Chapter 7 - Design and Implementation

MI Qing (Lecturer)

Telephone: 15210503242

Email: miqing@bjut.edu.cn

Office: 410, Information Building

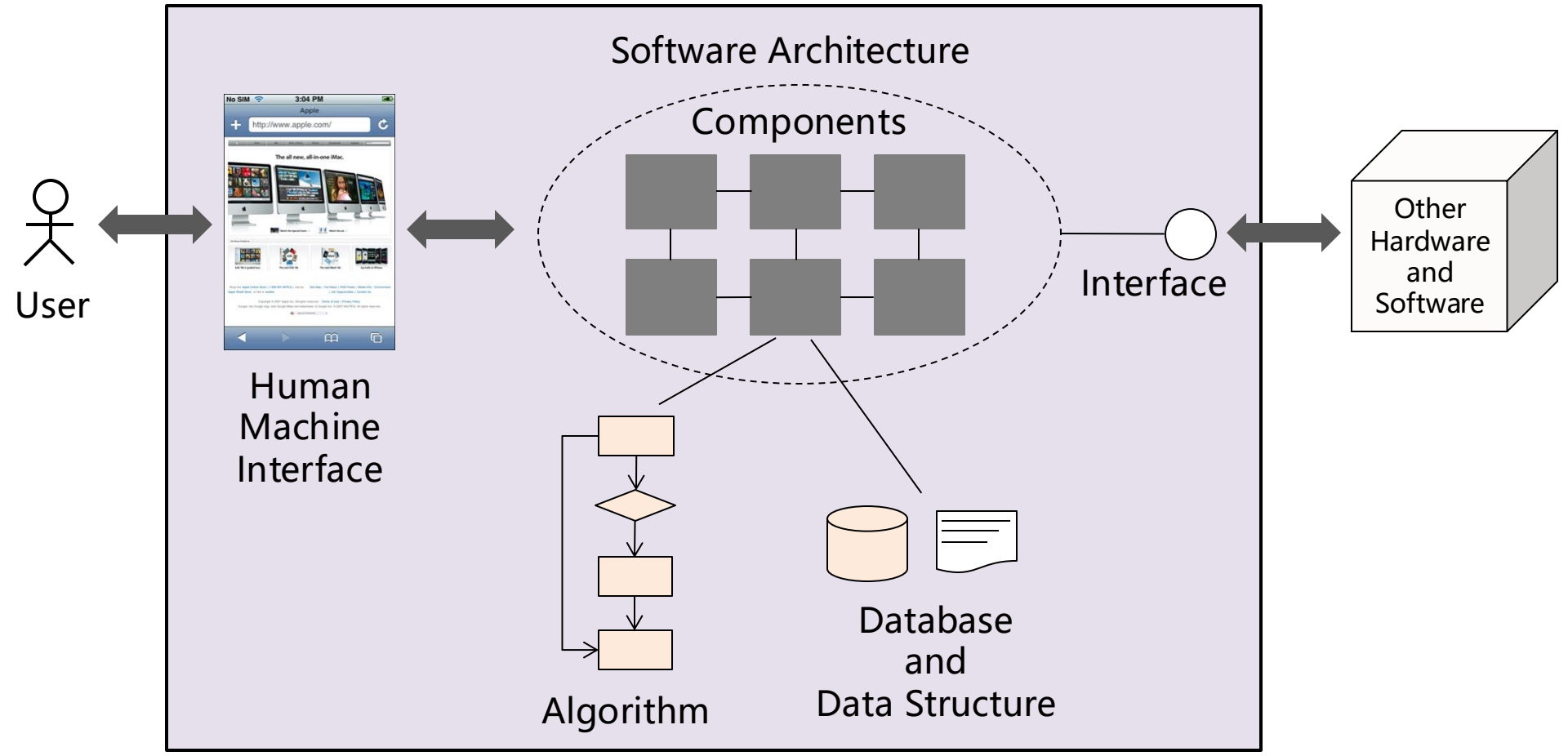
Topics Covered

- ✧ Object-oriented design using the UML
- ✧ OOD principles
- ✧ Design patterns
- ✧ Refactoring
- ✧ Implementation issues

