisation? Can this fit with a RESTful model?
- Is the service independent?
- Does the service have to maintain state? Is a database required?
- Could the service be used by clients outside the organisation?
- Are different users of the service likely to have different non-functional requirements?

# Service identification example

- ✧ A large company, which sells computer equipment, has arranged special prices for approved configurations for some customers.
- ✧ To facilitate automated ordering, the company wishes to produce a catalog service that will allow customers to select the equipment that they need.
- ✧ Unlike a consumer catalog, orders are not placed directly through a catalog interface. Instead, goods are ordered through the web-based procurement system of each company that accesses the catalog as a web service.
- ✧ Most companies have their own budgeting and approval procedures for orders and their own ordering process must be followed when an order is placed.

# Catalog services

- ✧ Created by a supplier to show which good can be ordered from them by other companies
- ✧ Service requirements
  - Specific version of catalogue should be created for each client
  - Catalogue shall be downloadable
  - The specification and prices of up to 6 items may be compared
  - Browsing and searching facilities shall be provided
  - A function shall be provided that allows the delivery date for ordered items to be predicted
  - Virtual orders shall be supported which reserve the goods for 48 hours to allow a company order to be placed

# Catalogue: Non-functional requirements

- ✧ Access shall be restricted to employees of accredited organisations
- ✧ Prices and configurations offered to each organisation shall be confidential
- ✧ The catalogue shall be available from 0700 to 1100
- ✧ The catalogue shall be able to process up to 10 requests per second

# Functional descriptions of catalog service operations

Operation	Description
MakeCatalog	Creates a version of the catalog tailored for a specific customer. Includes an optional parameter to create a downloadable PDF version of the catalog.
Lookup	Displays all of the data associated with a specified catalog item.
Search	This operation takes a logical expression and searches the catalog according to that expression. It displays a list of all items that match the search expression.

# Functional descriptions of catalog service operations

Operation	Description
Compare	Provides a comparison of up to six characteristics (e.g., price, dimensions, processor speed, etc.) of up to four catalog items.
CheckDelivery	Returns the predicted delivery date for an item if ordered that day.
MakeVirtualOrder	Reserves the number of items to be ordered by a customer and provides item information for the customer's own procurement system.

# Service interface design

- Involves thinking about the operations associated with the service and the messages exchanged
- The number of messages exchanged to complete a service request should normally be minimised.
- Service state information may have to be included in messages

# Interface design stages

- Logical interface design
  - Starts with the service requirements and defines the operation names and parameters associated with the service. Exceptions should also be defined
- Message design (SOAP)
  - For SOAP-based services, design the structure and organisation of the input and output messages. Notations such as the UML are a more abstract representation than XML
  - The logical specification is converted to a WSDL description
- Interface design (REST)
  - Design how the required operations map onto REST operations and what resources are required.

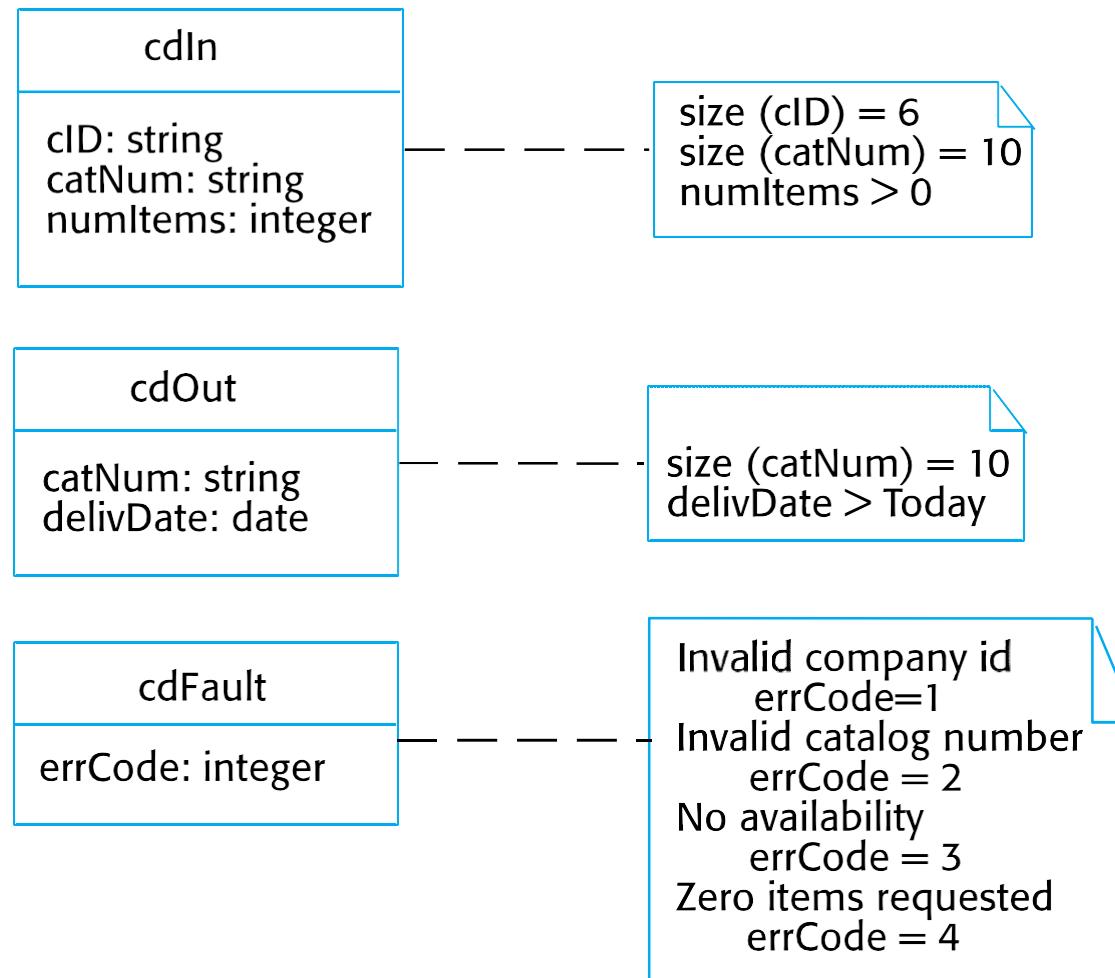
# Catalog interface design

Operation	Inputs	Outputs	Exceptions
MakeCatalog	$mcIn$ Company id PDF-flag	$mcOut$ URL of the catalog for that company	$mcFault$ Invalid company id
Lookup	$lookIn$ Catalog URL Catalog number	$lookOut$ URL of page with the item information	$lookFault$ Invalid catalog number
Search	$searchIn$ Catalog URL Search string	$searchOut$ URL of web page with search results	$searchFault$ Badly formed search string

# Catalog interface design

Operation	Inputs	Outputs	Exceptions
Compare	$compIn$ Catalog URL Entry attribute (up to 6) Catalog number (up to 4)	$compOut$ URL of page showing comparison table	$compFault$ Invalid company id Invalid catalog number Unknown attribute
CheckDelivery	$cdIn$ Company id Catalog number Number of items required	$cdOut$ Catalog number Expected delivery date	$cdFault$ Invalid company id No availability Zero items requested
MakeVirtualOrder	$poIn$ Company id Number of items required Catalog number	$poOut$ Catalog number Number of items required Predicted delivery date Unit price estimate Total price estimate	$poFault$ Invalid company id Invalid catalog number Zero items requested

# UML definition of input and output messages



# RESTful Interface

- There should be a resource representing a company-specific catalog. This should have a URL of the form `<base catalog>/<company name>` and should be created using a POST operation.
- Each catalog item should have its own URL of the form: `<base catalog>/<company name>/<item identifier>`.
- The GET operation is used to retrieve items.
- Lookup is implemented by using the URL of an item in a catalog as the GET parameter.
- Search is implemented by using GET with the company catalog as the URL and the search string as a query parameter. This GET operation returns a list of URLs of the items matching the search

# RESTful interface

- ✧ The **Compare** operation can be implemented as a sequence of GET operations, to retrieve the individual items, followed by a POST operation to create the comparison table and a final GET operation to return this to the user.
- ✧ The **CheckDelivery** and **MakeVirtualOrder** operations require an additional resource, representing a virtual order.
  - A POST operation is used to create this resource with the number of items required. The company id is used to automatically fill in the order form and the delivery date is calculated. This can then be retrieved using a GET operation.

# Service implementation and deployment

- Programming services using a standard programming language or a workflow language
- Services then have to be tested by creating input messages and checking that the output messages produced are as expected
- Deployment involves publicising the service and installing it on a web server. Current servers provide support for service installation

# Legacy system services

- Services can be implemented by implementing a service interface to existing legacy systems
- Legacy systems offer extensive functionality and this can reduce the cost of service implementation
- External applications can access this functionality through the service interfaces

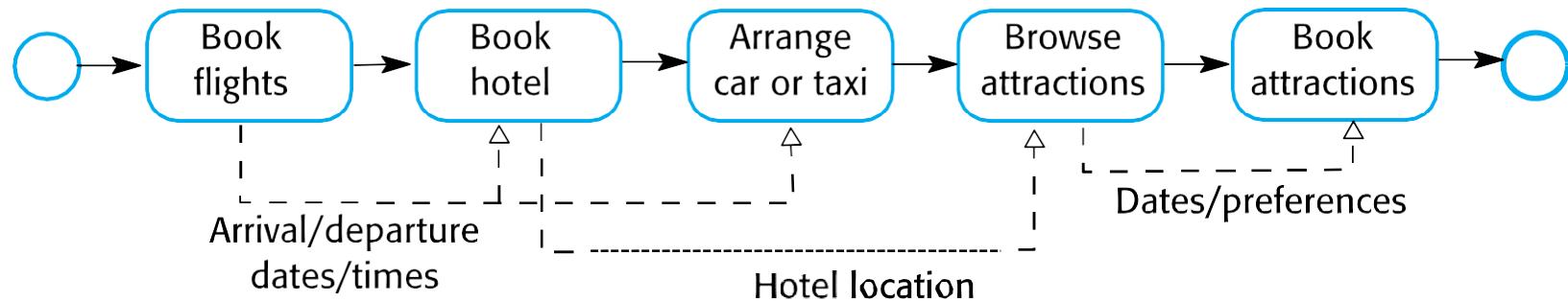
# Service descriptions

- Information about your business, contact details, etc. This is important for trust reasons. Users of a service have to be confident that it will not behave maliciously.
- An informal description of the functionality provided by the service. This helps potential users to decide if the service is what they want.
- A description of how to use the services     SOAP-based and RESTful.
- Subscription information that allows users to register for information about updates to the service.

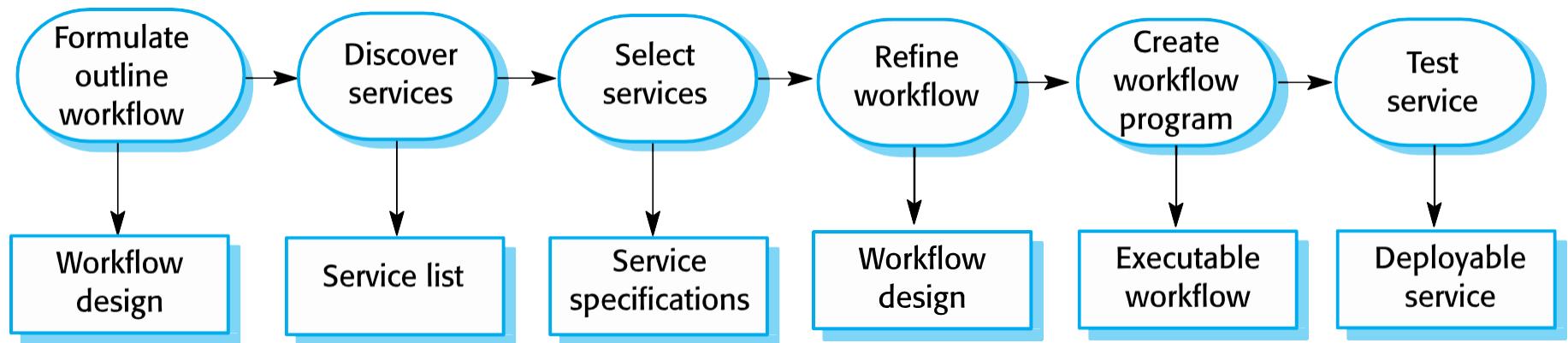
# Software development with services

- Existing services are composed and configured to create new composite services and applications
- The basis for service composition is often a workflow
  - Workflows are logical sequences of activities that, together, model a coherent business process
  - For example, provide a travel reservation services which allows flights, car hire and hotel bookings to be coordinated

# Vacation package workflow



# Service construction by composition



# Construction by composition

- Formulate outline workflow
  - In this initial stage of service design, you use the requirements for the composite service as a basis for creating an ‘ideal’ service design.
- Discover services
  - During this stage of the process, you search service registries or catalogs to discover what services exist, who provides these services and the details of the service provision.
- Select possible services
  - Your selection criteria will obviously include the functionality of the services offered. They may also include the cost of the services and the quality of service (responsiveness, availability, etc.) offered.

# Construction by composition

## ✧ *Refine workflow.*

- This involves adding detail to the abstract description and perhaps adding or removing workflow activities.

## ✧ *Create workflow program*

- During this stage, the abstract workflow design is transformed to an executable program and the service interface is defined. You can use a conventional programming language, such as Java or a workflow language, such as WS-BPEL.

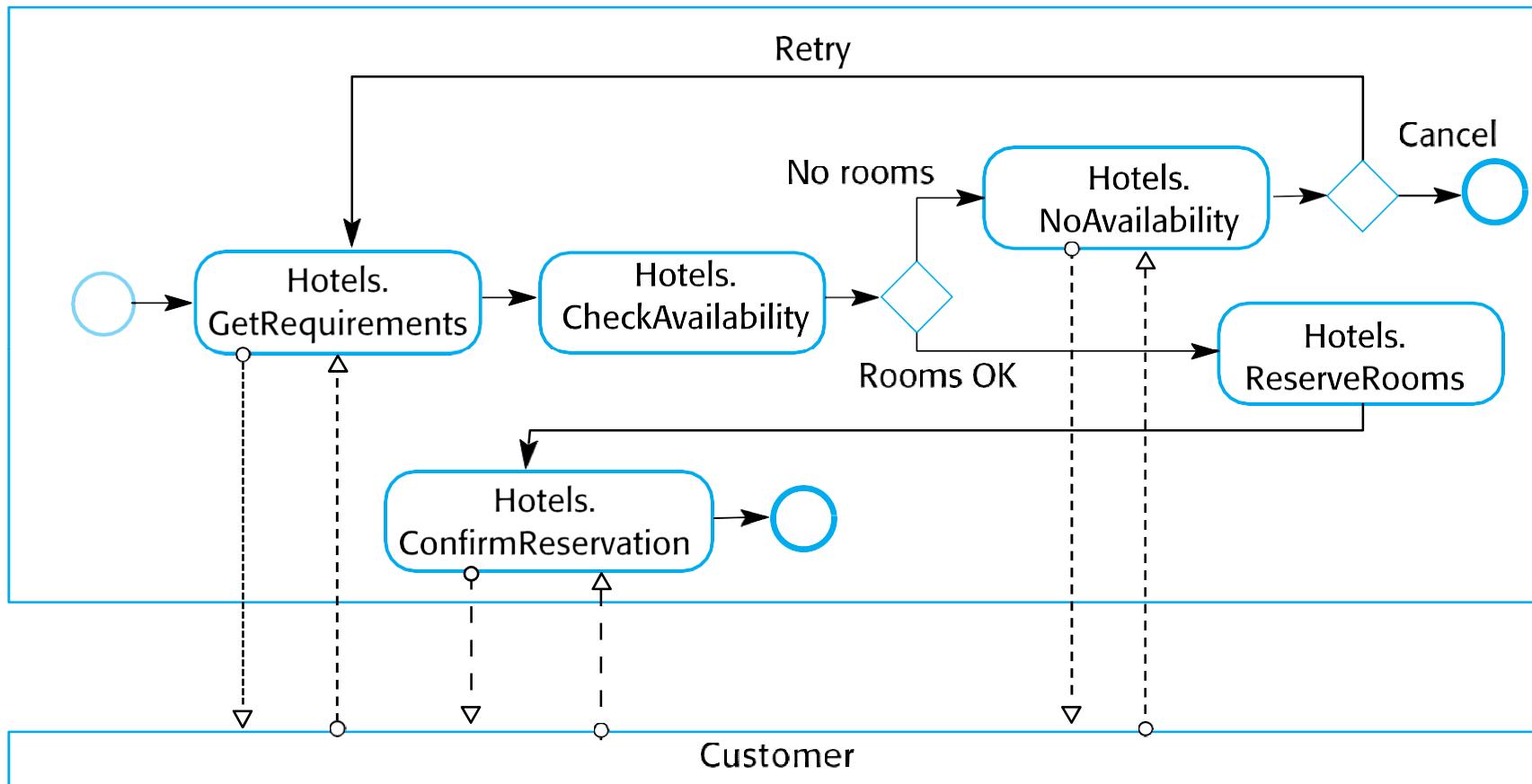
## ✧ *Test completed service or application*

- The process of testing the completed, composite service is more complex than component testing in situations where external services are used.

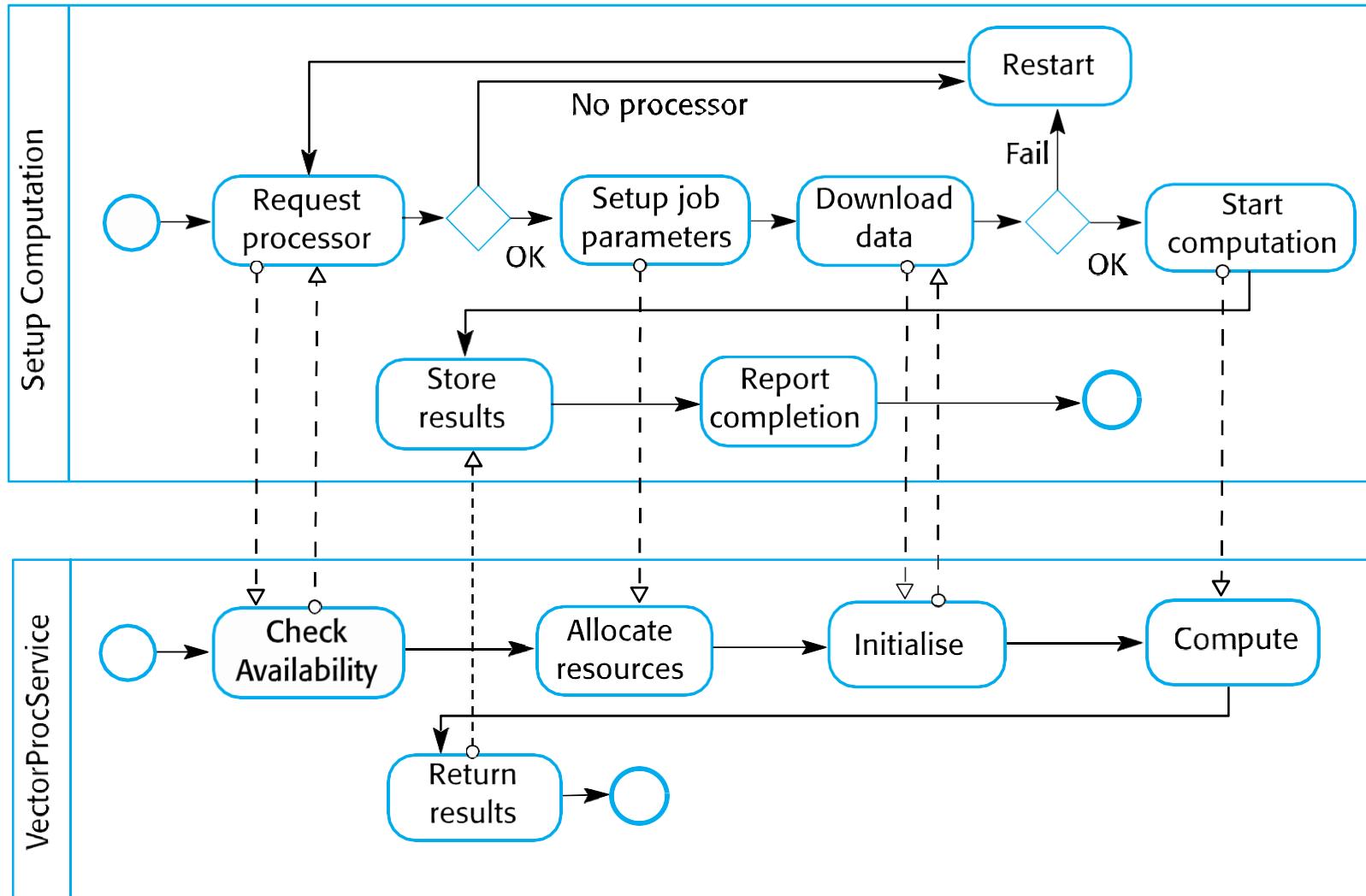
## Workflow design and implementation

- ❖ WS-BPEL is an XML-standard for workflow specification. However, WS-BPEL descriptions are long and unreadable
- ❖ Graphical workflow notations, such as BPMN, are more readable and WS-BPEL can be generated from them
- ❖ In inter-organisational systems, separate workflows are created for each organisation and linked through message exchange.
- ❖ Workflows can be used with both SOAP-based and RESTful services.

# A fragment of a hotel booking workflow



# Interacting workflows



## Testing service compositions

- ✧ Testing is intended to find defects and demonstrate that the system meets its functional and non-functional requirements.
- ✧ Service testing is difficult as (external) services are ‘black-boxes’. Testing techniques that rely on the program source code cannot be used.

# Service testing problems

- External services may be modified by the service provider thus invalidating tests which have been completed.
- Dynamic binding means that the service used in an application may vary - the application tests are not, therefore, reliable.
- The non-functional behaviour of the service is unpredictable because it depends on load.
- If services have to be paid for as used, testing a service may be expensive.
- It may be difficult to invoke compensating actions in external services as these may rely on the failure of other services which cannot be simulated.

# Key points

- ✧ Service-oriented architecture is an approach to software engineering where reusable, standardized services are the basic building blocks for application systems.
- ✧ Services may be implemented within a service-oriented architecture using a set of XML-based web service standards. These include standards for service communication, interface definition and service enactment in workflows.
- ✧ Alternatively, a RESTful architecture may be used which is based on resources and standard operations on these resources.
- ✧ A RESTful approach uses the http and https protocols for service communication and maps operations on the standard http verbs POST, GET, PUT and DELETE.

# Key Points

- ✧ Utility services provide general-purpose functionality; business services implement part of a business process; coordination services coordinate service execution.
- ✧ Service engineering involves identifying candidate services for implementation, defining service interfaces and implementing, testing and deploying services.
- ✧ The development of software using services involves composing and configuring services to create new composite services and systems.
- ✧ Graphical workflow languages, such as BPMN, may be used to describe a business process and the services used in that process

# Reference

- Chapter 18 Service-oriented Software Engineering ; Software Engineering; Ian Sommerville

# Reliability Engineering

Lecture 25

## Topics covered

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

# Software reliability

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

# Faults, errors and failures

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

# Faults and failures

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

# Fault management

## ✧ Fault avoidance

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

## ✧ Fault detection

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

## ✧ Fault tolerance

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

# Reliability achievement

## ✧ Fault avoidance

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

## ✧ Fault detection and removal

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

## ✧ Fault tolerance

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

Availability and  
reliability

# Availability and reliability

## ✧ Reliability

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

## ✧ Availability

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

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

# Reliability and specifications

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

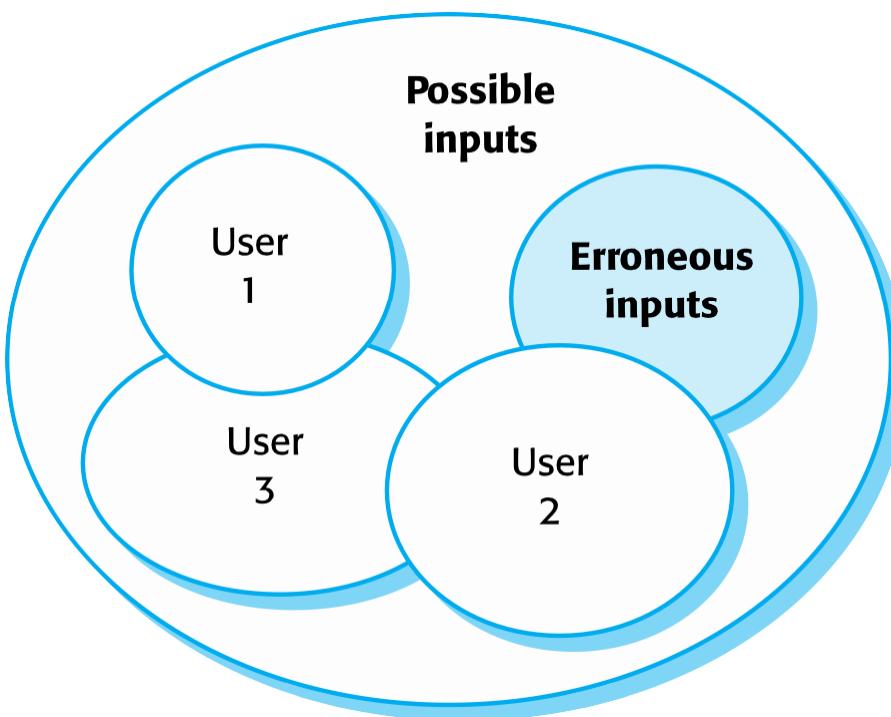
# Perceptions of reliability

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

# Availability perception

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

# Software usage patterns



# Reliability in use

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

# Reliability requirements

# System reliability requirements

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

# Reliability metrics

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

# Probability of failure on demand (POFOD)

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

# Rate of fault occurrence (ROCOF)

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

# Availability

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

# Availability specification

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

## Non-functional reliability requirements

- ✧ Non-functional reliability requirements are specifications of the required reliability and availability of a system using one of the reliability metrics (POFOD, ROCOF or AVAIL).
- ✧ Quantitative reliability and availability specification has been used for many years in safety-critical systems but is uncommon for business critical systems.
- ✧ However, as more and more companies demand 24x7 service from their systems, it makes sense for them to be precise about their reliability and availability expectations.

# Benefits of reliability specification

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

## Specifying reliability requirements

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

# ATM reliability specification

- ✧ Key concerns

- To ensure that their ATMs carry out customer services as requested and that they properly record customer transactions in the account database.
- To ensure that these ATM systems are available for use when required.

- ✧ Database transaction mechanisms may be used to correct transaction problems so a low-level of ATM reliability is all that is required

- ✧ Availability, in this case, is more important than reliability

# ATM availability specification

- ✧ System services

- The customer account database service;
- The individual services provided by an ATM such as ‘withdraw cash’, ‘provide account information’, etc.

- ✧ The database service is critical as failure of this service means that all of the ATMs in the network are out of action.

- ✧ You should specify this to have a high level of availability.

- Database availability should be around 0.9999, between 7 am and 11pm.
- This corresponds to a downtime of less than 1 minute per week.

## ATM availability specification

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

# Insulin pump reliability specification

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

# Functional reliability requirements

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

# Examples of functional reliability requirements

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

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

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

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

# Fault-tolerant architectures

## Fault tolerance

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

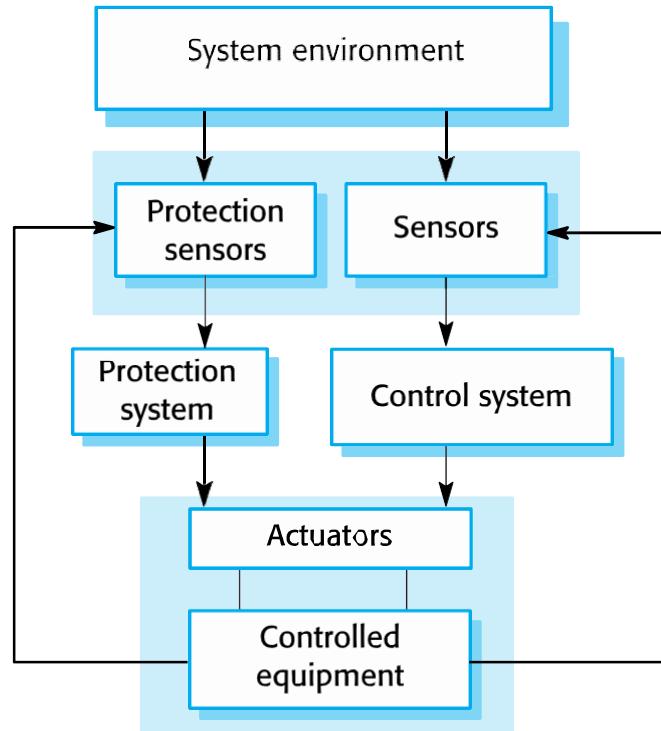
# Fault-tolerant system architectures

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

# Protection systems

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

# Protection system architecture



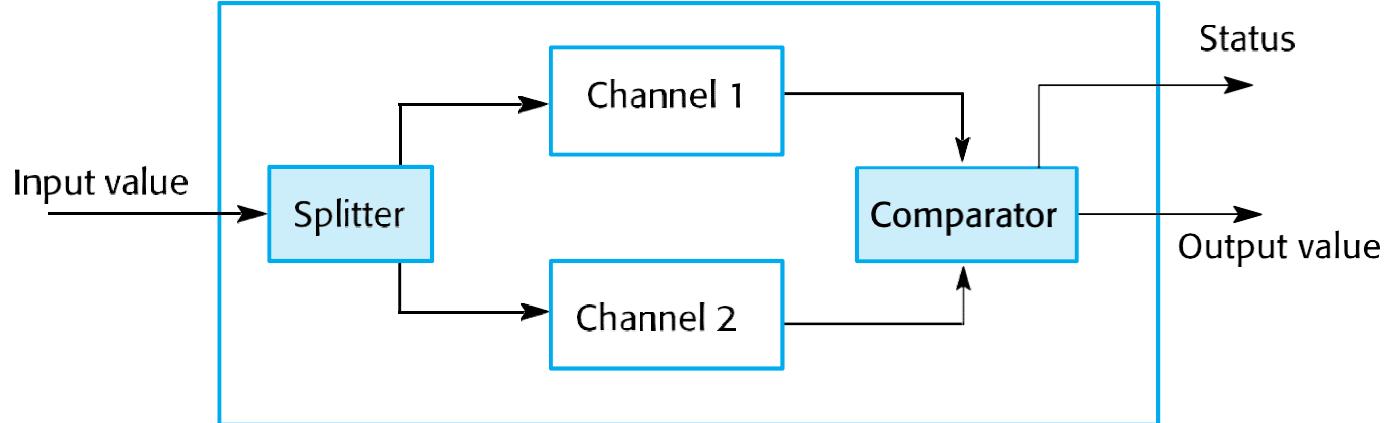
# Protection system functionality

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

# Self-monitoring architectures

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

# Self-monitoring architecture



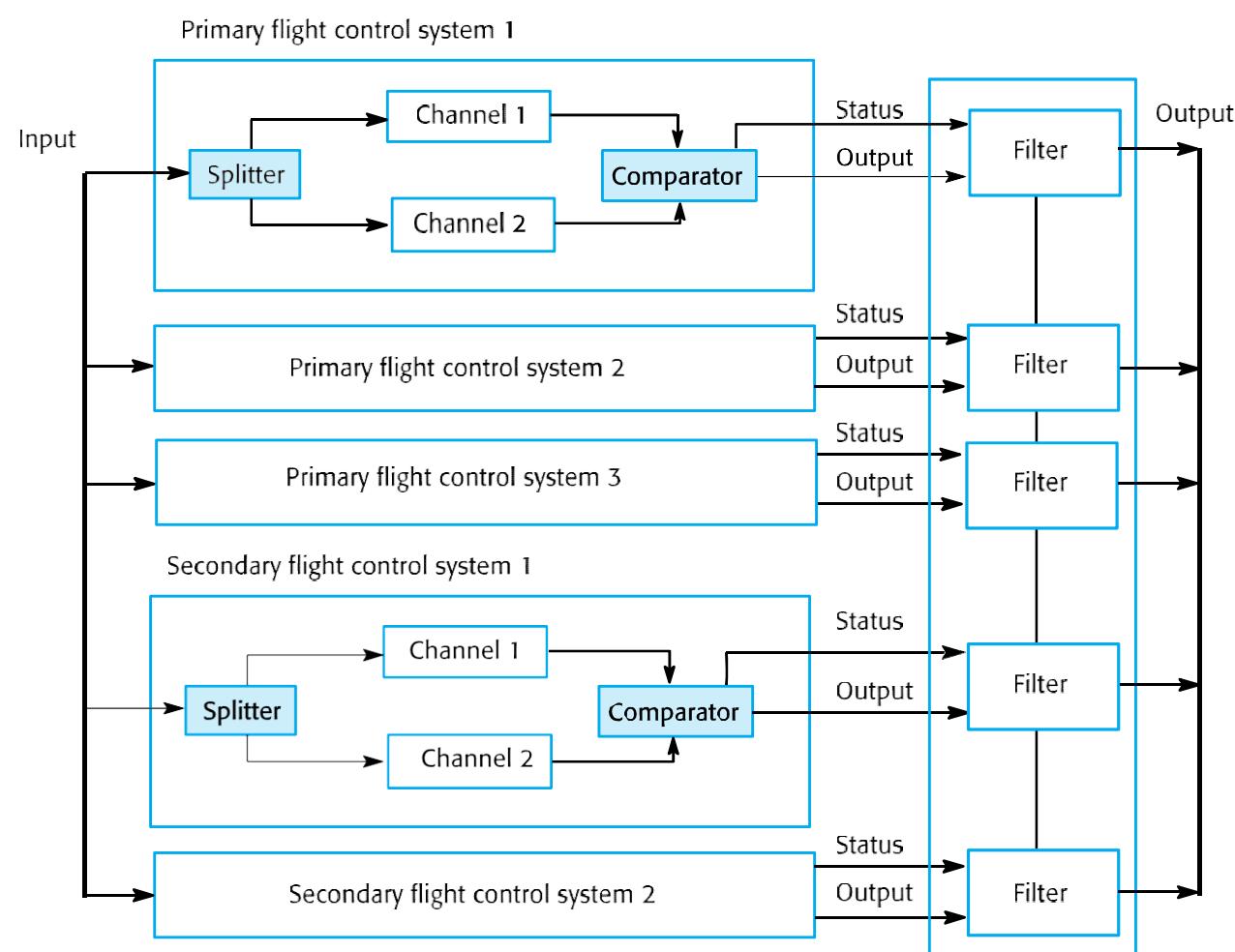
# Self-monitoring systems

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

# Airbus architecture discussion

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

# Airbus flight control system architecture



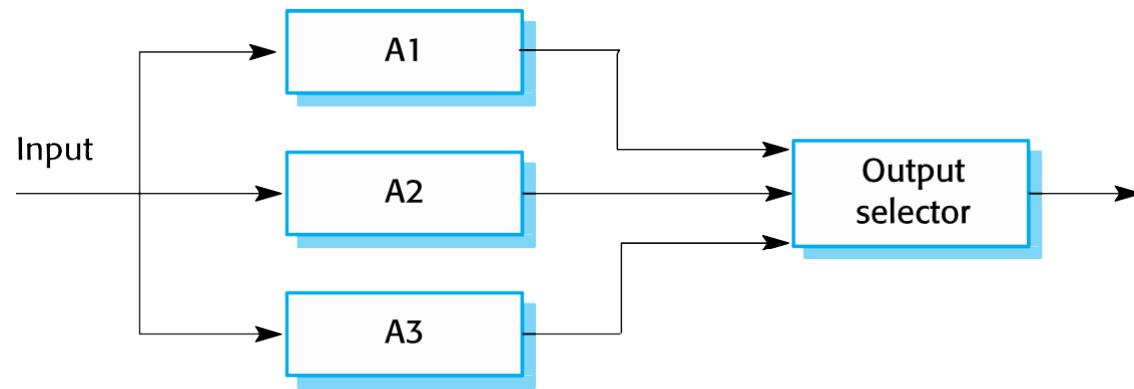
## N-version programming

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

## Hardware fault tolerance

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

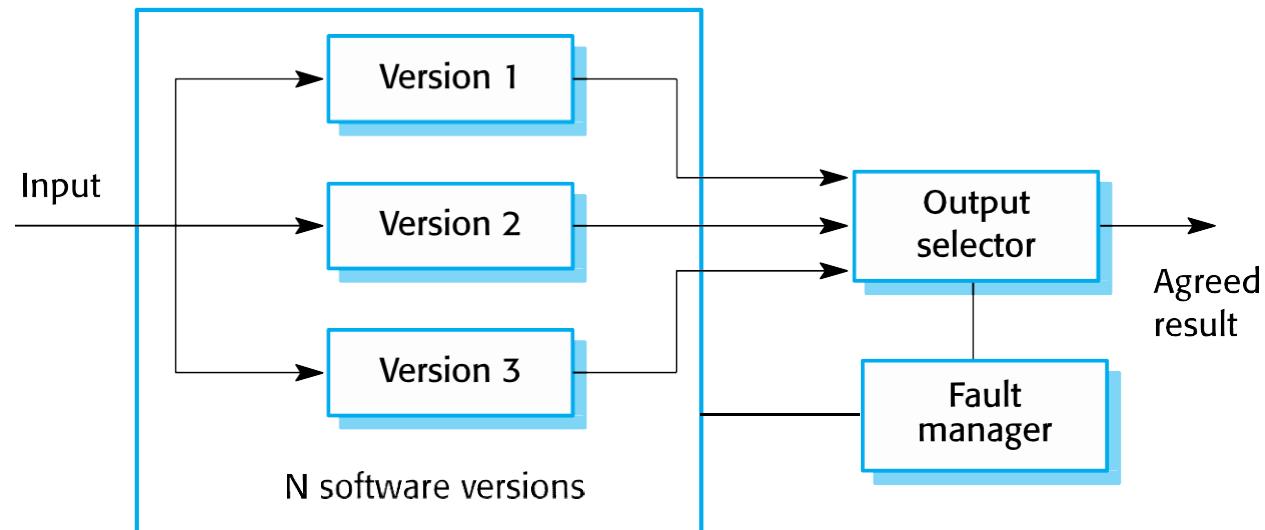
# Triple modular redundancy



## N-version programming

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

# N-version programming



# Software diversity

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

# Problems with design diversity

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

# Specification dependency

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

# Improvements in practice

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

# Programming for reliability

# Dependable programming

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

# Good practice guidelines for dependable programming

## Dependable programming guidelines

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

# (1) Limit the visibility of information in a program

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

## (2) Check all inputs for validity

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

# Validity checks

## ✧ Range checks

- Check that the input falls within a known range.