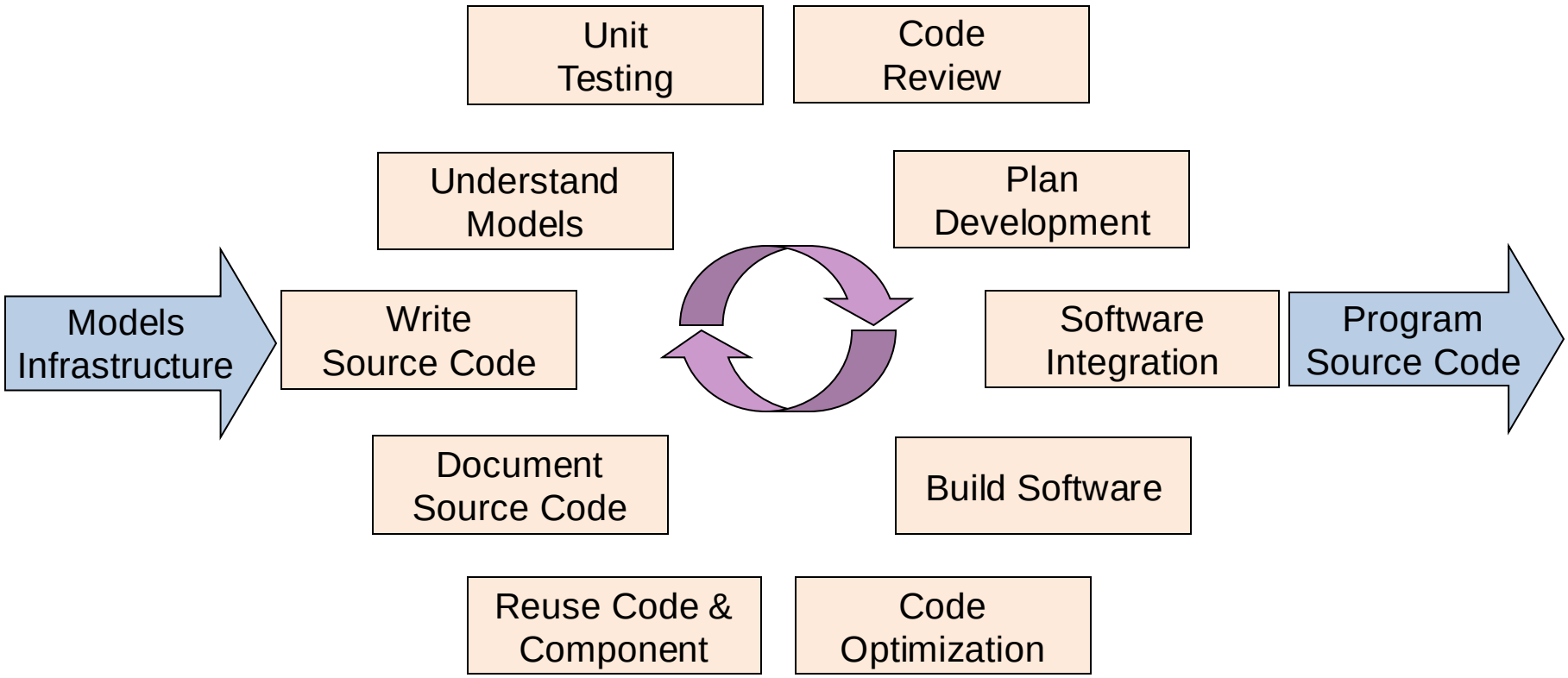
Design and Implementation

- Software design and implementation is the stage in the software engineering process at which an executable software system is developed.
- Software design and implementation activities are invariably inter-leaved.
  - Software design is a creative activity in which you identify software components and their relationships, based on a customer's requirements.
  - Implementation is the process of realizing the design as a program.

Design and Implementation



Design and Implementation



Google Style Guide
https://code.google.com/p/google-styleguide/

Software Design Descriptions (IEEE 1016-2009)

1. 引言
   - 1.1 目的
   - 1.2 范围
   - 1.3 定义和缩写词
2. 参考文献
3. 分解说明
   - 3.1 模块分解
   - 3.2 并发进程
   - 3.3 数据分解
4. 依赖关系说明
   - 4.1 模块间的依赖关系
   - 4.2 进程间的依赖关系
   - 4.3 数据间的依赖关系
5. 接口说明
   - 5.1 模块接口
   - 5.2 进程接口
6. 详细设计
   - 6.1 模块详细设计
   - 6.2 数据详细设计

附录

注：3-5是体系结构设计；6是详细设计。

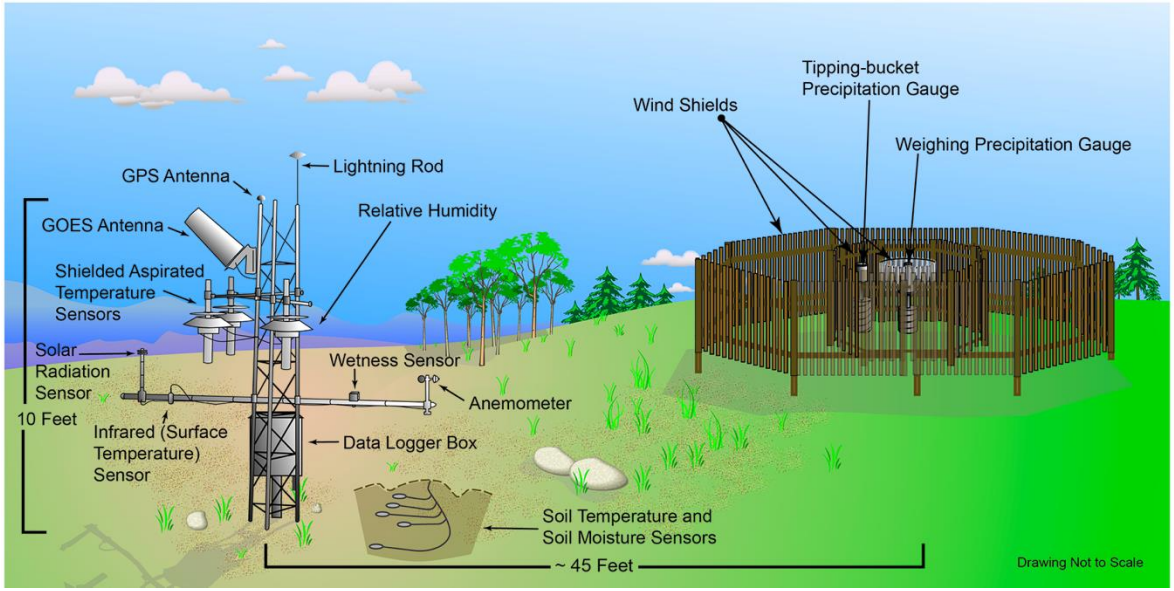
Object-Oriented Design using UML

An Object-Oriented Design Process

- Object-oriented design processes involve designing object classes and the relationships between these classes.
- To develop a system design from concept to detailed:
  1. Understand and define the context and the external interactions with the system.
  2. Design the system architecture.
  3. Identify the principal objects in the system.
  4. Develop design models.
  5. Specify interfaces.

A Wilderness Weather Station

- Wilderness weather stations are deployed in remote areas. Each weather station records local weather information and periodically transfers this to a weather information system, using a satellite link.