## ✧ Size checks

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

## ✧ Representation checks

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

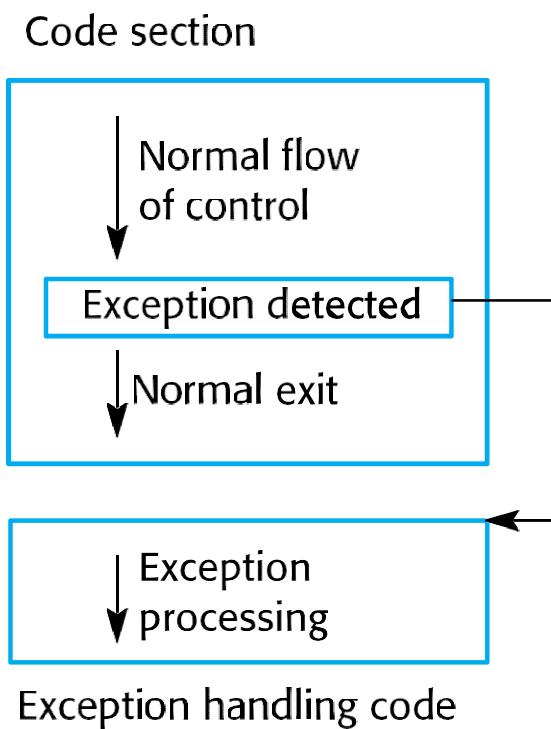
## ✧ Reasonableness checks

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

### **(3) Provide a handler for all exceptions**

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

# Exception handling



# Exception handling

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

## (4) Minimize the use of error-prone constructs

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

# Error-prone constructs

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

# Error-prone constructs

## ✧ Parallelism

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

## ✧ Recursion

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

## ✧ Interrupts

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

## ✧ Inheritance

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

# Error-prone constructs

## ✧ Aliasing

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

## ✧ Unbounded arrays

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

## ✧ Default input processing

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

# (5) Provide restart capabilities

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

## (6) Check array bounds

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

## **(7) Include timeouts when calling external components**

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

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

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

# Reliability measurement

# Reliability measurement

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

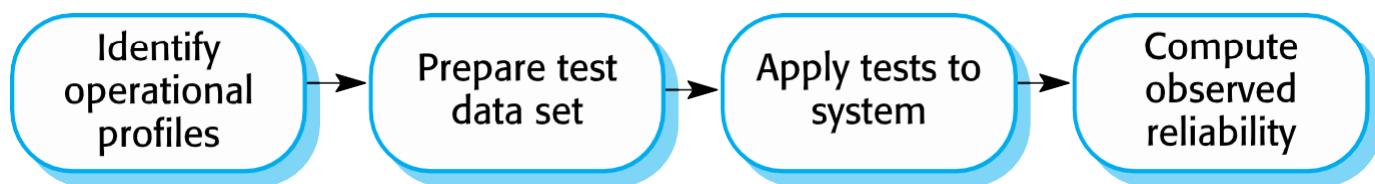
## Reliability testing

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

# Statistical testing

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

# Reliability measurement



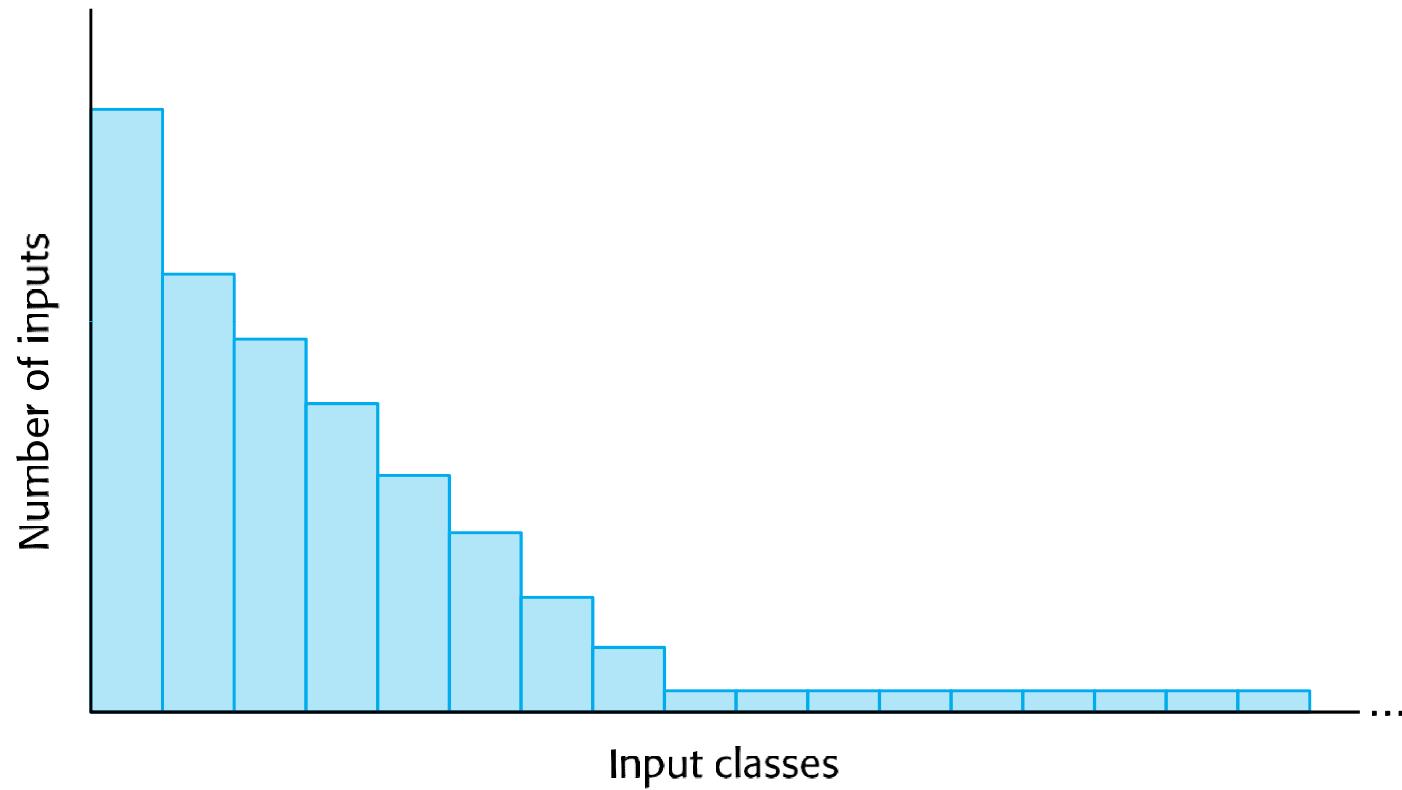
# Reliability measurement problems

- ✧ Operational profile uncertainty
  - The operational profile may not be an accurate reflection of the real use of the system.
- ✧ High costs of test data generation
  - Costs can be very high if the test data for the system cannot be generated automatically.
- ✧ Statistical uncertainty
  - You need a statistically significant number of failures to compute the reliability but highly reliable systems will rarely fail.
- ✧ Recognizing failure
  - It is not always obvious when a failure has occurred as there may be conflicting interpretations of a specification.

## Operational profiles

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

# An operational profile



## Operational profile generation

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

## Key points

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

# Key points

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

## Key points

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

# Reference

- Chapter 11 Reliability Engineering; Software engineering 10th Edition; Ian Sommerville

# Software Testing Strategies

(Source: Pressman, R. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, 2005)

Lecture 25

# Topics of Discussion

- A strategic approach to testing
- Test strategies for conventional software
- Test strategies for object-oriented software
- Validation testing
- System testing
- The art of debugging

# Introduction

- A strategy for software testing integrates the design of software test cases into a well-planned series of steps that result in successful development of the software
- The strategy provides a road map that describes the steps to be taken, when, and how much effort, time, and resources will be required
- The strategy incorporates test planning, test case design, test execution, and test result collection and evaluation
- The strategy provides guidance for the practitioner and a set of milestones for the manager
- Because of time pressures, progress must be measurable and problems must surface as early as possible

# A Strategic Approach to Testing

# General Characteristics of Strategic Testing

- To perform effective testing, a software team should conduct effective formal technical reviews
- Testing begins at the component level and work outward toward the integration of the entire computer-based system
- Different testing techniques are appropriate at different points in time
- Testing is conducted by the developer of the software and (for large projects) by an independent test group
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy

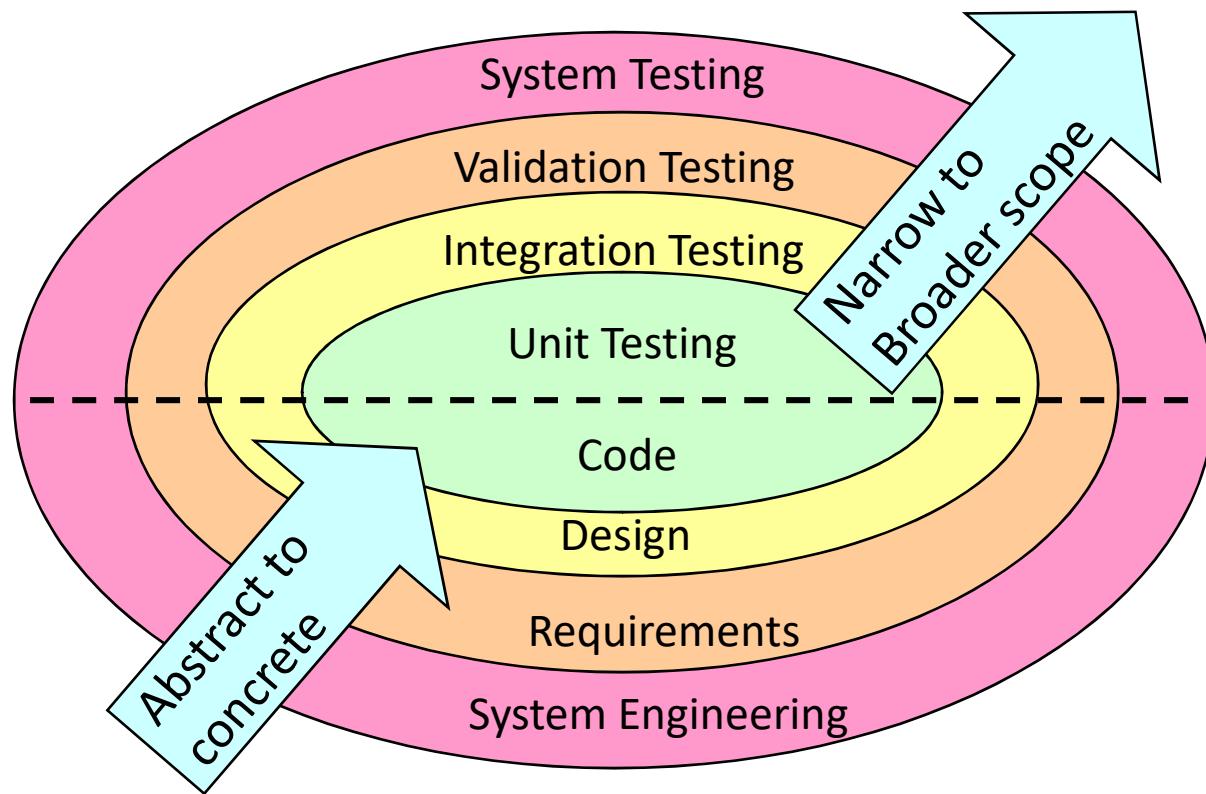
# Verification and Validation

- Software testing is part of a broader group of activities called verification and validation that are involved in software quality assurance
- Verification (Are the algorithms coded correctly?)
  - The set of activities that ensure that software correctly implements a specific function or algorithm
- Validation (Does it meet user requirements?)
  - The set of activities that ensure that the software that has been built is traceable to customer requirements

# Organizing for Software Testing

- Testing should aim at "breaking" the software
- Common misconceptions
  - The developer of software should do no testing at all
  - The software should be given to a secret team of testers who will test it unmercifully
  - The testers get involved with the project only when the testing steps are about to begin
- Reality: Independent test group
  - Removes the inherent problems associated with letting the builder test the software that has been built
  - Removes the conflict of interest that may otherwise be present
  - Works closely with the software developer during analysis and design to ensure that thorough testing occurs

# A Strategy for Testing Conventional Software



# Levels of Testing for Conventional Software

- Unit testing
  - Concentrates on each component/function of the software as implemented in the source code
- Integration testing
  - Focuses on the design and construction of the software architecture
- Validation testing
  - Requirements are validated against the constructed software
- System testing
  - The software and other system elements are tested as a whole

# Testing Strategy applied to Conventional Software

- Unit testing
  - Exercises specific paths in a component's control structure to ensure complete coverage and maximum error detection
  - Components are then assembled and integrated
- Integration testing
  - Focuses on inputs and outputs, and how well the components fit together and work together
- Validation testing
  - Provides final assurance that the software meets all functional, behavioral, and performance requirements
- System testing
  - Verifies that all system elements (software, hardware, people, databases) mesh properly and that overall system function and performance is achieved

# Testing Strategy applied to Object-Oriented Software

- Must broaden testing to include detections of errors in analysis and design models
- Unit testing loses some of its meaning and integration testing changes significantly
- Use the same philosophy but different approach as in conventional software testing
- Test "in the small" and then work out to testing "in the large"
  - Testing in the small involves class attributes and operations; the main focus is on communication and collaboration within the class
  - Testing in the large involves a series of regression tests to uncover errors due to communication and collaboration among classes
- Finally, the system as a whole is tested to detect errors in fulfilling requirements

# When is Testing Complete?

- There is no definitive answer to this question
- Every time a user executes the software, the program is being tested
- Sadly, testing usually stops when a project is running out of time, money, or both
- One approach is to divide the test results into various severity levels
  - Then consider testing to be complete when certain levels of errors no longer occur or have been repaired or eliminated

# Ensuring a Successful Software Test Strategy

- Specify product requirements in a quantifiable manner long before testing commences
- State testing objectives explicitly in measurable terms
- Understand the user of the software (through use cases) and develop a profile for each user category
- Develop a testing plan that emphasizes rapid cycle testing to get quick feedback to control quality levels and adjust the test strategy
- Build robust software that is designed to test itself and can diagnose certain kinds of errors
- Use effective formal technical reviews as a filter prior to testing to reduce the amount of testing required
- Conduct formal technical reviews to assess the test strategy and test cases themselves
- Develop a continuous improvement approach for the testing process through the gathering of metrics

# Test Strategies for Conventional Software

# Unit Testing

- Focuses testing on the function or software module
- Concentrates on the internal processing logic and data structures
- Is simplified when a module is designed with high cohesion
  - Reduces the number of test cases
  - Allows errors to be more easily predicted and uncovered
- Concentrates on critical modules and those with **high cyclomatic complexity** when testing resources are limited

# Targets for Unit Test Cases

- Module interface
  - Ensure that information flows properly into and out of the module
- Local data structures
  - Ensure that data stored temporarily maintains its integrity during all steps in an algorithm execution
- Boundary conditions
  - Ensure that the module operates properly at boundary values established to limit or restrict processing
- Independent paths (basis paths)
  - Paths are exercised to ensure that all statements in a module have been executed at least once
- Error handling paths
  - Ensure that the algorithms respond correctly to specific error conditions

# Common Computational Errors in Execution Paths

- Misunderstood or incorrect arithmetic precedence
- Mixed mode operations (e.g., int, float, char)
- Incorrect initialization of values
- Precision inaccuracy and round-off errors
- Incorrect symbolic representation of an expression (int vs. float)

# Other Errors to Uncover

- Comparison of different data types
- Incorrect logical operators or precedence
- Expectation of equality when precision error makes equality unlikely (using == with float types)
- Incorrect comparison of variables
- Improper or nonexistent loop termination
- Failure to exit when divergent iteration is encountered
- Improperly modified loop variables
- Boundary value violations

# Problems to uncover in Error Handling

- Error description is unintelligible or ambiguous
- Error noted does not correspond to error encountered
- Error condition causes operating system intervention prior to error handling
- Exception condition processing is incorrect
- Error description does not provide enough information to assist in the location of the cause of the error

# Drivers and Stubs for Unit Testing

- Driver
  - A simple main program that accepts test case data, passes such data to the component being tested, and prints the returned results
- Stubs
  - Serve to replace modules that are subordinate to (called by) the component to be tested
  - It uses the module's exact interface, may do minimal data manipulation, provides verification of entry, and returns control to the module undergoing testing
- Drivers and stubs both represent overhead
  - Both must be written but don't constitute part of the installed software product

# Integration Testing

- Defined as a systematic technique for constructing the software architecture
  - At the same time integration is occurring, conduct tests to uncover errors associated with interfaces
- Objective is to take unit tested modules and build a program structure based on the prescribed design
- Two Approaches
  - Non-incremental Integration Testing
  - Incremental Integration Testing

# Non-incremental Integration Testing

- Commonly called the “Big Bang” approach
- All components are combined in advance
- The entire program is tested as a whole
- Chaos results
- Many seemingly-unrelated errors are encountered
- Correction is difficult because isolation of causes is complicated
- Once a set of errors are corrected, more errors occur, and testing appears to enter an endless loop

# Incremental Integration Testing

- Three kinds
  - Top-down integration
  - Bottom-up integration
  - Sandwich integration
- The program is constructed and tested in small increments
- Errors are easier to isolate and correct
- Interfaces are more likely to be tested completely
- A systematic test approach is applied

# Top-down Integration

- Modules are integrated by moving downward through the control hierarchy, beginning with the main module
- Subordinate modules are incorporated in either a depth-first or breadth-first fashion
  - DF: All modules on a major control path are integrated
  - BF: All modules directly subordinate at each level are integrated
- Advantages
  - This approach verifies major control or decision points early in the test process
- Disadvantages
  - Stubs need to be created to substitute for modules that have not been built or tested yet; this code is later discarded
  - Because stubs are used to replace lower level modules, no significant data flow can occur until much later in the integration/testing process

# Bottom-up Integration

- Integration and testing starts with the most atomic modules in the control hierarchy
- Advantages
  - This approach verifies low-level data processing early in the testing process
  - Need for stubs is eliminated
- Disadvantages
  - Driver modules need to be built to test the lower-level modules; this code is later discarded or expanded into a full-featured version
  - Drivers inherently do not contain the complete algorithms that will eventually use the services of the lower-level modules; consequently, testing may be incomplete or more testing may be needed later when the upper level modules are available

# Sandwich Integration

- Consists of a combination of both top-down and bottom-up integration
- Occurs both at the highest level modules and also at the lowest level modules
- Proceeds using functional groups of modules, with each group completed before the next
  - High and low-level modules are grouped based on the control and data processing they provide for a specific program feature
  - Integration within the group progresses in alternating steps between the high and low level modules of the group
  - When integration for a certain functional group is complete, integration and testing moves onto the next group
- Reaps the advantages of both types of integration while minimizing the need for drivers and stubs
- Requires a disciplined approach so that integration doesn't tend towards the “big bang” scenario

# Regression Testing

- Each new addition or change to baselined software may cause problems with functions that previously worked flawlessly
- Regression testing re-executes a small subset of tests that have already been conducted
  - Ensures that changes have not propagated unintended side effects
  - Helps to ensure that changes do not introduce unintended behavior or additional errors
  - May be done manually or through the use of automated capture/playback tools
- Regression test suite contains three different classes of test cases
  - A representative sample of tests that will exercise all software functions
  - Additional tests that focus on software functions that are likely to be affected by the change
  - Tests that focus on the actual software components that have been changed

# Smoke Testing

- Taken from the world of hardware
  - Power is applied and a technician checks for sparks, smoke, or other dramatic signs of fundamental failure
- Designed as a pacing mechanism for time-critical projects
  - Allows the software team to assess its project on a frequent basis
- Includes the following activities
  - The software is compiled and linked into a build
  - A series of breadth tests is designed to expose errors that will keep the build from properly performing its function
    - The goal is to uncover “show stopper” errors that have the highest likelihood of throwing the software project behind schedule
  - The build is integrated with other builds and the entire product is smoke tested daily
    - Daily testing gives managers and practitioners a realistic assessment of the progress of the integration testing
  - After a smoke test is completed, detailed test scripts are executed

# Benefits of Smoke Testing

- Integration risk is minimized
  - Daily testing uncovers incompatibilities and show-stoppers early in the testing process, thereby reducing schedule impact
- The quality of the end-product is improved
  - Smoke testing is likely to uncover both functional errors and architectural and component-level design errors
- Error diagnosis and correction are simplified
  - Smoke testing will probably uncover errors in the newest components that were integrated
- Progress is easier to assess
  - As integration testing progresses, more software has been integrated and more has been demonstrated to work
  - Managers get a good indication that progress is being made

# Test Strategies for Object-Oriented Software

# Test Strategies for Object-Oriented Software

- With object-oriented software, you can no longer test a single operation in isolation (conventional thinking)
- Traditional top-down or bottom-up integration testing has little meaning
- Class testing for object-oriented software is the equivalent of unit testing for conventional software
  - Focuses on operations encapsulated by the class and the state behavior of the class
- Drivers can be used
  - To test operations at the lowest level and for testing whole groups of classes
  - To replace the user interface so that tests of system functionality can be conducted prior to implementation of the actual interface
- Stubs can be used
  - In situations in which collaboration between classes is required but one or more of the collaborating classes has not yet been fully implemented

# Test Strategies for Object-Oriented Software (continued)

- Two different object-oriented testing strategies
  - Thread-based testing
    - Integrates the set of classes required to respond to one input or event for the system
    - Each thread is integrated and tested individually
    - Regression testing is applied to ensure that no side effects occur
  - Use-based testing
    - First tests the independent classes that use very few, if any, server classes
    - Then the next layer of classes, called dependent classes, are integrated
    - This sequence of testing layer of dependent classes continues until the entire system is constructed

# Validation Testing

# Background

- Validation testing follows integration testing
- The distinction between conventional and object-oriented software disappears
- Focuses on user-visible actions and user-recognizable output from the system
- Demonstrates conformity with requirements
- Designed to ensure that
  - All functional requirements are satisfied
  - All behavioral characteristics are achieved
  - All performance requirements are attained
  - Documentation is correct
  - Usability and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability)
- After each validation test
  - The function or performance characteristic conforms to specification and is accepted
  - A deviation from specification is uncovered and a deficiency list is created
- A configuration review or audit ensures that all elements of the software configuration have been properly developed, cataloged, and have the necessary detail for entering the support phase of the software life cycle

# Alpha and Beta Testing

- Alpha testing
  - Conducted at the developer's site by end users
  - Software is used in a natural setting with developers watching intently
  - Testing is conducted in a controlled environment
- Beta testing
  - Conducted at end-user sites
  - Developer is generally not present
  - It serves as a live application of the software in an environment that cannot be controlled by the developer
  - The end-user records all problems that are encountered and reports these to the developers at regular intervals
- After beta testing is complete, software engineers make software modifications and prepare for release of the software product to the entire customer base

# System Testing

# Different Types

- Recovery testing
  - Tests for recovery from system faults
  - Forces the software to fail in a variety of ways and verifies that recovery is properly performed
  - Tests reinitialization, checkpointing mechanisms, data recovery, and restart for correctness
- Security testing
  - Verifies that protection mechanisms built into a system will, in fact, protect it from improper access
- Stress testing
  - Executes a system in a manner that demands resources in abnormal quantity, frequency, or volume
- Performance testing
  - Tests the run-time performance of software within the context of an integrated system
  - Often coupled with stress testing and usually requires both hardware and software instrumentation
  - Can uncover situations that lead to degradation and possible system failure

# The Art of Debugging

# Debugging Process

- Debugging occurs as a consequence of successful testing
- It is still very much an art rather than a science
- Good debugging ability may be an innate human trait
- Large variances in debugging ability exist
- The debugging process begins with the execution of a test case
- Results are assessed and the difference between expected and actual performance is encountered
- This difference is a symptom of an underlying cause that lies hidden
- The debugging process attempts to match symptom with cause, thereby leading to error correction

# Why is Debugging so Difficult?

- The symptom and the cause may be geographically remote
- The symptom may disappear (temporarily) when another error is corrected
- The symptom may actually be caused by nonerrors (e.g., round-off accuracies)
- The symptom may be caused by human error that is not easily traced

(continued on next slide)

# Why is Debugging so Difficult? (continued)

- The symptom may be a result of timing problems, rather than processing problems
- It may be difficult to accurately reproduce input conditions, such as asynchronous real-time information
- The symptom may be intermittent such as in embedded systems involving both hardware and software
- The symptom may be due to causes that are distributed across a number of tasks running on different processes

# Debugging Strategies

- Objective of debugging is to find and correct the cause of a software error
- Bugs are found by a combination of systematic evaluation, intuition, and luck
- Debugging methods and tools are not a substitute for careful evaluation based on a complete design model and clear source code
- There are three main debugging strategies
  - Brute force
  - Backtracking
  - Cause elimination

# Strategy #1: Brute Force

- Most commonly used and least efficient method
- Used when all else fails
- Involves the use of memory dumps, run-time traces, and output statements
- Leads many times to wasted effort and time

# Strategy #2: Backtracking

- Can be used successfully in small programs
- The method starts at the location where a symptom has been uncovered
- The source code is then traced backward (manually) until the location of the cause is found
- In large programs, the number of potential backward paths may become unmanageably large

# Strategy #3: Cause Elimination

- Involves the use of induction or deduction and introduces the concept of binary partitioning
  - Induction (specific to general): Prove that a specific starting value is true; then prove the general case is true
  - Deduction (general to specific): Show that a specific conclusion follows from a set of general premises
- Data related to the error occurrence are organized to isolate potential causes
- A cause hypothesis is devised, and the aforementioned data are used to prove or disprove the hypothesis
- Alternatively, a list of all possible causes is developed, and tests are conducted to eliminate each cause
- If initial tests indicate that a particular cause hypothesis shows promise, data are refined in an attempt to isolate the bug

# Three Questions to ask Before Correcting the Error

- Is the cause of the bug reproduced in another part of the program?
  - Similar errors may be occurring in other parts of the program
- What next bug might be introduced by the fix that I'm about to make?
  - The source code (and even the design) should be studied to assess the coupling of logic and data structures related to the fix
- What could we have done to prevent this bug in the first place?
  - This is the first step toward software quality assurance
  - By correcting the process as well as the product, the bug will be removed from the current program and may be eliminated from all future programs



# Reference

- Chapter 13 Pressman, R. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, 2005

# **Software Engineering**

## **Software Cost Estimation**

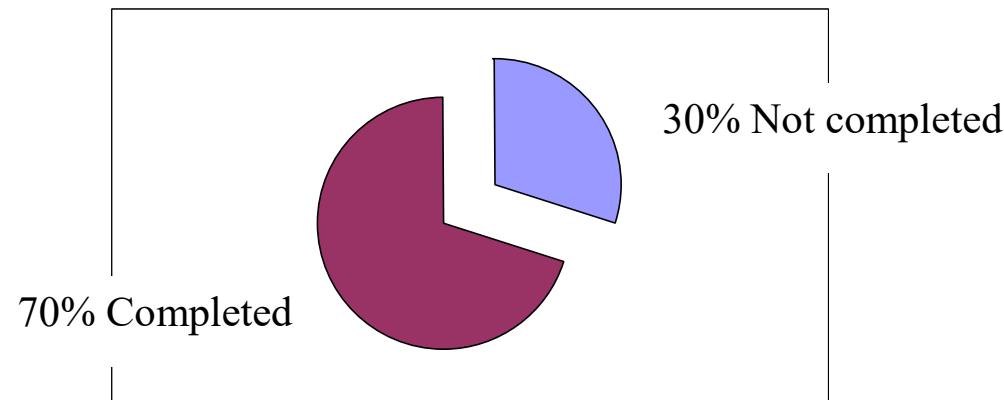
# Objectives

- To introduce the fundamentals of software costing and pricing
- To explain software productivity metric
- To explain why different techniques for software estimation:
  - LOC model
  - Function points model
  - Object point model
  - COCOMO (COnstructive COst MOdel): 2 algorithmic cost estimation model
  - UCP: Use Case Points

# What is Software Cost Estimation

- Predicting the cost of resources required for a software development process

# Software is a Risky Business



- 53% of projects cost almost 200% of original estimate.
- Estimated \$81 billion spent on failed U.S. projects in 1995.
  - All surveyed projects used waterfall lifecycle.

# Software is a Risky Business

- British Computer Society (BCS) survey:
  - 1027 projects
  - Only 130 were successful !
- Success was defined as:
  - deliver all system **requirements**
  - within **budget**
  - within **time**
  - to the **quality** agreed on

# Why **early** Cost Estimation?

- Cost estimation is needed **early** for s/w pricing
- S/W price = cost + profit

# Fundamental estimation questions

- **Effort**
  - How much effort is required to complete an activity?
  - Units: man-day (person-day), man-week, man-month,..
- **Duration**
  - How much **calendar time** is needed to complete an activity? Resources assigned
  - Units: hour, day, week, month, year,..
- **Cost of an activity**
  - What is the total cost of an activity?
- Project estimation and scheduling are interleaved management activities

# Software Cost Components

1. Effort costs (dominant factor in most projects)
  - salaries
  - Social and insurance & benefits
2. Tools costs: Hardware and software for development
  - Depreciation on relatively small # of years 300K US\$
3. Travel and Training costs (for particular client)
4. Overheads(OH): Costs must take overheads into account
  - costs of building, air-conditioning, heating, lighting
  - costs of networking and communications (tel, fax, )
  - costs of shared facilities (e.g library, staff restaurant, etc.)
  - depreciation costs of assets
  - Activity Based Costing (ABC)

# S/W Pricing Policy

S/W price is influenced by

- economic consideration
- political consideration
- and business consideration

# Software Pricing Policy/Factors

Factor	Description
Market opportunity	A development organisation may quote a low price because it wishes to move into a new segment of the software market. Accepting a low profit on one project may give the opportunity of more profit later. The experience gained may allow new products to be developed.
Cost estimate uncertainty	If an organisation is unsure of its cost estimate, it may increase its price by some contingency over and above its normal profit.
Contractual terms	A customer may be willing to allow the developer to retain ownership of the source code and reuse it in other projects. The price charged may then be less than if the software source code is handed over to the customer.
Requirements volatility	If the requirements are likely to change, an organisation may lower its price to win a contract. After the contract is awarded, high prices may be charged for changes to the requirements.
Financial health	Developers in financial difficulty may lower their price to gain a contract. It is better to make a small profit or break even than to go out of business.

# Programmer Productivity

- Rate of s/w production
  - Needs for measurements
  - Measure software produced per time unit (Ex: LOC/hr)
    - rate of s/w production
    - software produced including documentation
- Not quality-oriented: although quality assurance is a factor in productivity assessment

# Productivity measures

S/W productivity measures are based on:

- Size related measures:
  - Based on some output from the software process
  - Number lines of delivered source code (LOC)
- Function-related measures
  - based on an estimate of the functionality of the delivered software:
    - Function-points (are the best known of this type of measure)
    - Object-points
    - UCP

# Measurement problems

- Estimating the size of the measure
- Estimating the total number of programmer-months which have elapsed
- Estimating contractor productivity (e.g. documentation team) and incorporating this estimate in overall estimate

# Lines Of Code (LOC)

- Program length (LOC) can be used to predict program characteristics e.g. person-month effort and ease of maintenance
- What's a line of code?
  - The measure was first proposed when programs were typed on cards with one line per card
  - How does this correspond to statements as in Java which can span several lines or where there can be several statements on one line?
- What programs should be counted as part of the system?
- Assumes linear relationship between system size and volume of documentation

# Versions of LOC

- DSI : Delivered Source Instructions
- KLOC Thousands of LOC
- DSI
  - One instruction is one LOC
  - Declarations are counted
  - Comments are not counted

# LOC

- Advantages
  - Simple to measure
- Disadvantages
  - Defined on code: it can not measure the size of specification
  - Based on one specific view of size: length.. [What about complexity and functionality !!](#)
  - Bad s/w may yield more LOC
  - Language dependent
- Therefore: Other s/w size attributes must be included

# LOC Productivity

- The lower level the language, the less productive the programmer
  - The same functionality takes more code to implement in a lower-level language than in a high-level language
- Measures of productivity *based on LOC* suggest that programmers who write verbose code are more productive than programmers who write compact code !!!

# Function Points: FP

Function Points is used in 2 contexts:

- Past: To develop **metrics** from historical data
- Future: Use of available metrics to size the s/w of a **new** project

# Function Points

- Based on a combination of program characteristics
- The number of :
  - External (user) inputs: input transactions that update internal files
  - External (user) outputs: reports, error messages
  - User interactions: inquiries
  - **Logical** internal files used by the system:  
Example a purchase order logical file composed of 2 physical files/tables  
Purchase\_Order and Purchase\_Order\_Item
  - External interfaces: files shared with other systems
- A weight (ranging from **3 for simple** to **15 for complex** features) is associated with each of these above
- The function point count is computed by multiplying each raw count by the weight and summing all values

# Function Points - Calculation

<u>measurement parameter</u>	<u>count</u>	<u>weighting factor</u>			<u>=</u>	<input type="text"/>
		simple	avg.	complex		
number of user inputs	<input type="text"/>	X	3	4	6	<input type="text"/>
number of user outputs	<input type="text"/>	X	4	5	7	<input type="text"/>
number of user inquiries	<input type="text"/>	X	3	4	6	<input type="text"/>
number of files	<input type="text"/>	X	7	10	15	<input type="text"/>
number of ext.interfaces	<input type="text"/>	X	5	7	10	<input type="text"/>
count-total	<hr/>					<input type="text"/>
complexity multiplier	<hr/>					<input type="text"/>
function points	<hr/>					<input type="text"/>

# Function Points – Taking Complexity into Account -14 Factors Fi

Each factor is rated on a scale of:

**Zero:** not important or not applicable

**Five:** absolutely essential

1. Backup and recovery
2. Data communication
3. Distributed processing functions
4. Is performance critical?
5. Existing operating environment
6. On-line data entry
7. Input transaction built over multiple screens

## Function Points – Taking Complexity into Account -**14 Factors Fi** (cont.)

8. Master files updated on-line
9. Complexity of inputs, outputs, files, inquiries
10. Complexity of processing
11. Code design for re-use
12. Are conversion/installation included in design?
13. Multiple installations
14. Application designed to facilitate change by the user

## Function Points – Taking Complexity into Account -**14 Factors Fi** (cont.)

$$FP = UFC * [ 0.65 + 0.01 * \sum_{i=1}^{i=14} F_i ]$$

UFC: Unadjusted function point count

$0 \leq F \leq 5$

# FP: Advantages & Disadvantages

- Advantages
  - Available early .. We need only a detailed specification
  - Not restricted to code
  - Language independent
  - More accurate than LOC
- Disadvantages
  - Ignores quality issues of output
  - Subjective counting .. depend on the estimator
  - Hard to automate.. Automatic function-point counting is impossible

# Function points and LOC

- FPs can be used to estimate LOC depending on the average number of LOC per FP for a given language
  - $LOC = AVC * \text{number of function points}$
  - AVC is a language-dependent factor varying from approximately 300 for assemble language to 12-40 for a 4GL

# Relation Between FP & LOC

Programming Language	LOC/FP (average)
Assembly language	320
C	128
COBOL	106
FORTRAN	106
Pascal	90
C++	64
Ada	53
Visual Basic	32
Smalltalk	22
Power Builder (code generator)	16
SQL	12

# Function Points & Normalisation

- Function points are used to normalise measures (same as for LOC) for:
  - S/w productivity
  - Quality
- Error (bugs) per FP (discovered at programming)
- Defects per FP (discovered after programming)
- \$ per FP
- Pages of documentation per FP
- FP per person-month

# Expected Software Size

- Based on three-point
- Compute Expected Software Size ( $S$ ) as weighted average of:
  - Optimistic estimate:  $S(\text{opt})$
  - Most likely estimate:  $S(\text{ml})$
  - Pessimistic estimate:  $S(\text{pess})$
- $S = \{ S(\text{opt}) + 4 S(\text{ml}) + S(\text{pess}) \} / 6$ 
  - Beta probability distribution

# Example 1: LOC Approach

- A system is composed of 7 subsystems as below.
- Given for each subsystem the size in LOC and the 2 metrics: productivity LOC/pm (pm: person month) ,Cost \$/LOC
- Calculate the system total cost in \$ and effort in months .

Functions	estimated LOC	LOC/pm	\$/LOC
UICF	2340	315	14
2DGA	5380	220	20
3DGA	6800	220	20
DSM	3350	240	18
CGDF	4950	200	22
PCF	2140	140	28
DAM	8400	300	18

# Example 1: LOC Approach

Functions	estimated LOC	LOC/pm	\$/LOC	Cost	Effort (months)
UICF	2340	315	14	32,000	7.4
2DGA	5380	220	20	107,000	24.4
3DGA	6800	220	20	136,000	30.9
DSM	3350	240	18	60,000	13.9
CGDF	4950	200	22	109,000	24.7
PCF	2140	140	28	60,000	15.2
DAM	8400	300	18	151,000	28.0
Totals	33,360			655,000	145.0

# Example 2: LOC Approach

Assuming

- Estimated project LOC = 33200
- Organisational productivity (similar project type) = 620 LOC/p-m
- Burdened labour rate = 8000 \$/p-m

Then

- Effort =  $33200 / 620 = (53.6) = 54$  p-m
- Cost per LOC =  $8000 / 620 = (12.9) = 13$  \$/LOC
- Project total Cost =  $8000 * 54 = 432000$  \$

# Example 3: FP Approach

	A	B	C	D	E	F	G
1	Info Domain	Optimistic	Likely	Pessim.	Est Count	Weight	FP count
2	# of inputs	22	26	30	26	4	104
3	# of outputs	16	18	20	18	5	90
4	# of inquiries	16	21	26	21	4	84
5	# of files	4	5	6	5	10	50
6	# of external interfaces	1	2	3	2	7	14
7	<b>UFC: Unadjusted Function Count</b>						<b>342</b>
8			Complexity adjustment factor				1.17
9					FP		<b>400</b>

# Example 3: FP Approach (cont.)

## Complexity Factor

Complexity factor: Fi	value=0	value=1	value=2	value=3	value=4	value=5	Fi
Backup and recovery	0	0	0	0	1	0	4
Data communication	0	0	1	0	0	0	2
Distributed processing functions	0	0	0	0	0	0	0
Is performance critical?	0	0	0	0	1	0	4
Existing operating environment	0	0	0	1	0	0	3
On-line data entry	0	0	0	0	1	0	4
Input transaction built over multiple screens	0	0	0	0	0	1	5
Master files updated on-line	0	0	0	1	0	0	3
Complexity of inputs, outputs, files, inquiries	0	0	0	0	0	1	5
Complexity of processing	0	0	0	0	0	1	5
Code design for re-use	0	0	0	0	1	0	4
Are conversion/installation included in design?	0	0	0	1	0	0	3
Multiple installations	0	0	0	0	0	1	5
Application designed to facilitate change by the user	0	0	0	0	0	1	5
						Sigma (F)	52
Complexity adjustment factor	$0.65 + 0.01 * \text{Sigma (F)} =$				1.17		

## Example 3: FP Approach (cont.)

Assuming

$$\sum_i F_i = 52$$

$$FP = UFC * [ 0.65 + 0.01 * \sum_i F_i ]$$

$$FP = 342 * 1.17 = 400$$

Complexity adjustment factor = 1.17

# Example 4: FP Approach (cont.)

Assuming

- Estimated FP = 401
- Organisation average productivity (similar project type) = 6.5 FP/p-m (person-month)
- Burdened labour rate = 8000 \$/p-m

Then

- Estimated effort =  $401 / 6.5 = (61.65) = 62$  p-m
- Cost per FP =  $8000 / 6.5 = 1231$  \$/FP
- Project cost =  $8000 * 62 = 496000$  \$

# Object Points (for 4GLs)

- Object points are an alternative function-related measure to function points **when 4GLs** or similar languages are used for development
- Object points are **NOT** the same as object classes
- The number of object points in a program is a weighted estimate of
  - The number of separate **screens** that are displayed
  - The number of **reports** that are produced by the system
  - The number of **3GL modules** that must be developed to supplement the 4GL code
  - [C:\Software\\_Eng\Cocomo\Software Measurement Page, COCOMO II, object points.htm](C:\Software_Eng\Cocomo\Software Measurement Page, COCOMO II, object points.htm)

# Object Points – Weighting

Object Type	Simple	Medium	Difficult
Screen	1	2	3
Report	2	5	8
Each 3GL module	10	10	10

# Object Points – Weighting (cont.)

- ***srvr***: number of server **data tables** used with screen/report
- ***clnt***: number of client **data tables** used with screen/report

For Screens				For Reports			
Number of Views contained	# and source of data tables			Number of Sections contained	# and source of data tables		
	Total < 4 (< 2 srvr < 3 clnt)	Total < 8 (2/3 srvr 3-5 clnt)	Total 8+ (> 3 srvr > 5 clnt)		Total < 4 (< 2 srvr < 3 clnt)	Total < 8 (2/3 srvr 3-5 clnt)	Total 8+ (> 3 srvr > 5 clnt)
< 3	simple	simple	medium	0 or 1	simple	simple	medium
3 - 7	simple	medium	difficult	2 or 3	simple	medium	difficult
> 8	medium	difficult	difficult	4 +	medium	difficult	difficult

# Object Point Estimation

- Object points are **easier** to estimate from a specification than function points
  - simply concerned with screens, reports and 3GL modules
- At an **early** point in the development process:
  - Object points can be easily estimated
  - It is very difficult to estimate the number of lines of code in a system

# Productivity Estimates

- LOC productivity
  - Real-time embedded systems, 40-160 LOC/P-month
  - Systems programs , 150-400 LOC/P-month
  - Commercial applications, 200-800 LOC/P-month
- Object points productivity

Developer's experience and Capability / ICASE maturity and capability	Very low	Low	Nominal	High	Very high
<b>PROD: Productivity Object-point per person-month</b>	4	7	13	25	50

# Object Point Effort Estimation

- Effort in p-m = NOP / PROD
  - NOP = number of OP of the system
  - Example: An application contains 840 OP (NOP=840) & Productivity is very high (= 50)
  - Then, Effort =  $840/50 = (16.8) = 17$  p-m

# Adjustment for % of Reuse

- Adjusted NOP = NOP \* (1 - % reuse / 100)
- Example:
  - An application contains 840 OP, of which 20% can be supplied by existing components.

$$\text{Adjusted NOP} = 840 * (1 - 20/100) = 672 \text{ OP}$$

$$\text{Adjusted effort} = 672/50 = (13.4) = 14 \text{ p-m}$$

# Factors affecting productivity

Factor	Description
Application domain experience	Knowledge of the application domain is essential for effective software development. Engineers who already understand a domain are likely to be the most productive.
Process quality	The development process used can have a significant effect on productivity. This is covered in Chapter 31.
Project size	The larger a project, the more time required for team communications. Less time is available for development so individual productivity is reduced.
Technology support	Good support technology such as CASE tools, supportive configuration management systems, etc. can improve productivity.
Working environment	As discussed in Chapter 28, a quiet working environment with private work areas contributes to improved productivity.