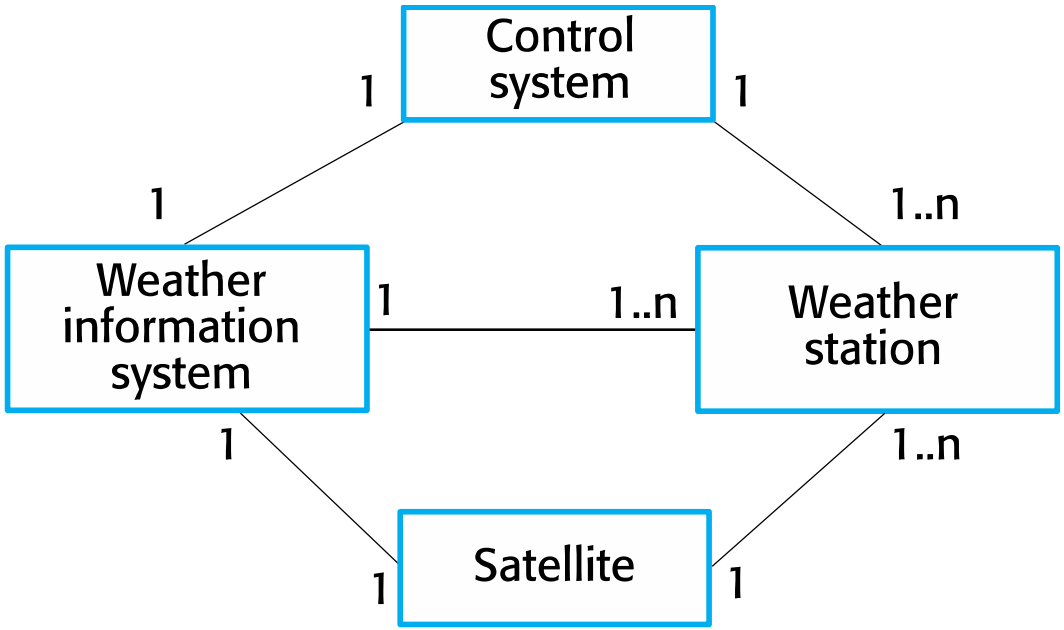


Step 1: System Context and Interactions

- Understanding the relationships between the software that is being designed and its external environment is essential for deciding how to provide the required system functionality and how to structure the system to communicate with its environment.
- Understanding of the context also lets you establish the boundaries of the system. Setting the system boundaries helps you decide what features are implemented in the system being designed and what features are in other associated systems.

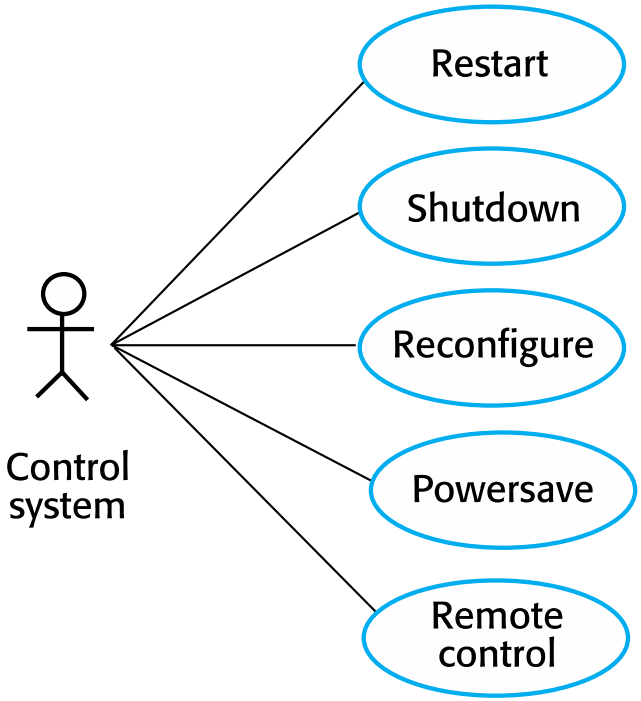
System Context for the Weather Station

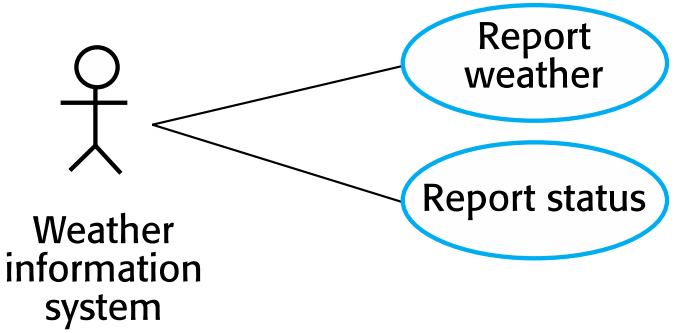
- A system context model is a structural model that demonstrates the other systems in the environment of the system being developed.



System Interactions for the Weather Station

- An interaction model is a dynamic model that shows how the system interacts with its environment as it is used.





Report weather: send weather data to the weather information system.
Report status: send status information to the weather information system.
Restart: if the weather station is shut down, restart the system.
Shutdown: shut down the weather station.
Reconfigure: reconfigure the weather station software.
Powersave: put the weather station into power-saving mode.
Remote control: send control commands to any weather station subsystem.

Use Case Description

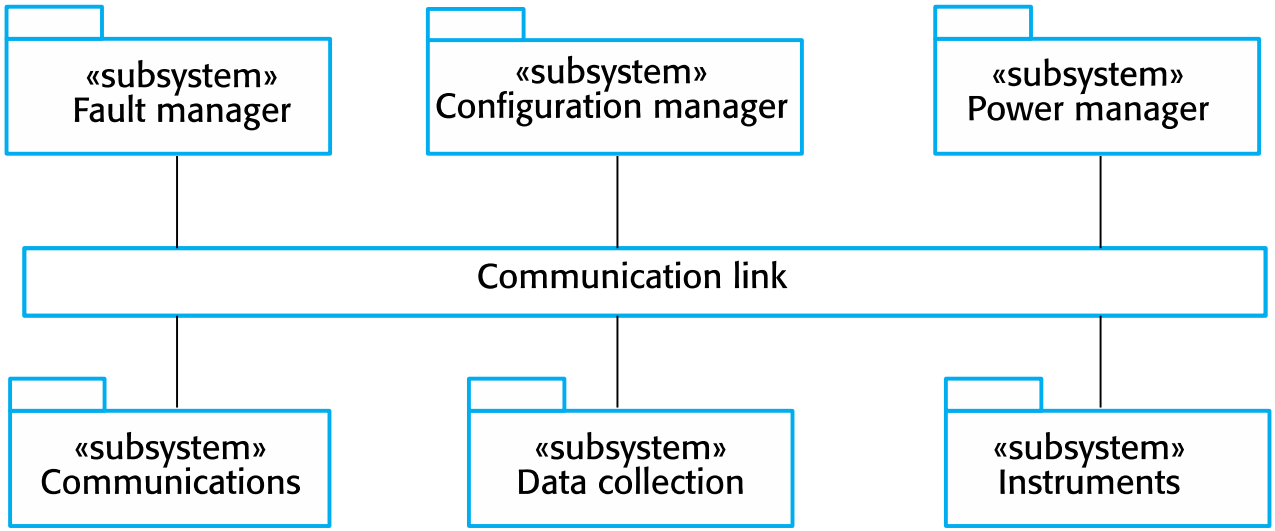
System	Weather station
Use case	Report weather
Actors	Weather information system, Weather station
Description	The weather station sends a summary of the weather data that has been collected from the instruments in the collection period to the weather information system. The data sent are the maximum, minimum, and average ground and air temperatures; the maximum, minimum, and average air pressures; the maximum, minimum, and average wind speeds; the total rainfall; and the wind direction as sampled at five-minute intervals.
Stimulus	The weather information system establishes a satellite communication link with the weather station and requests transmission of the data.
Response	The summarized data is sent to the weather information system.
Comments	Weather stations are usually asked to report once per hour but this frequency may differ from one station to another and may be modified in the future.

Step 2: Architectural Design

- Once interactions between the system and its environment have been understood, you use this information for designing the system architecture.
- You identify the major components that make up the system and their interactions, and then may organize the components using an architectural pattern such as a layered or client-server model.

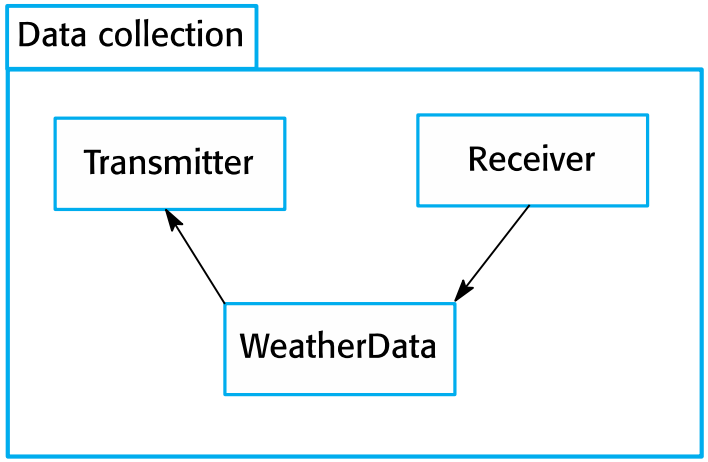
High-level Architecture of the Weather Station

- The weather station is composed of independent subsystems that communicate by broadcasting messages on a common infrastructure.
  - Each subsystem listens for messages on that infrastructure and picks up the messages that are intended for them.



Architecture of Data Collection System

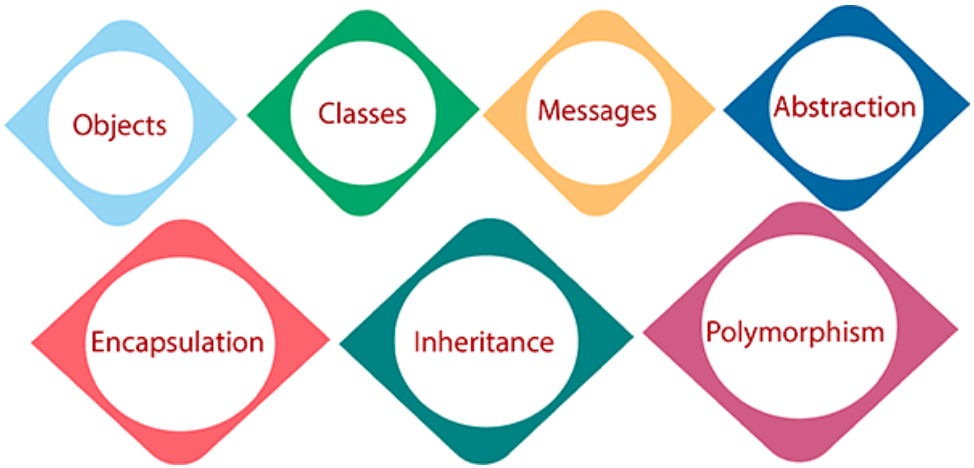
- The Transmitter and Receiver objects are concerned with managing communications.
- The WeatherData object encapsulates the information that is collected from the instruments and transmitted to the weather information system.



- Because the bandwidth to the satellite is relatively narrow, the weather station carries out some local processing and aggregation of the data.
- This arrangement follows the **producer-consumer pattern**.

Step 3: Object Class Identification

- There is no 'magic formula' for object identification. It relies on the skill, experience, and domain knowledge of system designers.
- Object identification is an iterative process. You are unlikely to get it right first time.



Approaches to Identification

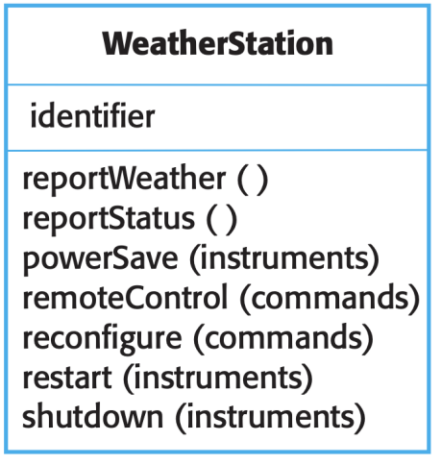
- Use a grammatical analysis of a natural language description of the system to be constructed. Objects and attributes are nouns; operations or services are verbs.
- Use tangible entities (things) in the application domain such as aircraft, roles such as manager, events such as request, interactions such as meetings, locations such as offices, organizational units such as companies, and so on.
- Use a scenario-based analysis where various scenarios of system use are identified and analyzed in turn.

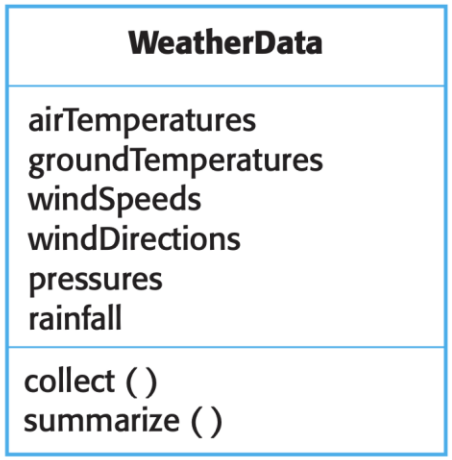
Weather Station Object Classes

- Object class identification in the weather station system may be based on the tangible hardware and data in the system:
  - Weather station
    - The basic interface of the weather station to its environment. It therefore reflects the interactions identified in the use-case model.
  - Weather data
    - Encapsulates the summarized data from the instruments.
  - Ground thermometer, Anemometer, Barometer
    - Application domain objects that are ‘hardware’ objects related to the instruments in the system.