# Quality and Productivity

- All metrics based on **volume/unit time** are flawed because they do not take quality into account
- Productivity may generally be increased at the cost of quality
- If change is constant, then an approach based on *counting lines of code (LOC)* is not meaningful

# Estimation techniques

- There is no simple way to make an accurate estimate of the effort required to develop a software system:
  - Initial estimates may be based on inadequate information in a user requirements definition
  - The software may run on unfamiliar computers or use new technology
  - The people in the project may be unknown
- Project cost estimates may be self-fulfilling
  - The estimate defines the budget and the product is adjusted to meet the budget

# **Software Cost Estimation**

## **Part 2**

Lecture 28

# Function points and LOC

- FPs can be used to estimate LOC depending on the average number of LOC per FP for a given language
  - $\text{LOC} = \text{AVC} * \text{number of function points}$
  - AVC is a language-dependent factor varying from approximately 300 for assemble language to 12-40 for a 4GL

# Relation Between FP & LOC

Programming Language	LOC/FP (average)
Assembly language	320
C	128
COBOL	106
FORTRAN	106
Pascal	90
C++	64
Ada	53
Visual Basic	32
Smalltalk	22
Power Builder (code generator)	16
SQL	12

# Function Points & Normalisation

- Function points are used to normalise measures (same as for LOC) for:
  - S/w productivity
  - Quality
- Error (bugs) per FP (discovered at programming)
- Defects per FP (discovered after programming)
- \$ per FP
- Pages of documentation per FP
- FP per person-month

# Expected Software Size

- Based on three-point
- Compute Expected Software Size ( $S$ ) as weighted average of:
  - Optimistic estimate:  $S(\text{opt})$
  - Most likely estimate:  $S(\text{ml})$
  - Pessimistic estimate:  $S(\text{pess})$
- $S = \{ S(\text{opt}) + 4 S(\text{ml}) + S(\text{pess}) \} / 6$ 
  - Beta probability distribution

# Example 1: LOC Approach

- A system is composed of 7 subsystems as below.
- Given for each subsystem the size in LOC and the 2 metrics: productivity LOC/pm (pm: person month) ,Cost \$/LOC
- Calculate the system total cost in \$ and effort in months .

Functions	estimated LOC	LOC/pm	\$/LOC
UICF	2340	315	14
2DGA	5380	220	20
3DGA	6800	220	20
DSM	3350	240	18
CGDF	4950	200	22
PCF	2140	140	28
DAM	8400	300	18

# Example 1: LOC Approach

Functions	estimated LOC	LOC/pm	\$/LOC	Cost	Effort (months)
UICF	2340	315	14	32,000	7.4
2DGA	5380	220	20	107,000	24.4
3DGA	6800	220	20	136,000	30.9
DSM	3350	240	18	60,000	13.9
CGDF	4950	200	22	109,000	24.7
PCF	2140	140	28	60,000	15.2
DAM	8400	300	18	151,000	28.0
Totals	33,360			655,000	145.0

# Example 2: LOC Approach

Assuming

- Estimated project LOC = 33200
- Organisational productivity (similar project type) = 620 LOC/p-m
- Burdened labour rate = 8000 \$/p-m

Then

- Effort =  $33200 / 620 = (53.6) = 54$  p-m
- Cost per LOC =  $8000 / 620 = (12.9) = 13$  \$/LOC
- Project total Cost =  $8000 * 54 = 432000$  \$

# Example 3: FP Approach

	A	B	C	D	E	F	G
1	Info Domain	Optimistic	Likely	Pessim.	Est Count	Weight	FP count
2	# of inputs	22	26	30	26	4	104
3	# of outputs	16	18	20	18	5	90
4	# of inquiries	16	21	26	21	4	84
5	# of files	4	5	6	5	10	50
6	# of external interfaces	1	2	3	2	7	14
7	<b>UFC: Unadjusted Function Count</b>						<b>342</b>
8			Complexity adjustment factor				1.17
9					FP		<b>400</b>

# Example 3: FP Approach (cont.)

## Complexity Factor

Complexity factor: Fi	value=0	value=1	value=2	value=3	value=4	value=5	Fi
Backup and recovery	0	0	0	0	1	0	4
Data communication	0	0	1	0	0	0	2
Distributed processing functions	0	0	0	0	0	0	0
Is performance critical?	0	0	0	0	1	0	4
Existing operating environment	0	0	0	1	0	0	3
On-line data entry	0	0	0	0	1	0	4
Input transaction built over multiple screens	0	0	0	0	0	1	5
Master files updated on-line	0	0	0	1	0	0	3
Complexity of inputs, outputs, files, inquiries	0	0	0	0	0	1	5
Complexity of processing	0	0	0	0	0	1	5
Code design for re-use	0	0	0	0	1	0	4
Are conversion/installation included in design?	0	0	0	1	0	0	3
Multiple installations	0	0	0	0	0	1	5
Application designed to facilitate change by the user	0	0	0	0	0	1	5
						Sigma (F)	52
Complexity adjustment factor	$0.65 + 0.01 * \text{Sigma (F)} =$				1.17		

## Example 3: FP Approach (cont.)

Assuming

$$\sum_i F_i = 52$$

$$FP = UFC * [ 0.65 + 0.01 * \sum_i F_i ]$$

$$FP = 342 * 1.17 = 400$$

Complexity adjustment factor = 1.17

# Example 4: FP Approach (cont.)

Assuming

- Estimated FP = 401
- Organisation average productivity (similar project type) = 6.5 FP/p-m (person-month)
- Burdened labour rate = 8000 \$/p-m

Then

- Estimated effort =  $401 / 6.5 = (61.65) = 62$  p-m
- Cost per FP =  $8000 / 6.5 = 1231$  \$/FP
- Project cost =  $8000 * 62 = 496000$  \$

# Object Points (for 4GLs)

- Object points are an alternative function-related measure to function points **when 4GLs** or similar languages are used for development
- Object points are **NOT** the same as object classes
- The number of object points in a program is a weighted estimate of
  - The number of separate **screens** that are displayed
  - The number of **reports** that are produced by the system
  - The number of **3GL modules** that must be developed to supplement the 4GL code
  - [C:\Software\\_Eng\Cocomo\Software Measurement Page, COCOMO II, object points.htm](C:\Software_Eng\Cocomo\Software Measurement Page, COCOMO II, object points.htm)

# Object Points – Weighting

Object Type	Simple	Medium	Difficult
Screen	1	2	3
Report	2	5	8
Each 3GL module	10	10	10

# Object Points – Weighting (cont.)

- ***srvr***: number of server **data tables** used with screen/report
- ***clnt***: number of client **data tables** used with screen/report

For Screens				For Reports			
Number of Views contained	# and source of data tables			Number of Sections contained	# and source of data tables		
	Total < 4 (< 2 srvr < 3 clnt)	Total < 8 (2/3 srvr 3-5 clnt)	Total 8+ (> 3 srvr > 5 clnt)		Total < 4 (< 2 srvr < 3 clnt)	Total < 8 (2/3 srvr 3-5 clnt)	Total 8+ (> 3 srvr > 5 clnt)
< 3	simple	simple	medium	0 or 1	simple	simple	medium
3 - 7	simple	medium	difficult	2 or 3	simple	medium	difficult
> 8	medium	difficult	difficult	4 +	medium	difficult	difficult

# Object Point Estimation

- Object points are **easier** to estimate from a specification than function points
  - simply concerned with screens, reports and 3GL modules
- At an **early** point in the development process:
  - Object points can be easily estimated
  - It is very difficult to estimate the number of lines of code in a system

# Productivity Estimates

- LOC productivity
  - Real-time embedded systems, 40-160 LOC/P-month
  - Systems programs , 150-400 LOC/P-month
  - Commercial applications, 200-800 LOC/P-month
- Object points productivity

Developer's experience and Capability / ICASE maturity and capability	Very low	Low	Nominal	High	Very high
<b>PROD: Productivity Object-point per person-month</b>	4	7	13	25	50

# Object Point Effort Estimation

- Effort in p-m = NOP / PROD
  - NOP = number of OP of the system
  - Example: An application contains 840 OP (NOP=840) & Productivity is very high (= 50)
  - Then, Effort =  $840/50 = (16.8) = 17$  p-m

# Adjustment for % of Reuse

- Adjusted NOP = NOP \* (1 - % reuse / 100)
- Example:
  - An application contains 840 OP, of which 20% can be supplied by existing components.

$$\text{Adjusted NOP} = 840 * (1 - 20/100) = 672 \text{ OP}$$

$$\text{Adjusted effort} = 672/50 = (13.4) = 14 \text{ p-m}$$

# Factors affecting productivity

Factor	Description
Application domain experience	Knowledge of the application domain is essential for effective software development. Engineers who already understand a domain are likely to be the most productive.
Process quality	The development process used can have a significant effect on productivity. This is covered in Chapter 31.
Project size	The larger a project, the more time required for team communications. Less time is available for development so individual productivity is reduced.
Technology support	Good support technology such as CASE tools, supportive configuration management systems, etc. can improve productivity.
Working environment	As discussed in Chapter 28, a quiet working environment with private work areas contributes to improved productivity.

# Quality and Productivity

- All metrics based on **volume/unit time** are flawed because they do not take quality into account
- Productivity may generally be increased at the cost of quality
- If change is constant, then an approach based on *counting lines of code (LOC)* is not meaningful

# Estimation techniques

- There is no simple way to make an accurate estimate of the effort required to develop a software system:
  - Initial estimates may be based on inadequate information in a user requirements definition
  - The software may run on unfamiliar computers or use new technology
  - The people in the project may be unknown
- Project cost estimates may be self-fulfilling
  - The estimate defines the budget and the product is adjusted to meet the budget

# Estimation techniques

- Algorithmic cost modelling
- Expert judgement
- Estimation by analogy
- Parkinson's Law
- Pricing to win

# Algorithmic code modelling

- A formula – empirical relation:
  - based on historical cost information and which is generally based on the size of the software
- The formulae used in a formal model arise from the analysis of historical data.

# Expert Judgement

- One or more experts in both software development and the **application domain** use their experience to predict software costs. Process iterates until some consensus is reached.
- Advantages: Relatively cheap estimation method. Can be accurate if experts have direct experience of similar systems
- Disadvantages: Very inaccurate if there are no experts!

# Estimation by Analogy

- Experience-based Estimates
- The cost of a project is computed by comparing the project to a **similar** project in the **same** application domain
- Advantages: Accurate if project data available
- Disadvantages: Impossible if no comparable project has been tackled. Needs systematically maintained cost database

# Estimation by Analogy : Problems

- However, new methods and technologies may make estimating based on experience inaccurate:
  - Object oriented rather than function-oriented development
  - Client-server systems rather than mainframe systems
  - Off the shelf components
  - Component-based software engineering
  - CASE tools and program generators

# Parkinson's Law

- “The project costs whatever resources are available”  
*(Resources are defined by the software house)*
- Advantages: No overspend
- Disadvantages: System is usually **unfinished**
  - The work is contracted to fit the budget available: by reducing functionality, quality

# Pricing to Win

- The project costs whatever the **customer budget is.**
- Advantages: You get the contract
- Disadvantages:
  - The probability that the customer gets the system he/she wants is small.
  - Costs do not accurately reflect the work required

# Pricing to Win

- This approach may seem unethical and unbusiness like
- However, when detailed information is lacking it may be the only appropriate strategy
- The project cost is agreed on the basis of an **outline** proposal and the development is **constrained by that cost**
- A detailed specification may be negotiated or an evolutionary approach used for system development

# Top-down and Bottom-up Estimation

- Top-down
  - Start at the system level and assess the overall system functionality
- Bottom-up
  - Start at the component level and estimate the effort required for each component. Add these efforts to reach a final estimate

# Top-down Estimation

- Usable *without* knowledge of the system architecture and the components that might be part of the system
- Takes into account costs such as integration, configuration management and documentation
- Can underestimate the cost of solving difficult low-level technical problems

# Bottom-up estimation

- Usable when
  - the architecture of the system is known and
  - components identified
- Accurate method if the system has been designed in detail
- May underestimate costs of system level activities such as integration and documentation

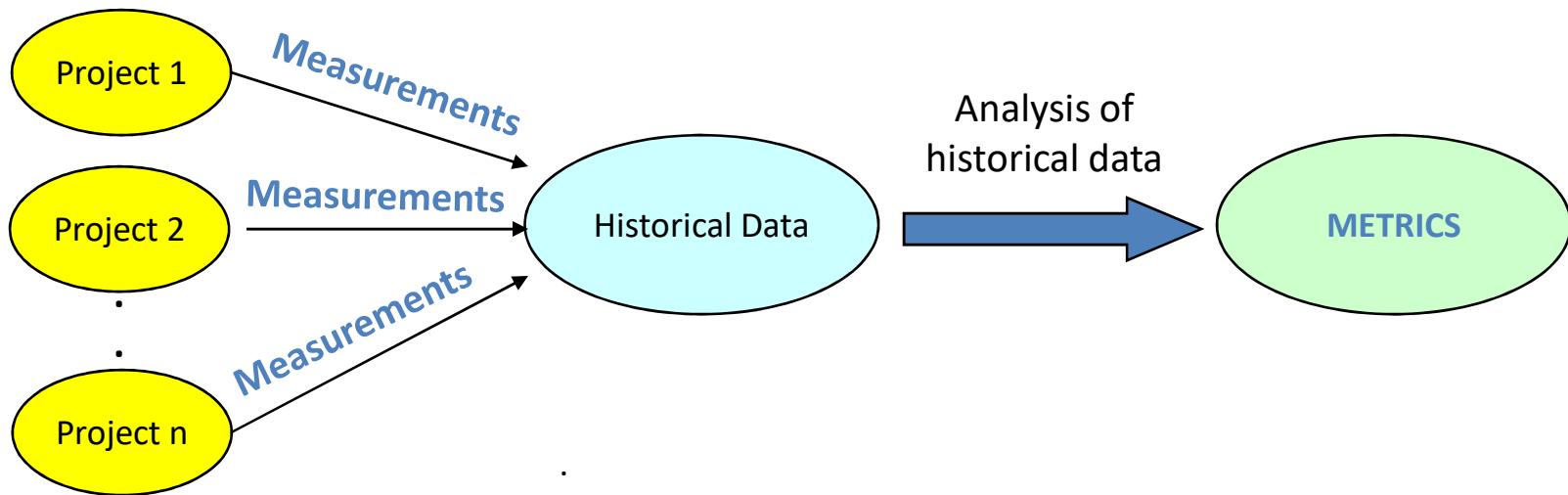
# Estimation Methods

- S/W project estimation should be based on several methods
- If these do not return approximately the same result, there is insufficient information available
- Some action should be taken to find out more in order to make more accurate estimates
- Pricing to win is sometimes the only applicable method

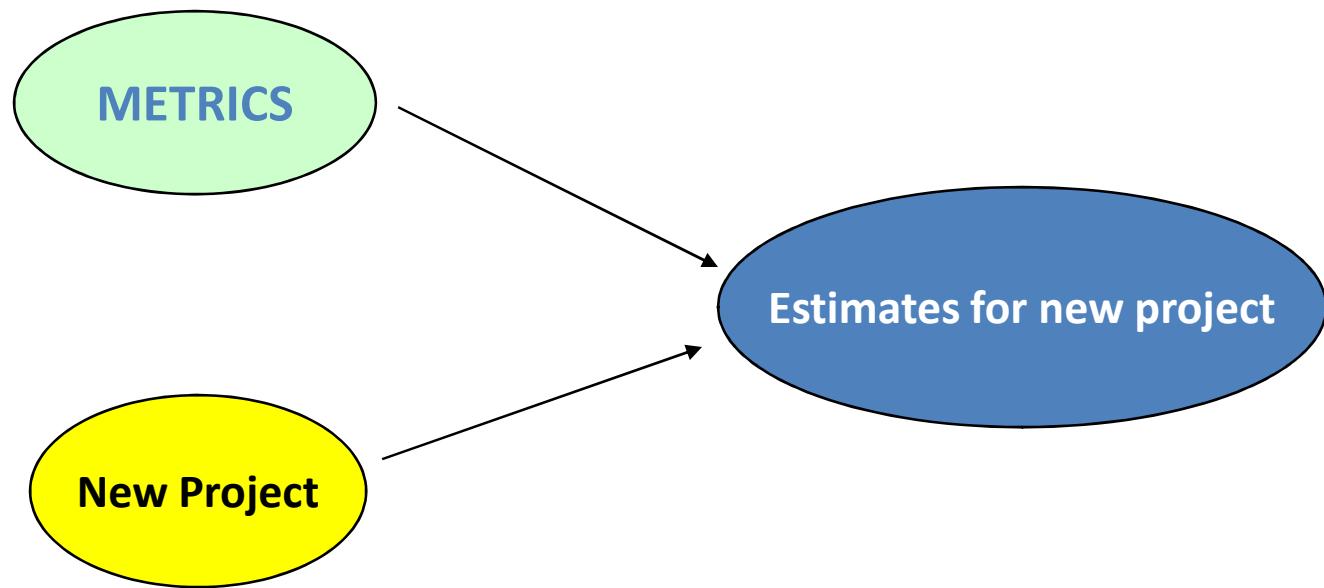
# Algorithmic Cost Modelling

- Most of the work in the cost estimation field has focused on algorithmic cost modelling.
- Costs are analysed using mathematical formulas linking costs or inputs with METRICS to produce an estimated output.
- The formula is based on the analysis of historical data.
- The accuracy of the model can be improved by calibrating the model to your specific development environment, (which basically involves adjusting the weighting parameters of the metrics).

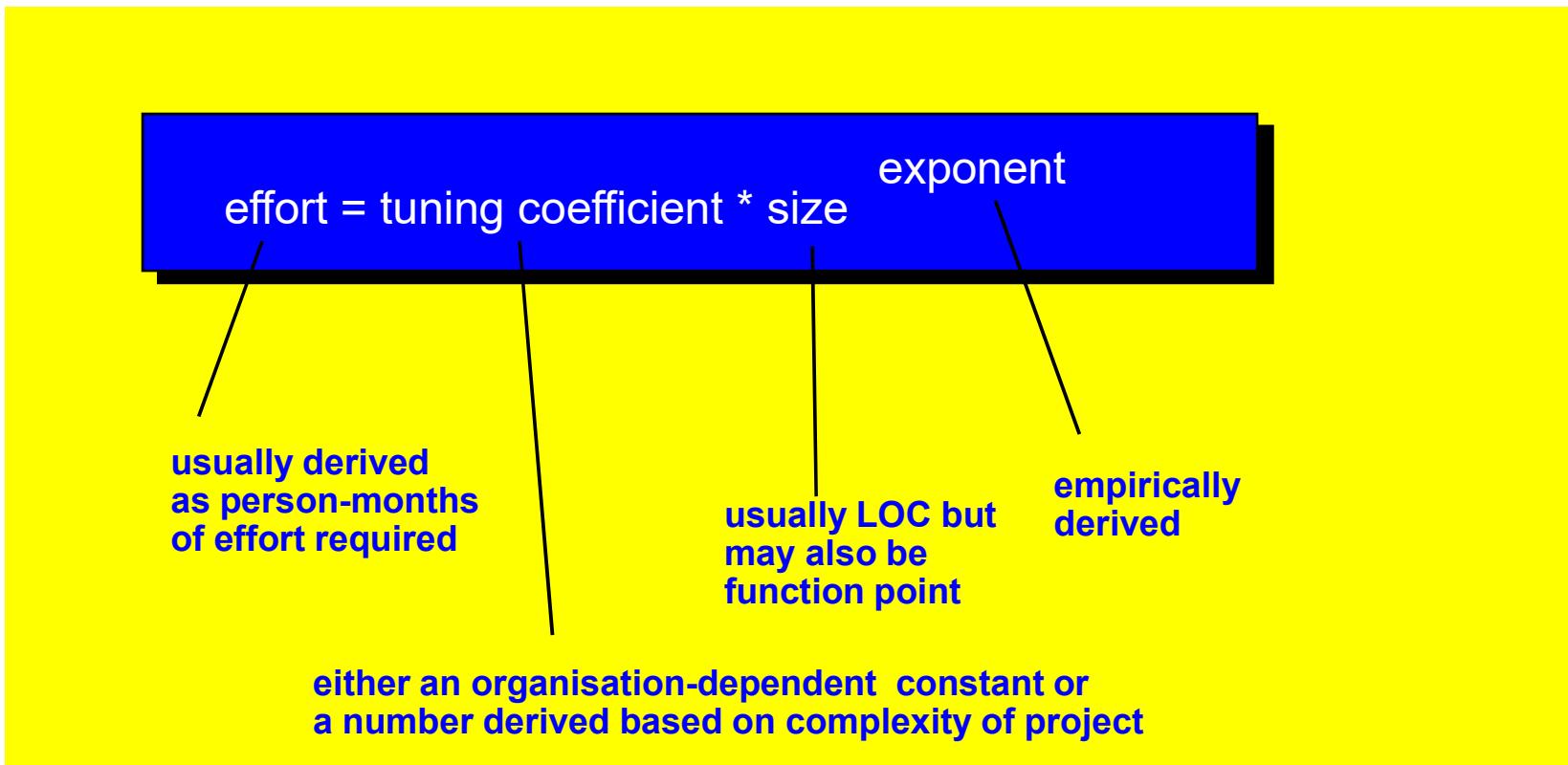
# Building Metrics from measurements



# New Project estimation using available Metrics



# Empirical Estimation Models - Algorithmic Cost Modelling



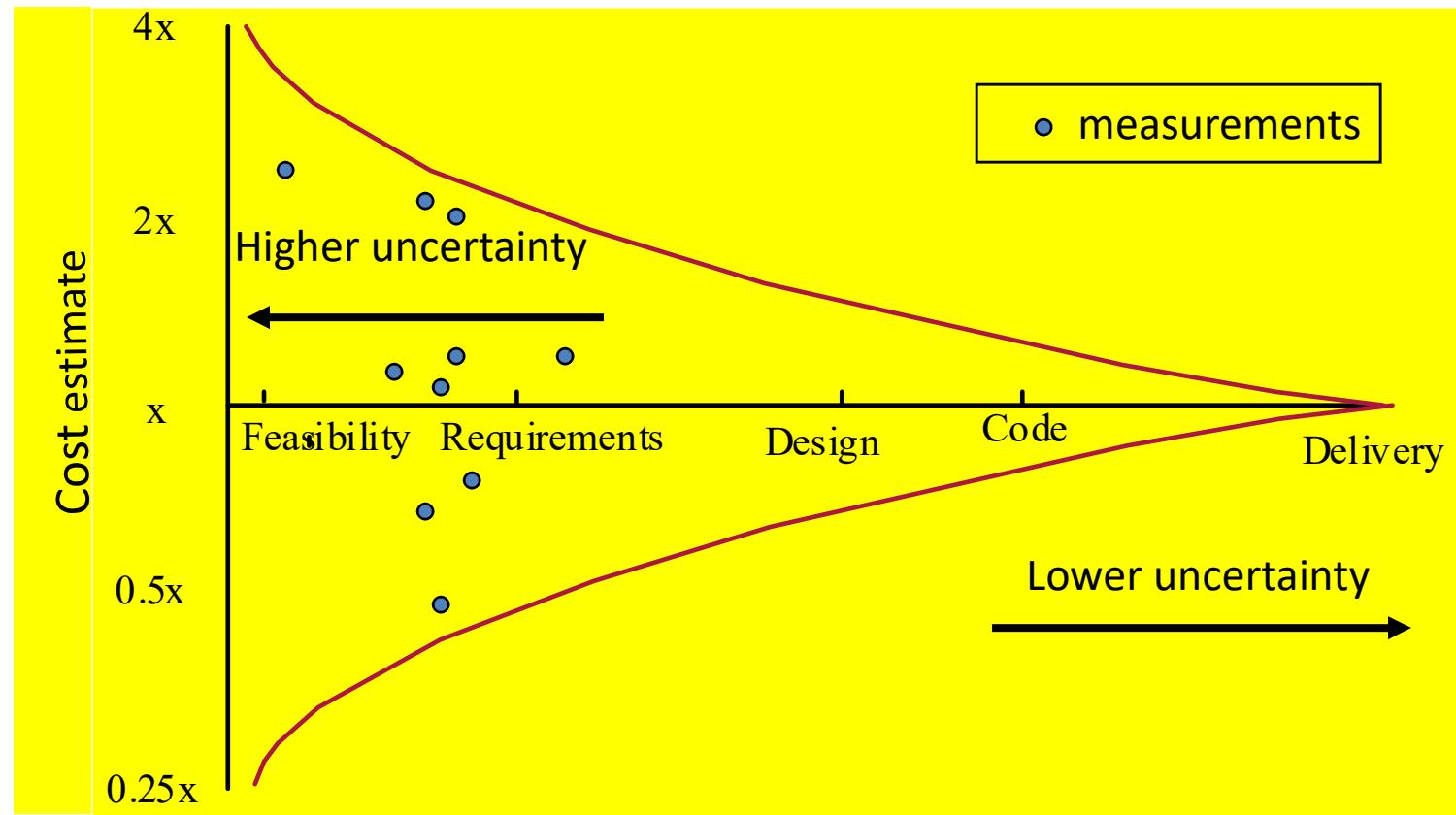
# Algorithmic Cost Modelling

- $\text{Effort} = A \times \text{Size}^B \times M$
- A is an **organisation-dependent constant**
- B reflects the **nonlinearity** (disproportionate) effort for large projects
- M is a multiplier reflecting product, process and people attributes
- Most commonly used product attribute for cost estimation is code size (LOC)
- Most models are basically similar but with different values for A, B and M

# Estimation Accuracy

- The size of a software system can only be known accurately when it is finished
- Several factors influence the final size
  - Use of COTS and components
  - Programming language
  - Distribution of system
- As the development process progresses then the size estimate becomes more accurate

# Estimate Uncertainty



# Configuration Management

Lecture 29

# Topics covered

- Version management
- System building
- Change management
- Release management

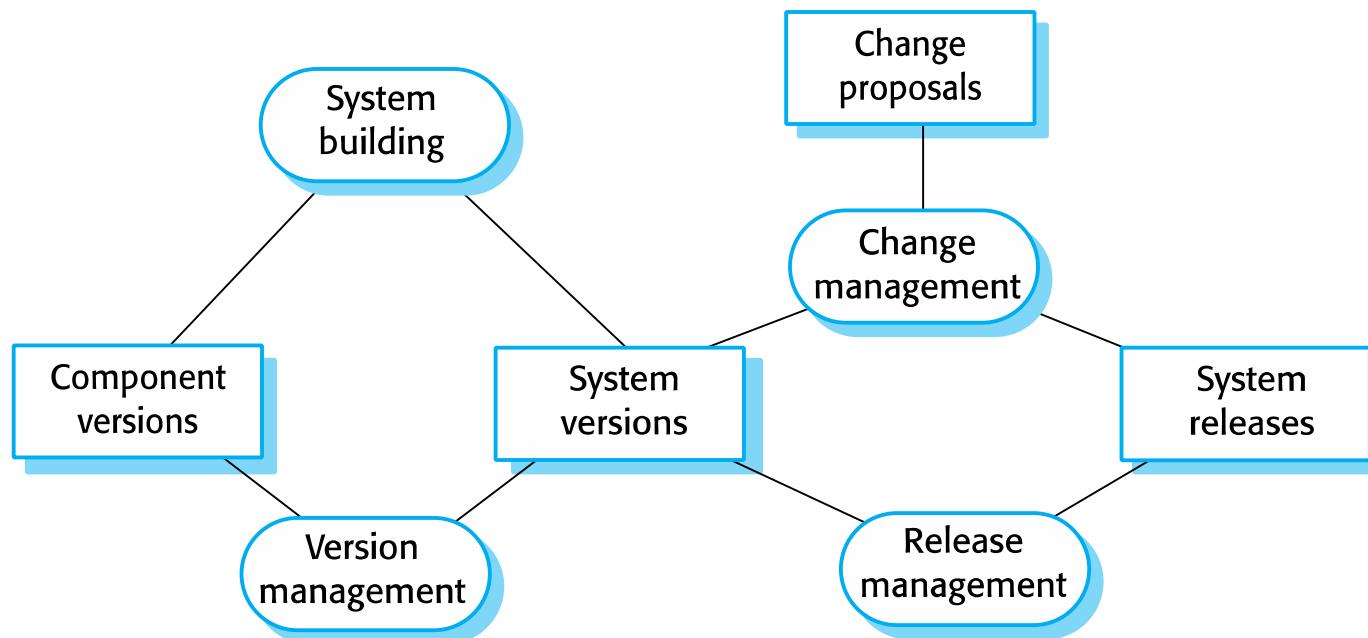
# Configuration management

- Software systems are constantly changing during development and use.
- Configuration management (CM) is concerned with the policies, processes and tools for managing changing software systems.
- You need CM because it is easy to lose track of what changes and component versions have been incorporated into each system version.
- CM is essential for team projects to control changes made by different developers

# CM activities

- Version management
  - Keeping track of the multiple versions of system components and ensuring that changes made to components by different developers do not interfere with each other.
- System building
  - The process of assembling program components, data and libraries, then compiling these to create an executable system.
- Change management
  - Keeping track of requests for changes to the software from customers and developers, working out the costs and impact of changes, and deciding the changes should be implemented.
- Release management
  - Preparing software for external release and keeping track of the system versions that have been released for customer use.

# Configuration management activities



# Agile development and CM

- Agile development, where components and systems are changed several times per day, is impossible without using CM tools.
- The definitive versions of components are held in a shared project repository and developers copy these into their own workspace.
- They make changes to the code then use system building tools to create a new system on their own computer for testing. Once they are happy with the changes made, they return the modified components to the project repository.

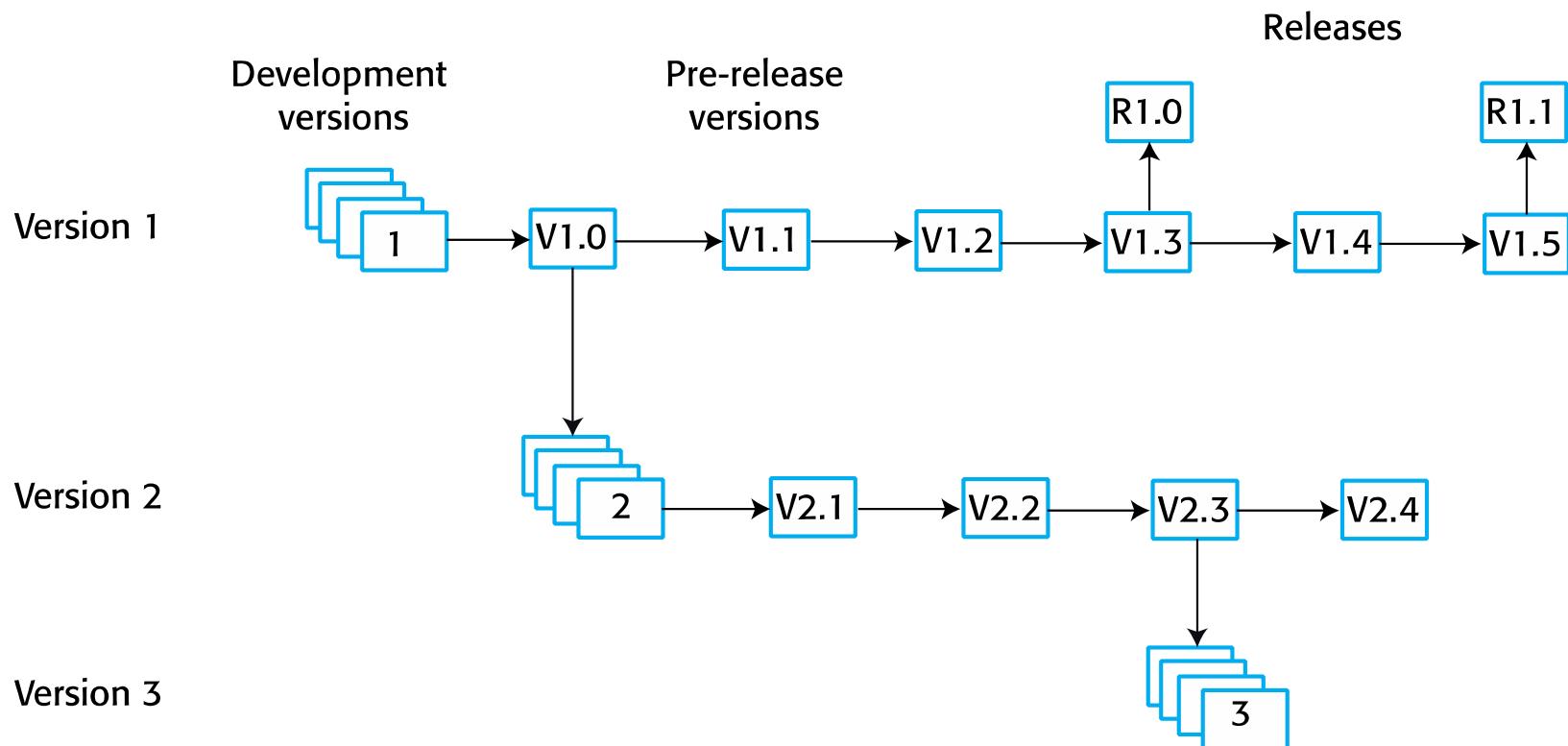
# Development phases

- A development phase where the development team is responsible for managing the software configuration and new functionality is being added to the software.
- A system testing phase where a version of the system is released internally for testing.
  - No new system functionality is added. Changes made are bug fixes, performance improvements and security vulnerability repairs.
- A release phase where the software is released to customers for use.
  - New versions of the released system are developed to repair bugs and vulnerabilities and to include new features.

# Multi-version systems

- For large systems, there is never just one ‘working’ version of a system.
- There are always several versions of the system at different stages of development.
- There may be several teams involved in the development of different system versions.

# Multi-version system development



# CM terminology

Term	Explanation
Baseline	A baseline is a collection of component versions that make up a system. Baselines are controlled, which means that the versions of the components making up the system cannot be changed. This means that it is always possible to recreate a baseline from its constituent components.
Branching	The creation of a new codeline from a version in an existing codeline. The new codeline and the existing codeline may then develop independently.
Codeline	A codeline is a set of versions of a software component and other configuration items on which that component depends.
Configuration (version) control	The process of ensuring that versions of systems and components are recorded and maintained so that changes are managed and all versions of components are identified and stored for the lifetime of the system.
Configuration item or software configuration item (SCI)	Anything associated with a software project (design, code, test data, document, etc.) that has been placed under configuration control. There are often different versions of a configuration item. Configuration items have a unique name.
Mainline	A sequence of baselines representing different versions of a system.

# CM terminology

Term	Explanation
Merging	The creation of a new version of a software component by merging separate versions in different codelines. These codelines may have been created by a previous branch of one of the codelines involved.
Release	A version of a system that has been released to customers (or other users in an organization) for use.
Repository	A shared database of versions of software components and meta-information about changes to these components.
System building	The creation of an executable system version by compiling and linking the appropriate versions of the components and libraries making up the system.
Version	An instance of a configuration item that differs, in some way, from other instances of that item. Versions always have a unique identifier.
Workspace	A private work area where software can be modified without affecting other developers who may be using or modifying that software.

# Version management

# Version management

- Version management (VM) is the process of keeping track of different versions of software components or configuration items and the systems in which these components are used.
- It also involves ensuring that changes made by different developers to these versions do not interfere with each other.
- Therefore version management can be thought of as the process of managing codelines and baselines.

# Codelines and baselines

- A codeline is a sequence of versions of source code with later versions in the sequence derived from earlier versions.
- Codelines normally apply to components of systems so that there are different versions of each component.
- A baseline is a definition of a specific system.
- The baseline therefore specifies the component versions that are included in the system plus a specification of the libraries used, configuration files, etc.

# Baselines

- Baselines may be specified using a configuration language, which allows you to define what components are included in a version of a particular system.
- Baselines are important because you often have to recreate a specific version of a complete system.
  - For example, a product line may be instantiated so that there are individual system versions for different customers. You may have to recreate the version delivered to a specific customer if, for example, that customer reports bugs in their system that have to be repaired.

# Codelines and baselines

Codeline (A)



Codeline (B)



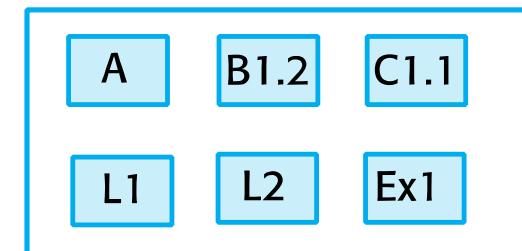
Codeline (C)



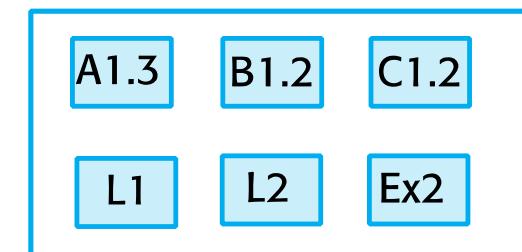
Libraries and external components



Baseline - V1



Baseline - V2



Mainline

# Version control systems

- Version control (VC) systems identify, store and control access to the different versions of components. There are two types of modern version control system
  - Centralized systems, where there is a single master repository that maintains all versions of the software components that are being developed. Subversion is a widely used example of a centralized VC system.
  - Distributed systems, where multiple versions of the component repository exist at the same time. Git is a widely-used example of a distributed VC system.

# Key features of version control systems

- Version and release identification
- Change history recording
- Support for independent development
- Project support
- Storage management

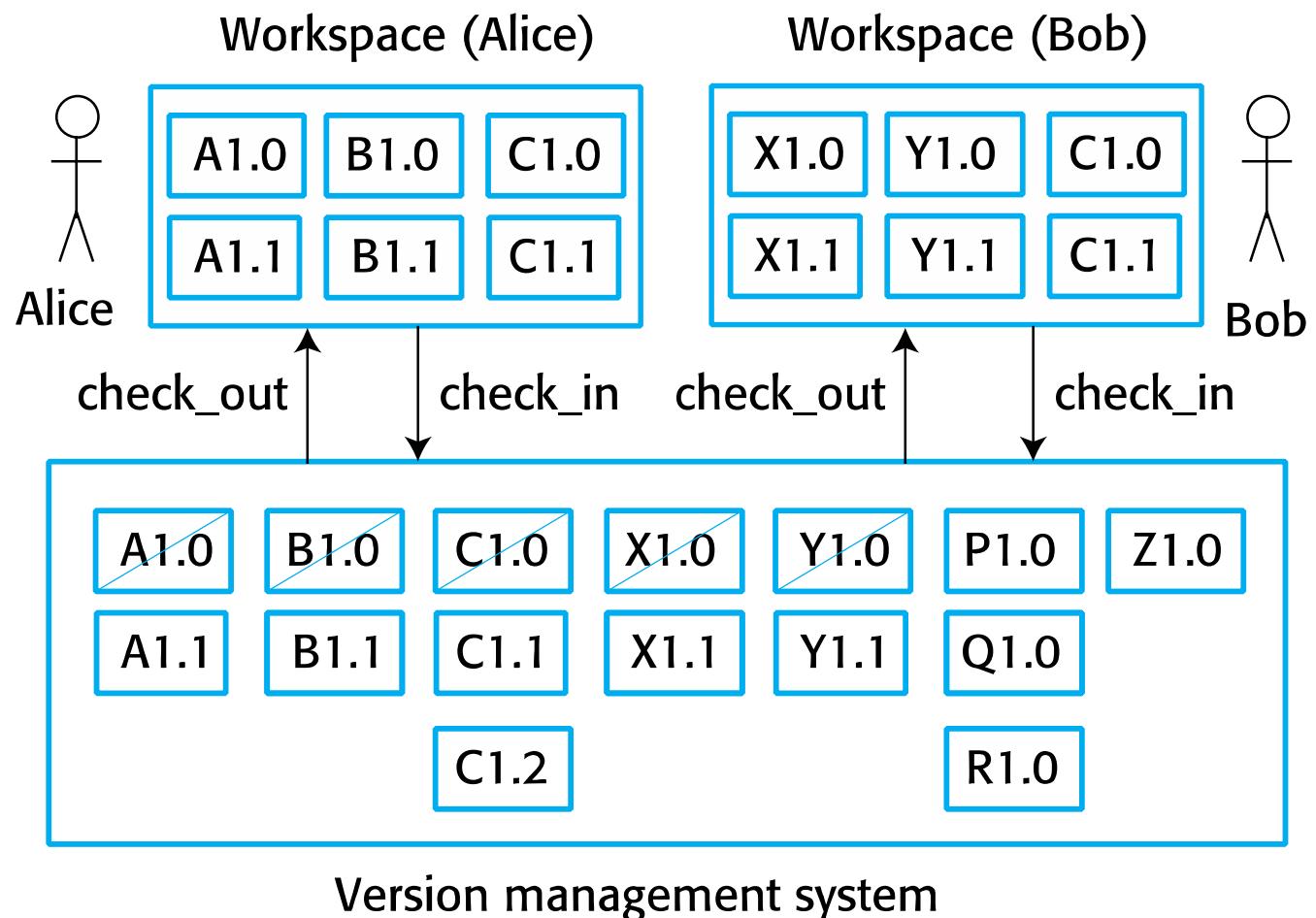
# Public repository and private workspaces

- To support independent development without interference, version control systems use the concept of a project repository and a private workspace.
- The project repository maintains the ‘master’ version of all components. It is used to create baselines for system building.
- When modifying components, developers copy (check-out) these from the repository into their workspace and work on these copies.
- When they have finished their changes, the changed components are returned (checked-in) to the repository.