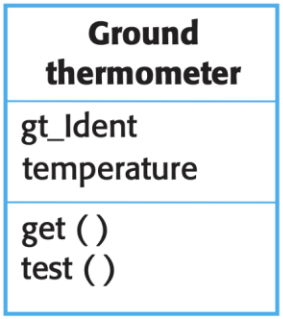
Weather Station Object Classes

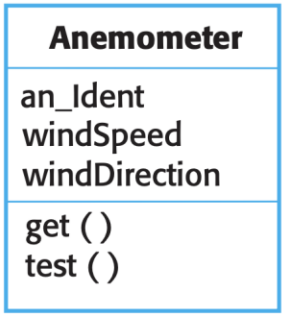
Each weather station should have its own identifier so that it can be uniquely identified in communications.

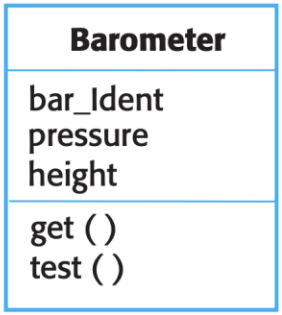




Look for common features and then design the inheritance hierarchy for the system (e.g., an Instrument superclass).





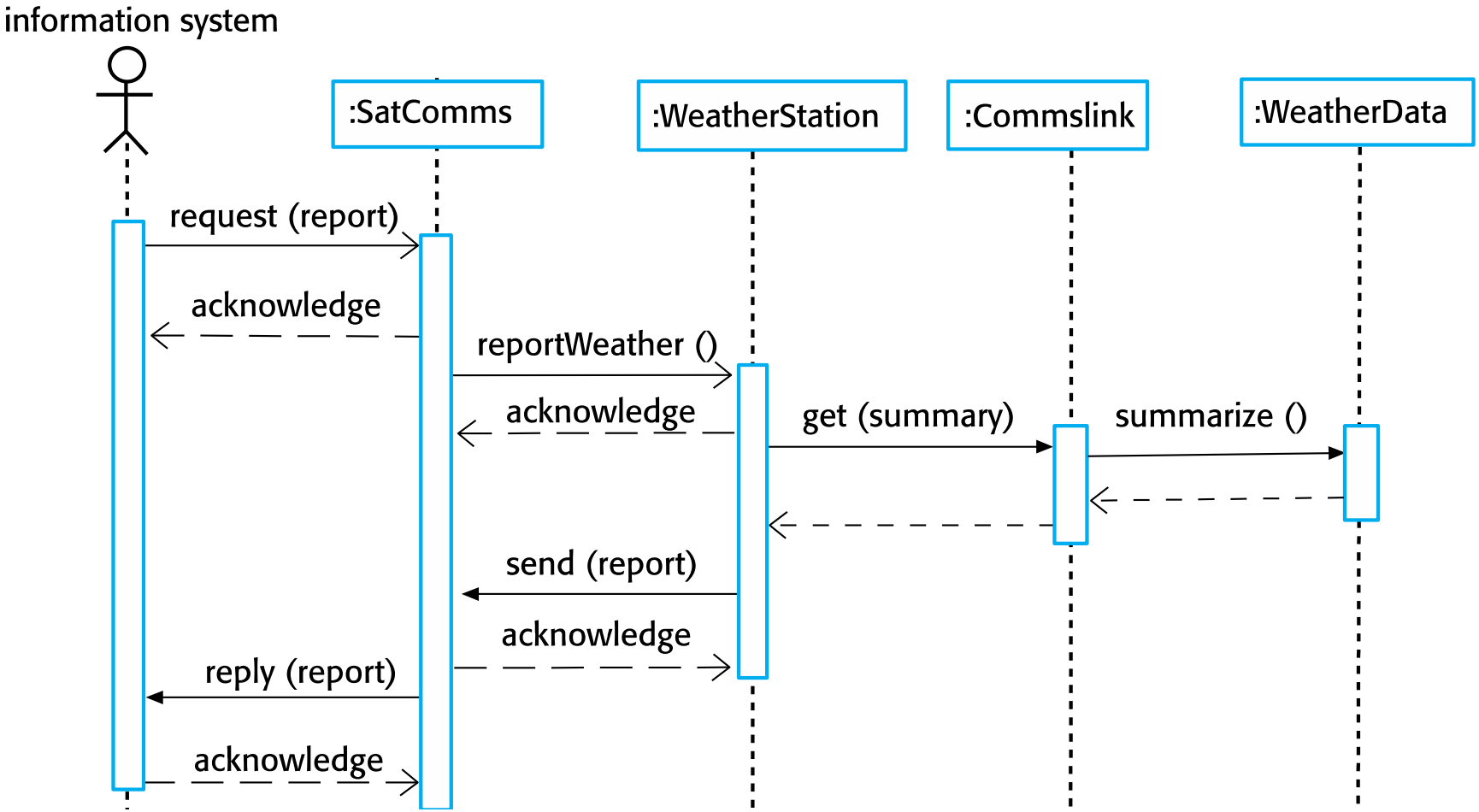


Instrument failures should be checked and reported.