# Centralized version control

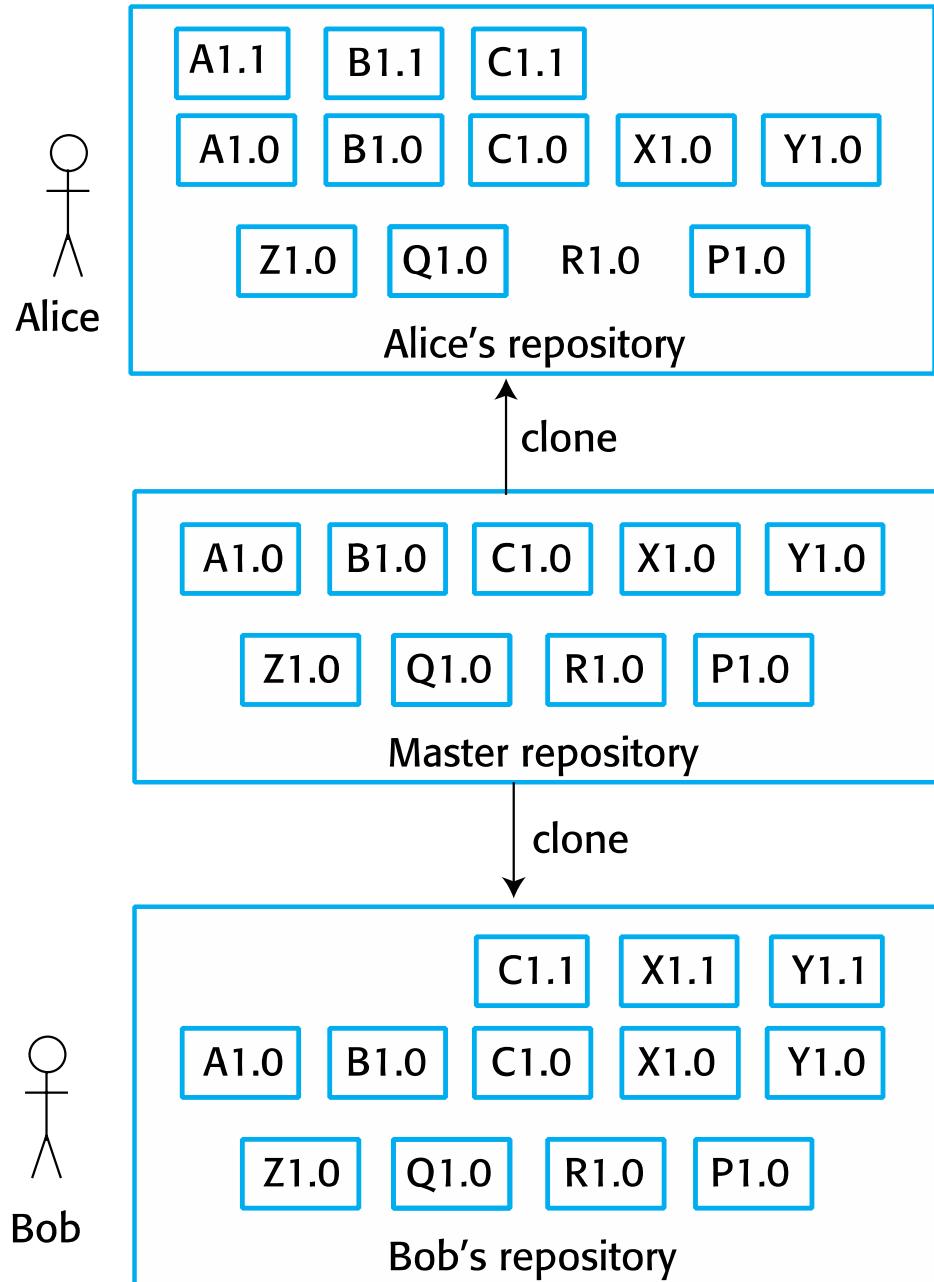
- Developers check out components or directories of components from the project repository into their private workspace and work on these copies in their private workspace.
- When their changes are complete, they check-in the components back to the repository.
- If several people are working on a component at the same time, each check it out from the repository. If a component has been checked out, the VC system warns other users wanting to check out that component that it has been checked out by someone else.

# Repository Check-in/Check-out



# Distributed version control

- A ‘master’ repository is created on a server that maintains the code produced by the development team.
- Instead of checking out the files that they need, a developer creates a clone of the project repository that is downloaded and installed on their computer.
- Developers work on the files required and maintain the new versions on their private repository on their own computer.
- When changes are done, they ‘commit’ these changes and update their private server repository. They may then ‘push’ these changes to the project repository.



# Repository cloning

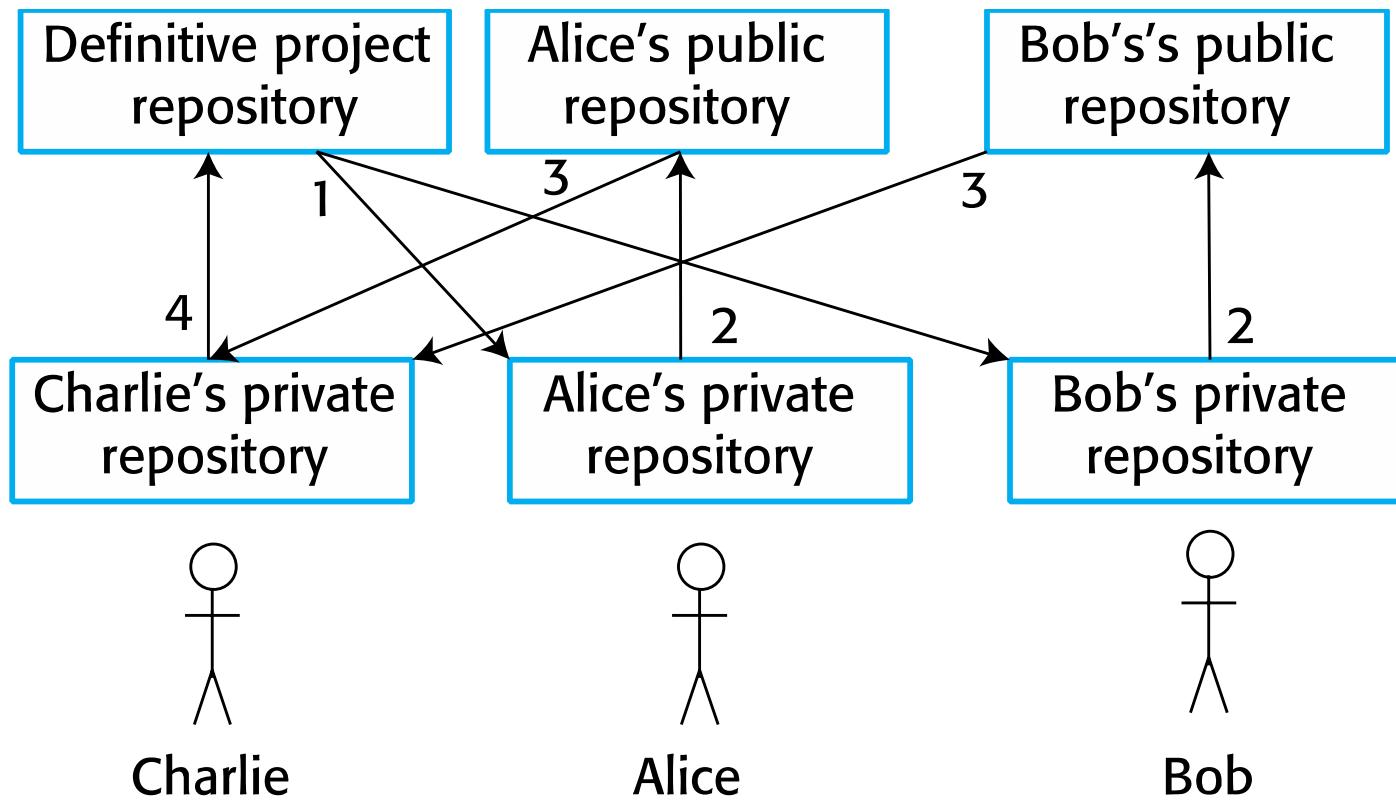
# Benefits of distributed version control

- It provides a backup mechanism for the repository.
  - If the repository is corrupted, work can continue and the project repository can be restored from local copies.
- It allows for off-line working so that developers can commit changes if they do not have a network connection.
- Project support is the default way of working.
  - Developers can compile and test the entire system on their local machines and test the changes that they have made.

# Open source development

- Distributed version control is essential for open source development.
  - Several people may be working simultaneously on the same system without any central coordination.
- As well as a private repository on their own computer, developers also maintain a public server repository to which they push new versions of components that they have changed.
  - It is then up to the open-source system ‘manager’ to decide when to pull these changes into the definitive system.

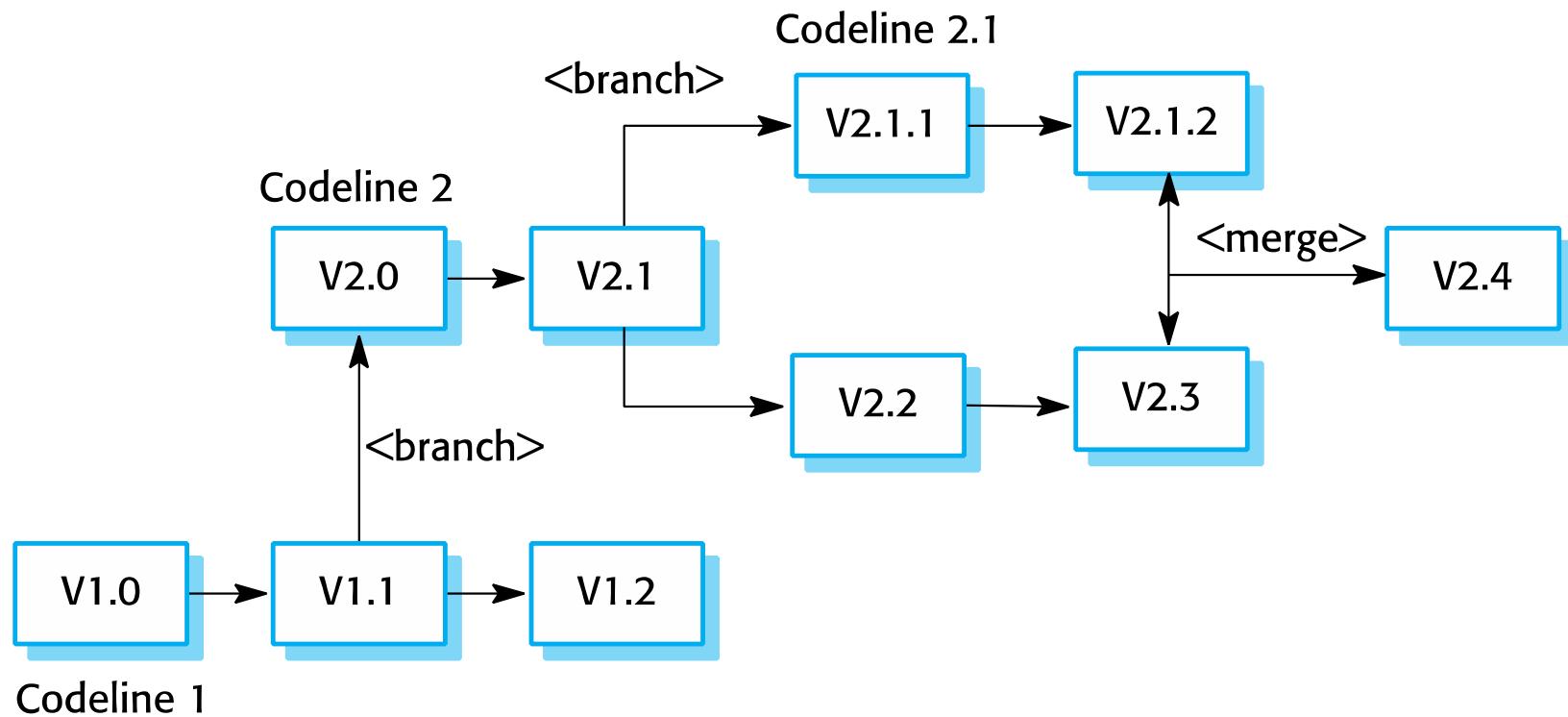
# Open-source development



# Branching and merging

- Rather than a linear sequence of versions that reflect changes to the component over time, there may be several independent sequences.
  - This is normal in system development, where different developers work independently on different versions of the source code and so change it in different ways.
- At some stage, it may be necessary to merge codeline branches to create a new version of a component that includes all changes that have been made.
  - If the changes made involve different parts of the code, the component versions may be merged automatically by combining the deltas that apply to the code.

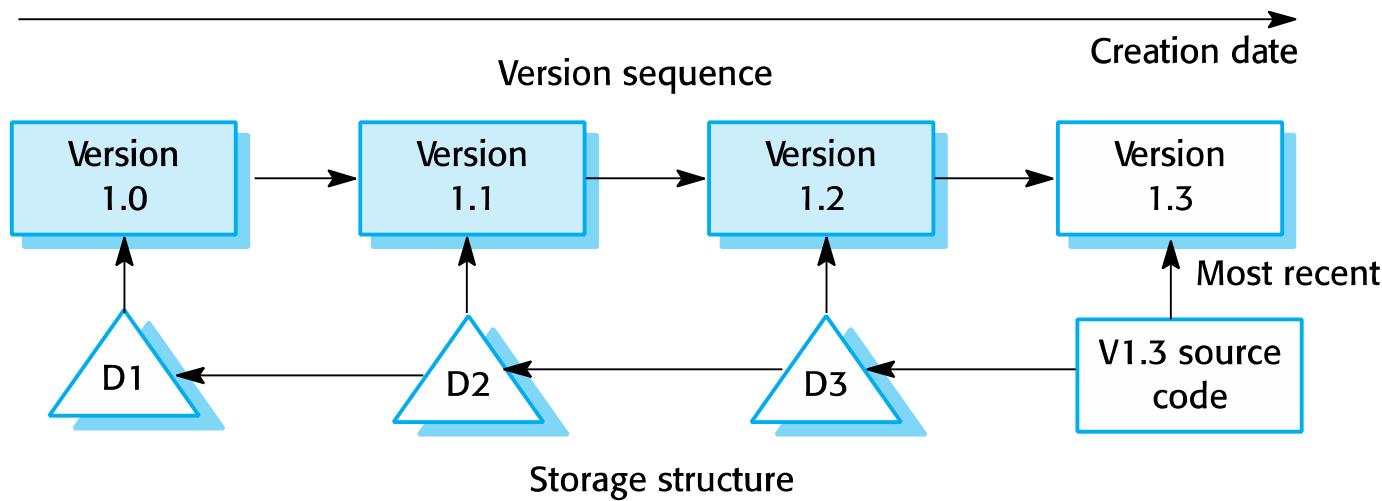
# Branching and merging



# Storage management

- When version control systems were first developed, storage management was one of their most important functions.
- Disk space was expensive and it was important to minimize the disk space used by the different copies of components.
- Instead of keeping a complete copy of each version, the system stores a list of differences (deltas) between one version and another.
  - By applying these to a master version (usually the most recent version), a target version can be recreated.

# Storage management using deltas



# Storage management in Git

- As disk storage is now relatively cheap, Git uses an alternative, faster approach.
- Git does not use deltas but applies a standard compression algorithm to stored files and their associated meta-information.
- It does not store duplicate copies of files. Retrieving a file simply involves decompressing it, with no need to apply a chain of operations.
- Git also uses the notion of packfiles where several smaller files are combined into an indexed single file.

# Configuration Management

## Part 2

Lecture 30

# System building

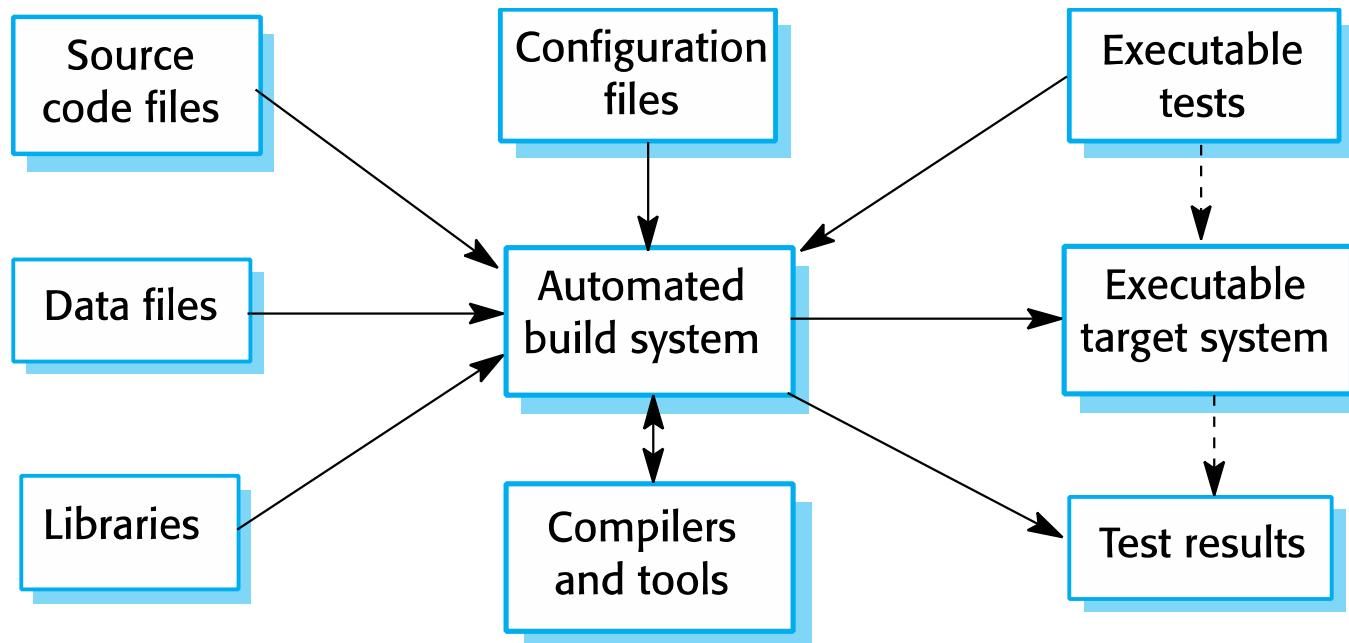
# System building

- System building is the process of creating a complete, executable system by compiling and linking the system components, external libraries, configuration files, etc.
- System building tools and version management tools must communicate as the build process involves checking out component versions from the repository managed by the version management system.
- The configuration description used to identify a baseline is also used by the system building tool.

# Build platforms

- The development system, which includes development tools such as compilers, source code editors, etc.
  - Developers check out code from the version management system into a private workspace before making changes to the system.
- The build server, which is used to build definitive, executable versions of the system.
  - Developers check-in code to the version management system before it is built. The system build may rely on external libraries that are not included in the version management system.
- The target environment, which is the platform on which the system executes.