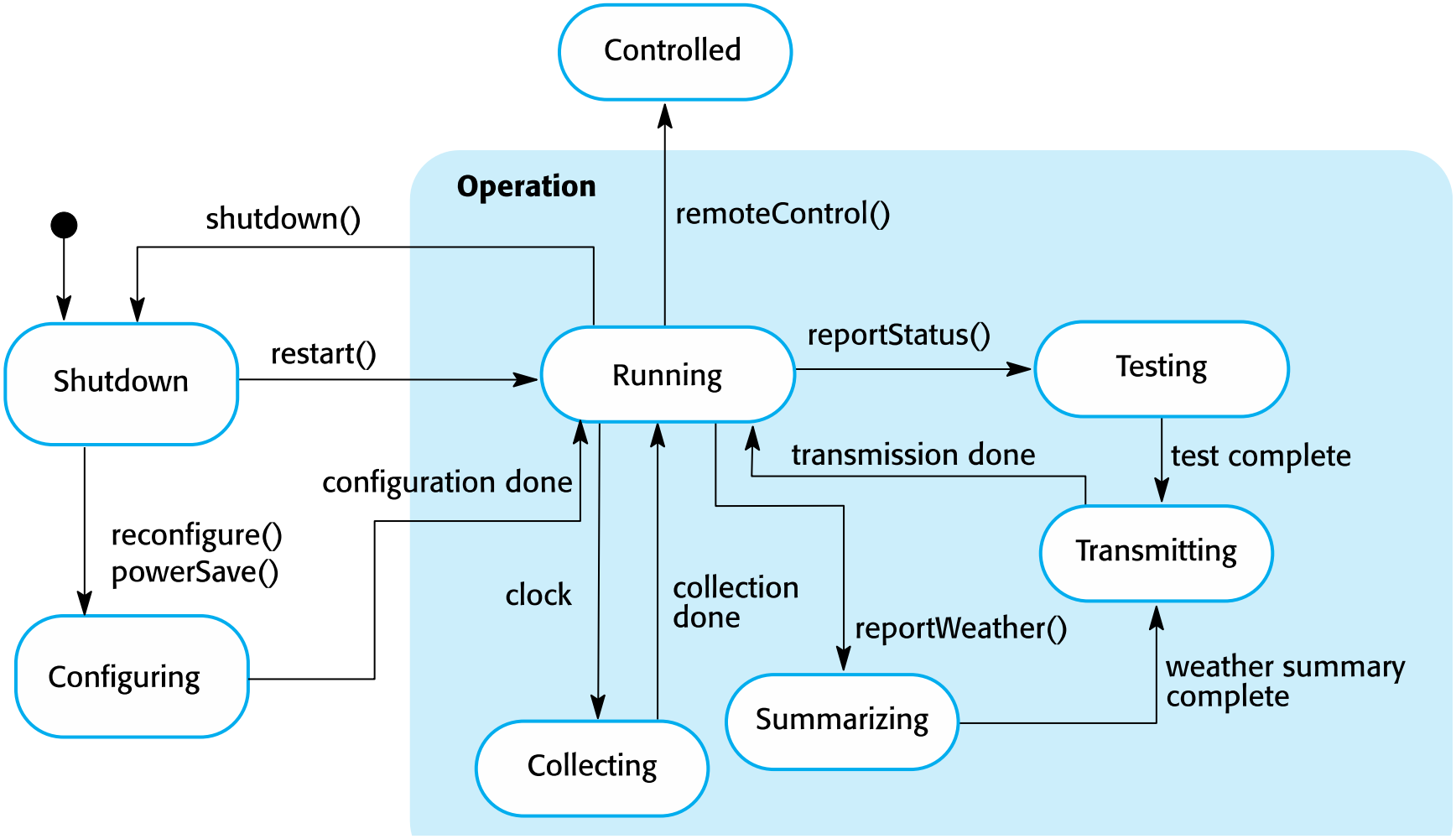
Step 4: Design Models

- Design or system models show the objects or object classes in a system. They also show the associations and relationships between these entities.
- An important step in the design process is to decide on the design models that you need and the level of detail required in these models.
- There are two kinds of design model:
  - Structural models describe the static structure of the system in terms of object classes and relationships.
  - Dynamic models describe the dynamic interactions between objects.

Sequence Diagram - Data Collection

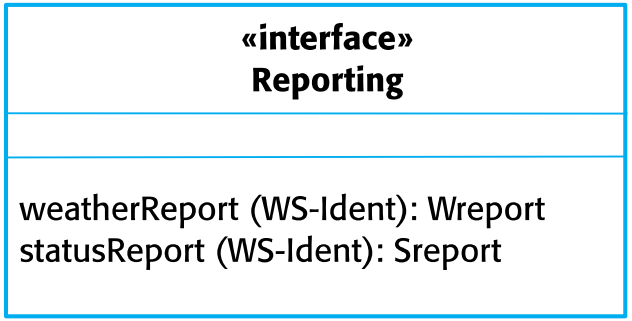


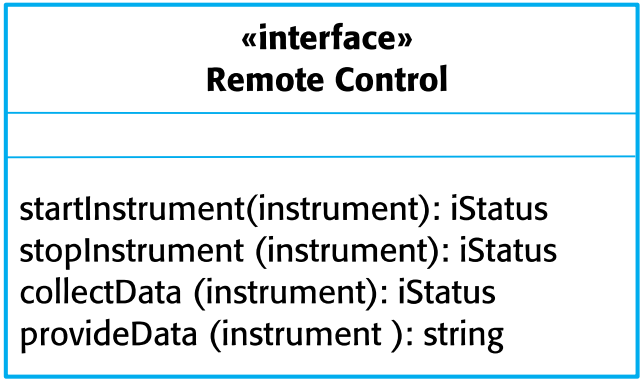
State Diagram - Weather Station



Step 5: Interface Specification

- Interfaces should to be specified so that objects and subsystems can be designed in parallel.
- Interface design is concerned with specifying the detail of the interface to an object or to a group of objects.
  - This means defining the signatures and semantics of the services that are provided by the object or by a group of objects.





OOD Principles

OOD Principles

SOLID原则，给例子问违反了哪个

- SOLID is a design principle that plays a very important role during object-oriented design.
  - The SOLID principles were first introduced by Robert J. Martin in his paper in 2000. But the SOLID acronym was introduced later by Michael Feathers.
- Software development becomes easy, reusable, flexible, and maintainable using these design principles.

Single Resposibility Principle
A class should have only a single responsibility (i.e. only one potential change in the software's specification should be able to affect the specification of the class)

Open / Closed Principle
A software module (it can be a class or method) should be open for extension but closed for modification.

Liskov Substitution Principle
Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.

Interface Segregation Principle
Clients should not be forced to depend upon the interfaces that they do not use.

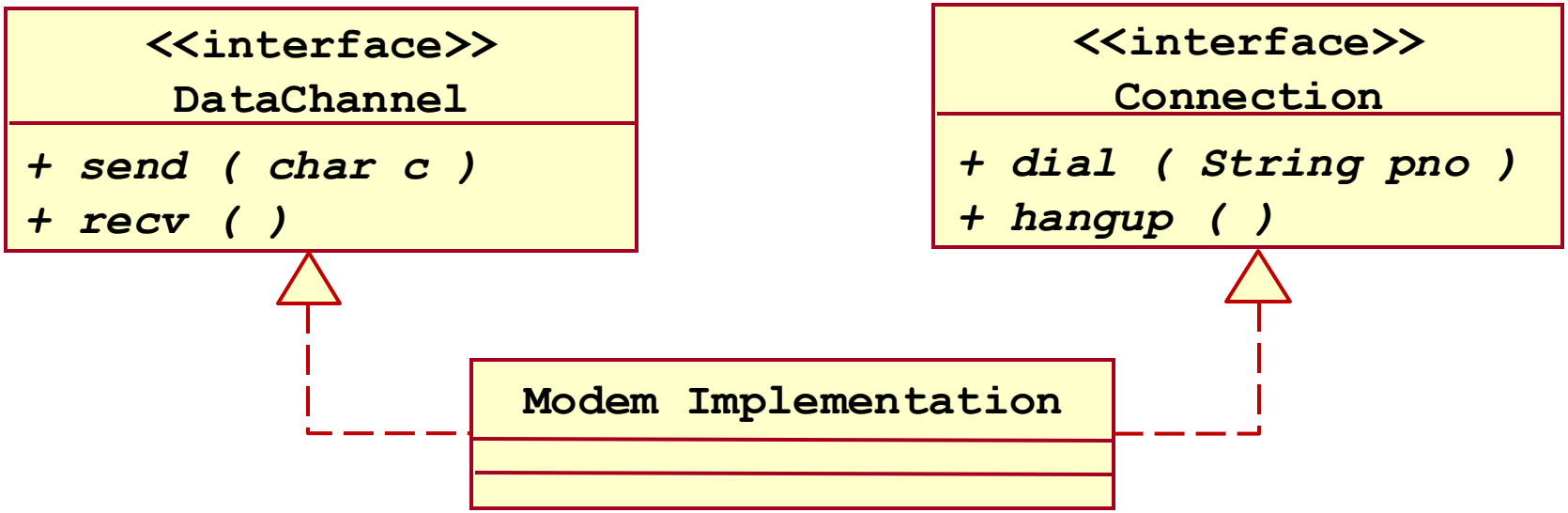
Dependency Inversion Principle
Program to an interface, not to an implementation.

Single Responsibility Principle

- The Single Responsibility Principle states that each class should have only one responsibility, meaning it should do one thing and do that thing well.
  - A responsibility is a reason to change, so each class should have only one reason to change.
- If a class has multiple responsibilities, there is a likelihood that it is used in a greater number of places. When one responsibility is changed, not only do we run a higher risk of introducing defects into other responsibilities in the same class, but there is a greater number of other classes that might be impacted.

Single Responsibility Principle

```
interface Modem {
    public void dial(String pno);
    public void hangup();
    public void send(Char c);
    public char recv();
}
```



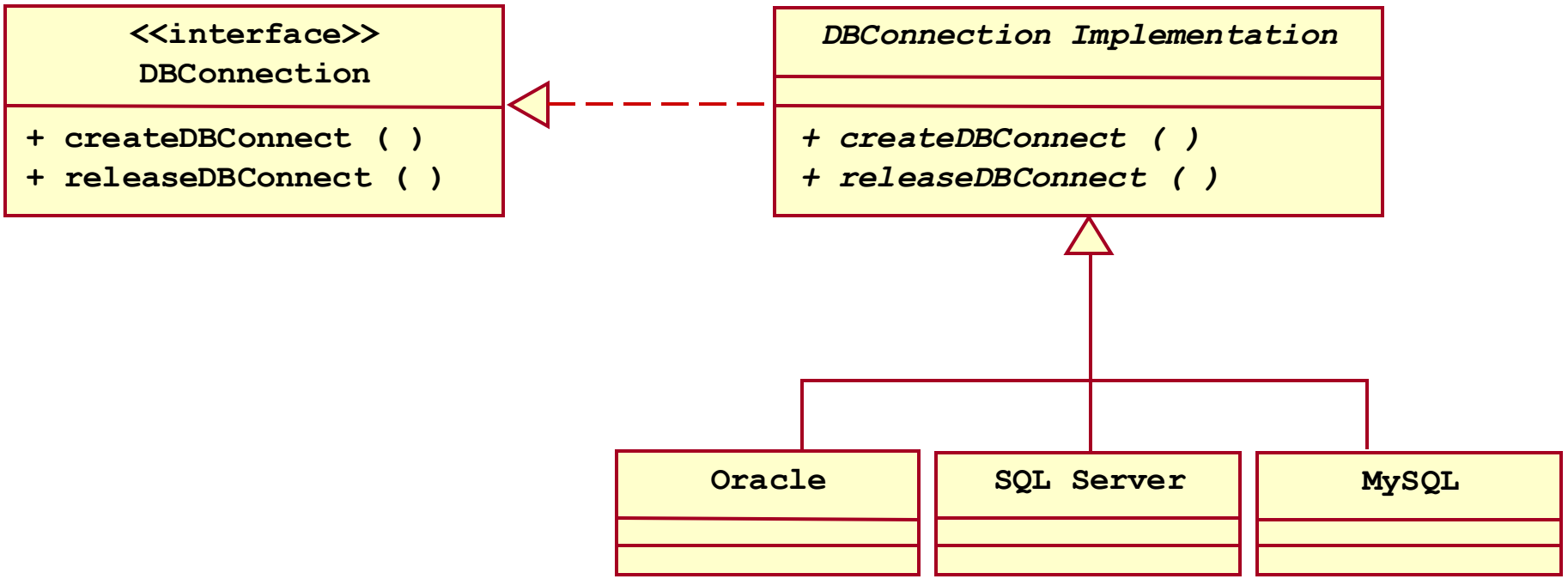
Open Closed Principle

- The Open Closed Principle states that software components should be open for extension but closed for modification.
- When requirements change, the design should minimize the amount of changes that need to occur on existing code. We should be able to extend a component by adding new code without having to modify existing code that already works.