# System building



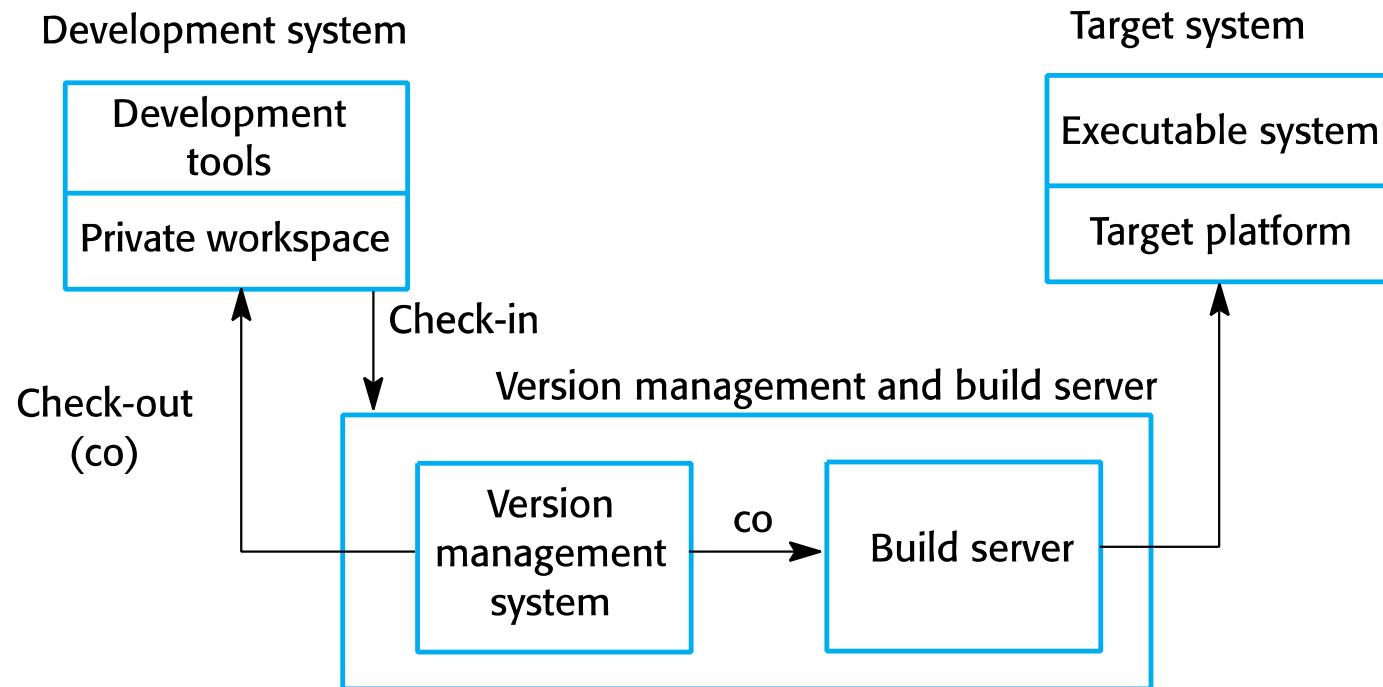
# Build system functionality

- Build script generation
- Version management system integration
- Minimal re-compilation
- Executable system creation
- Test automation
- Reporting
- Documentation generation

# System platforms

- The development system, which includes development tools such as compilers, source code editors, etc.
- The build server, which is used to build definitive, executable versions of the system. This server maintains the definitive versions of a system.
- The target environment, which is the platform on which the system executes.
  - For real-time and embedded systems, the target environment is often smaller and simpler than the development environment (e.g. a cell phone)

# Development, build, and target platforms



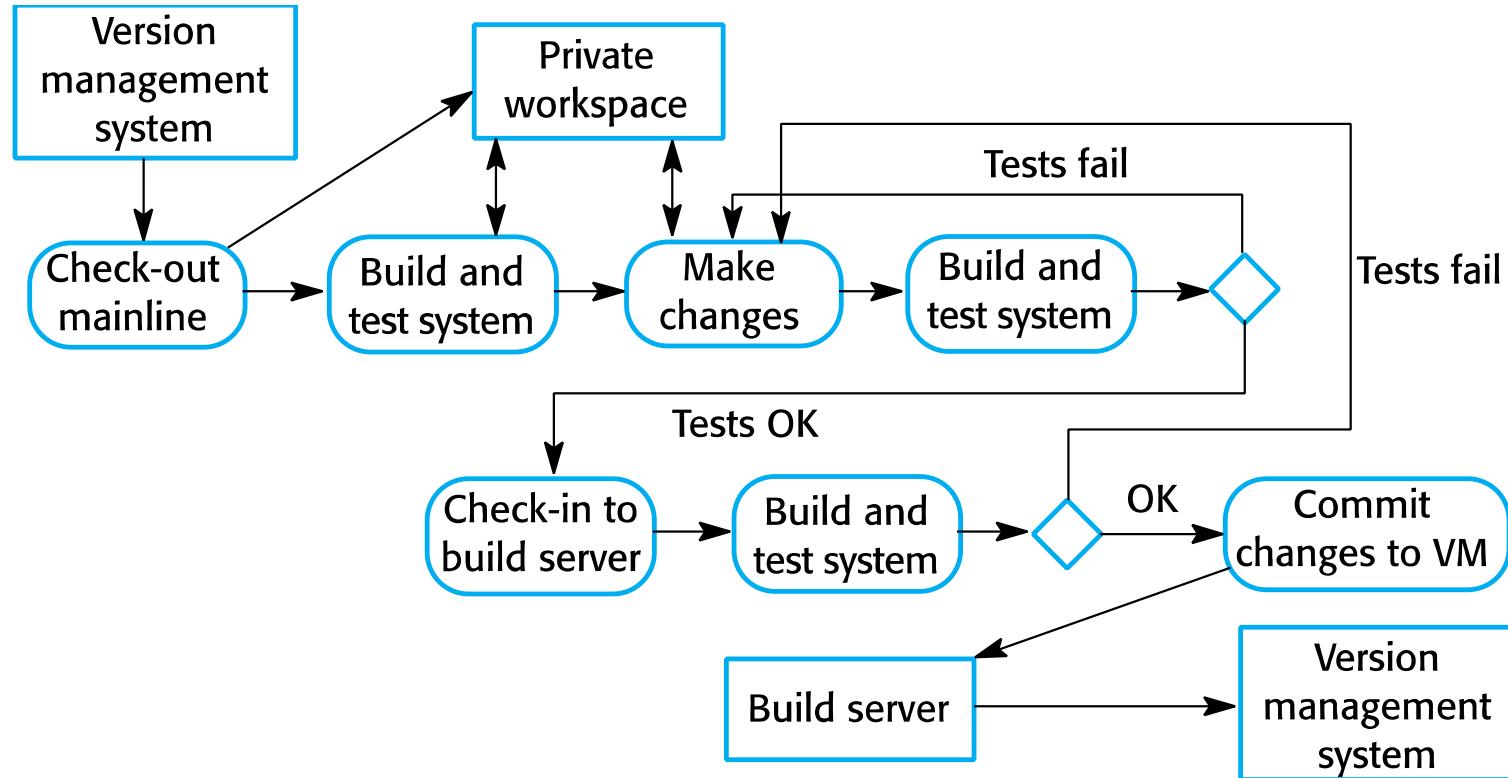
# Agile building

- Check out the mainline system from the version management system into the developer's private workspace.
- Build the system and run automated tests to ensure that the built system passes all tests. If not, the build is broken and you should inform whoever checked in the last baseline system. They are responsible for repairing the problem.
- Make the changes to the system components.
- Build the system in the private workspace and rerun system tests. If the tests fail, continue editing.

# Agile building

- Once the system has passed its tests, check it into the build system but do not commit it as a new system baseline.
- Build the system on the build server and run the tests. You need to do this in case others have modified components since you checked out the system. If this is the case, check out the components that have failed and edit these so that tests pass on your private workspace.
- If the system passes its tests on the build system, then commit the changes you have made as a new baseline in the system mainline.

# Continuous integration



# Pros and cons of continuous integration

- Pros
  - The advantage of continuous integration is that it allows problems caused by the interactions between different developers to be discovered and repaired as soon as possible.
  - The most recent system in the mainline is the definitive working system.
- Cons
  - If the system is very large, it may take a long time to build and test, especially if integration with other application systems is involved.
  - If the development platform is different from the target platform, it may not be possible to run system tests in the developer's private workspace.

# Daily building

- The development organization sets a delivery time (say 2 p.m.) for system components.
  - If developers have new versions of the components that they are writing, they must deliver them by that time.
  - A new version of the system is built from these components by compiling and linking them to form a complete system.
  - This system is then delivered to the testing team, which carries out a set of predefined system tests
  - Faults that are discovered during system testing are documented and returned to the system developers. They repair these faults in a subsequent version of the component.

# Minimizing recompilation

- Tools to support system building are usually designed to minimize the amount of compilation that is required.
- They do this by checking if a compiled version of a component is available. If so, there is no need to recompile that component.
- A unique signature identifies each source and object code version and is changed when the source code is edited.
- By comparing the signatures on the source and object code files, it is possible to decide if the source code was used to generate the object code component.

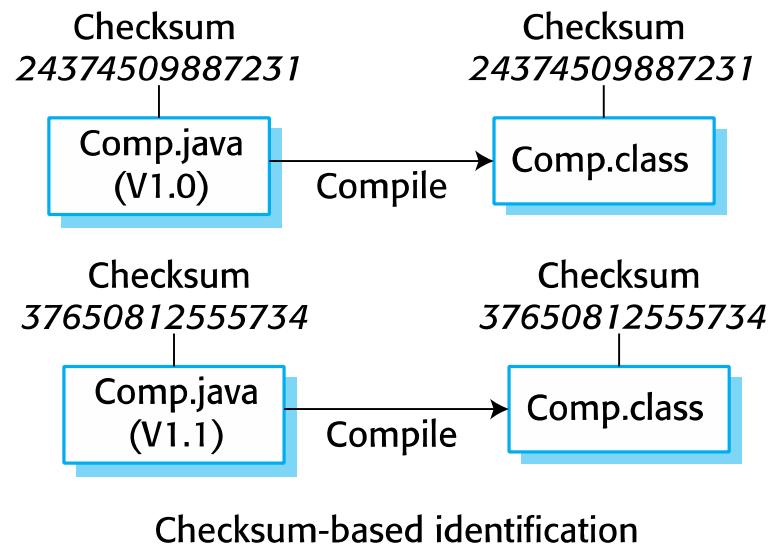
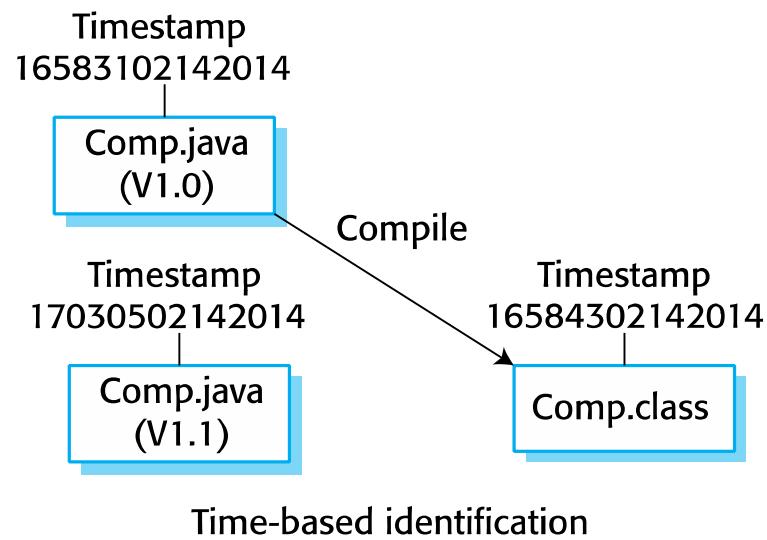
# File identification

- Modification timestamps
  - The signature on the source code file is the time and date when that file was modified. If the source code file of a component has been modified after the related object code file, then the system assumes that recompilation to create a new object code file is necessary.
- Source code checksums
  - The signature on the source code file is a checksum calculated from data in the file. A checksum function calculates a unique number using the source text as input. If you change the source code (even by 1 character), this will generate a different checksum. You can therefore be confident that source code files with different checksums are actually different.

# Timestamps vs checksums

- Timestamps
  - Because source and object files are linked by name rather than an explicit source file signature, it is not usually possible to build different versions of a source code component into the same directory at the same time, as these would generate object files with the same name.
- Checksums
  - When you recompile a component, it does not overwrite the object code, as would normally be the case when the timestamp is used. Rather, it generates a new object code file and tags it with the source code signature. Parallel compilation is possible and different versions of a component may be compiled at the same time.

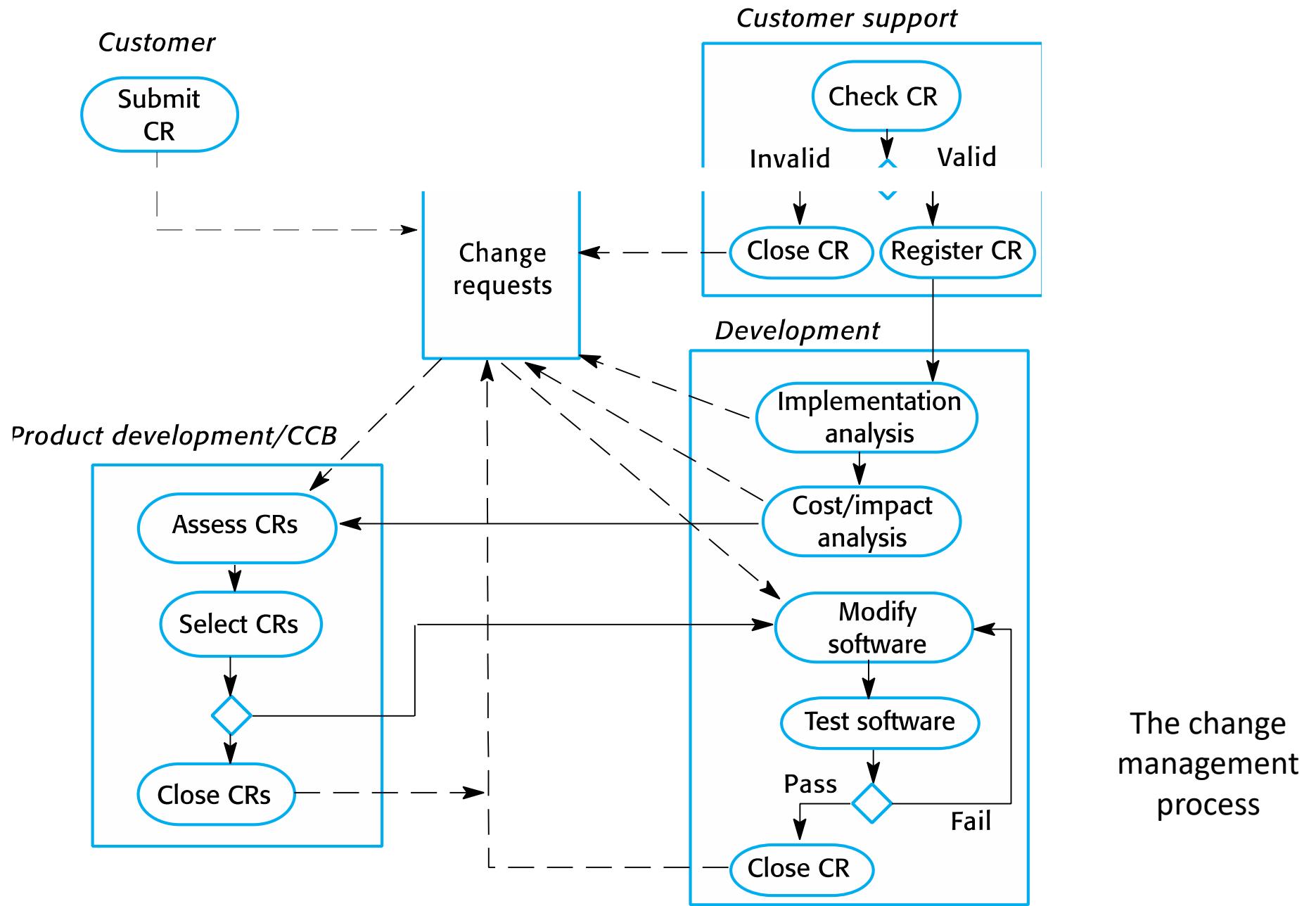
# Linking source and object code



# Change management

# Change management

- Organizational needs and requirements change during the lifetime of a system, bugs have to be repaired and systems have to adapt to changes in their environment.
- Change management is intended to ensure that system evolution is a managed process and that priority is given to the most urgent and cost-effective changes.
- The change management process is concerned with analyzing the costs and benefits of proposed changes, approving those changes that are worthwhile and tracking which components in the system have been changed.



# A partially completed change request form (a)

## Change Request Form

**Project:** SICSA/AppProcessing

**Number:** 23/02

**Change requester:** I. Sommerville

**Date:** 20/07/12

**Requested change:** The status of applicants (rejected, accepted, etc.) should be shown visually in the displayed list of applicants.

**Change analyzer:** R. Looek

**Analysis date:** 25/07/12

**Components affected:** ApplicantListDisplay, StatusUpdater

**Associated components:** StudentDatabase

# A partially completed change request form (b)

## Change Request Form

**Change assessment:** Relatively simple to implement by changing the display color according to status. A table must be added to relate status to colors. No changes to associated components are required.

**Change priority:** Medium

**Change implementation:**

**Estimated effort:** 2 hours

**Date to SGA app. team:** 28/07/12

**CCB decision date:** 30/07/12

**Decision:** Accept change. Change to be implemented in Release 1.2

**Change implementor:**                   **Date of change:**

**Date submitted to QA:**                   **QA decision:**

**Date submitted to CM:**

**Comments:**

# Factors in change analysis

- The consequences of not making the change
- The benefits of the change
- The number of users affected by the change
- The costs of making the change
- The product release cycle

# Derivation history

// SICSA project (XEP 6087)

//

// APP-SYSTEM/AUTH/RBAC/USER\_ROLE

//

// Object: currentRole

// Author: R. Looek

// Creation date: 13/11/2012

//

// © St Andrews University 2012

//

// Modification history

// Version Modifier Date

Change

Reason

// 1.0 J. Jones 11/11/2009

Add header

Submitted to CM

// 1.1 R. Looek 13/11/2012

New field

Change req. R07/02

# Change management and agile methods

- In some agile methods, customers are directly involved in change management.
- They propose a change to the requirements and work with the team to assess its impact and decide whether the change should take priority over the features planned for the next increment of the system.
- Changes to improve the software improvement are decided by the programmers working on the system.
- Refactoring, where the software is continually improved, is not seen as an overhead but as a necessary part of the development process.

# Release management

# Release management

- A system release is a version of a software system that is distributed to customers.
- For mass market software, it is usually possible to identify two types of release: major releases which deliver significant new functionality, and minor releases, which repair bugs and fix customer problems that have been reported.
- For custom software or software product lines, releases of the system may have to be produced for each customer and individual customers may be running several different releases of the system at the same time.

# Release components

- As well as the executable code of the system, a release may also include:
  - configuration files defining how the release should be configured for particular installations;
  - data files, such as files of error messages, that are needed for successful system operation;
  - an installation program that is used to help install the system on target hardware;
  - electronic and paper documentation describing the system;
  - packaging and associated publicity that have been designed for that release.

# Factors influencing system release planning

Factor	Description
Competition	For mass-market software, a new system release may be necessary because a competing product has introduced new features and market share may be lost if these are not provided to existing customers.
Marketing requirements	The marketing department of an organization may have made a commitment for releases to be available at a particular date.
Platform changes	You may have to create a new release of a software application when a new version of the operating system platform is released.
Technical quality of the system	If serious system faults are reported which affect the way in which many customers use the system, it may be necessary to issue a fault repair release. Minor system faults may be repaired by issuing patches (usually distributed over the Internet) that can be applied to the current release of the system.

# Release creation

- The executable code of the programs and all associated data files must be identified in the version control system.
- Configuration descriptions may have to be written for different hardware and operating systems.
- Update instructions may have to be written for customers who need to configure their own systems.
- Scripts for the installation program may have to be written.
- Web pages have to be created describing the release, with links to system documentation.
- When all information is available, an executable master image of the software must be prepared and handed over for distribution to customers or sales outlets.

# Release tracking

- In the event of a problem, it may be necessary to reproduce exactly the software that has been delivered to a particular customer.
- When a system release is produced, it must be documented to ensure that it can be re-created exactly in the future.
- This is particularly important for customized, long-lifetime embedded systems, such as those that control complex machines.
  - Customers may use a single release of these systems for many years and may require specific changes to a particular software system long after its original release date.

# Release reproduction

- To document a release, you have to record the specific versions of the source code components that were used to create the executable code.
- You must keep copies of the source code files, corresponding executables and all data and configuration files.
- You should also record the versions of the operating system, libraries, compilers and other tools used to build the software.

# Release planning

- As well as the technical work involved in creating a release distribution, advertising and publicity material have to be prepared and marketing strategies put in place to convince customers to buy the new release of the system.
- Release timing
  - If releases are too frequent or require hardware upgrades, customers may not move to the new release, especially if they have to pay for it.
  - If system releases are too infrequent, market share may be lost as customers move to alternative systems.

# Software as a service

- Delivering software as a service (SaaS) reduces the problems of release management.
- It simplifies both release management and system installation for customers.
- The software developer is responsible for replacing the existing release of a system with a new release and this is made available to all customers at the same time.

# Key points

- Configuration management is the management of an evolving software system. When maintaining a system, a CM team is put in place to ensure that changes are incorporated into the system in a controlled way and that records are maintained with details of the changes that have been implemented.
- The main configuration management processes are concerned with version management, system building, change management, and release management.
- Version management involves keeping track of the different versions of software components as changes are made to them.

# Key points

- System building is the process of assembling system components into an executable program to run on a target computer system.
- Software should be frequently rebuilt and tested immediately after a new version has been built. This makes it easier to detect bugs and problems that have been introduced since the last build.
- Change management involves assessing proposals for changes from system customers and other stakeholders and deciding if it is cost-effective to implement these in a new version of a system.
- System releases include executable code, data files, configuration files and documentation. Release management involves making decisions on system release dates, preparing all information for distribution and documenting each system release.