Open Closed Principle

```
bool createDBConnect(...)
{
    ...
    switch (dbType)
    {
        case ORACLE:
          建立 Oracle 连接;
              break;
        case SQLSERVER:
          建立 SQL Server 连接;
              break;
    }
    ...
}
```

Open Closed Principle



Liskov Substitution Principle

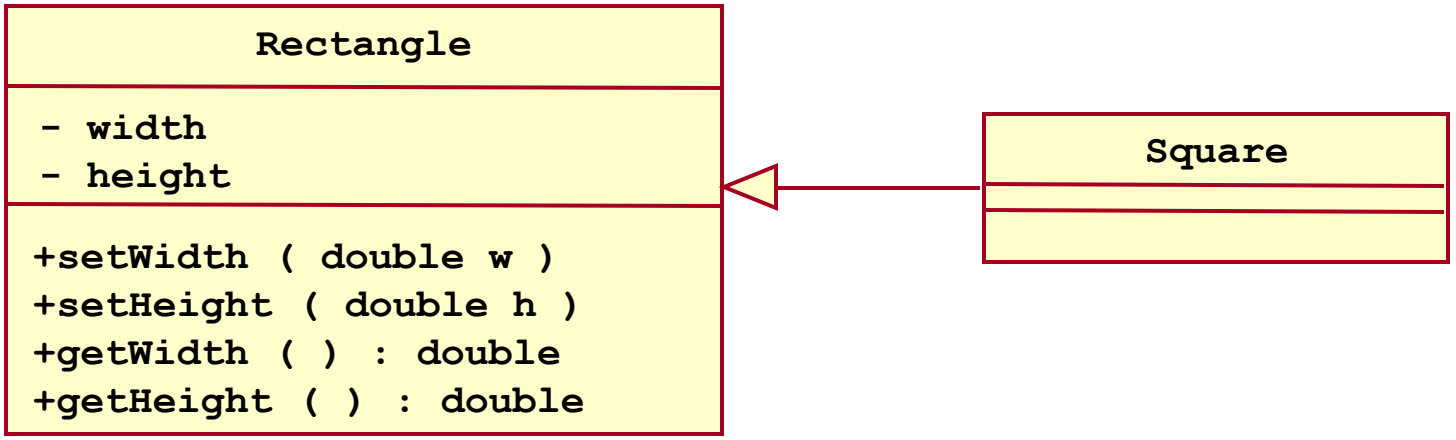
✧ The Liskov Substitution Principle states that subtypes must be substitutable for their base types without having to alter the base type.

> ***Inheritance should ensure that any property proved about supertype objects also holds for subtype objects.***
>
> **B. Liskov, 1987**

> ***Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.***
>
> **R. Martin, 1996**

子类型必须能够完全替换其父类型！

Liskov Substitution Principle



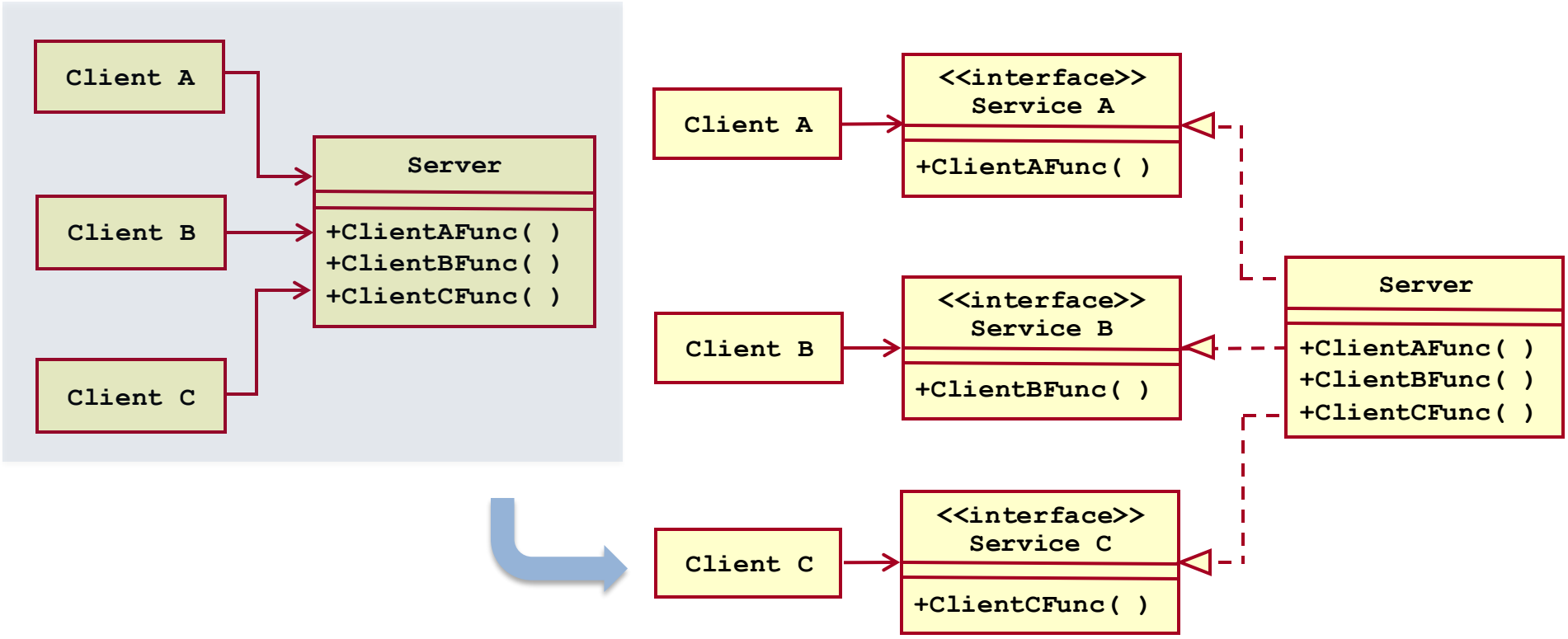

```
void Square::setWidth(double w)
{
    Rectangle::setWidth(w);
    Rectangle::setHeight(w);
}
void Square::setHeight(double h)
{
    Rectangle::setWidth(h);
    Rectangle::setHeight(h);
}
```

```
void g(Rectangle& r)
{
    r.setWidth(5);
    r.setHeight(4);
    assert(r.Area() == 20);
}
```

Interface Segregation Principle

- The Interface Segregation Principle states that clients should not be forced to depend on properties and methods that they do not use.
  - When designing software, we prefer smaller, more cohesive interfaces. If an interface is too large, we can logically split it up into multiple interfaces so that clients can focus on only the properties and methods that are of interest to them.
- Violation of the ISP increases coupling and makes maintenance more difficult.

Interface Segregation Principle

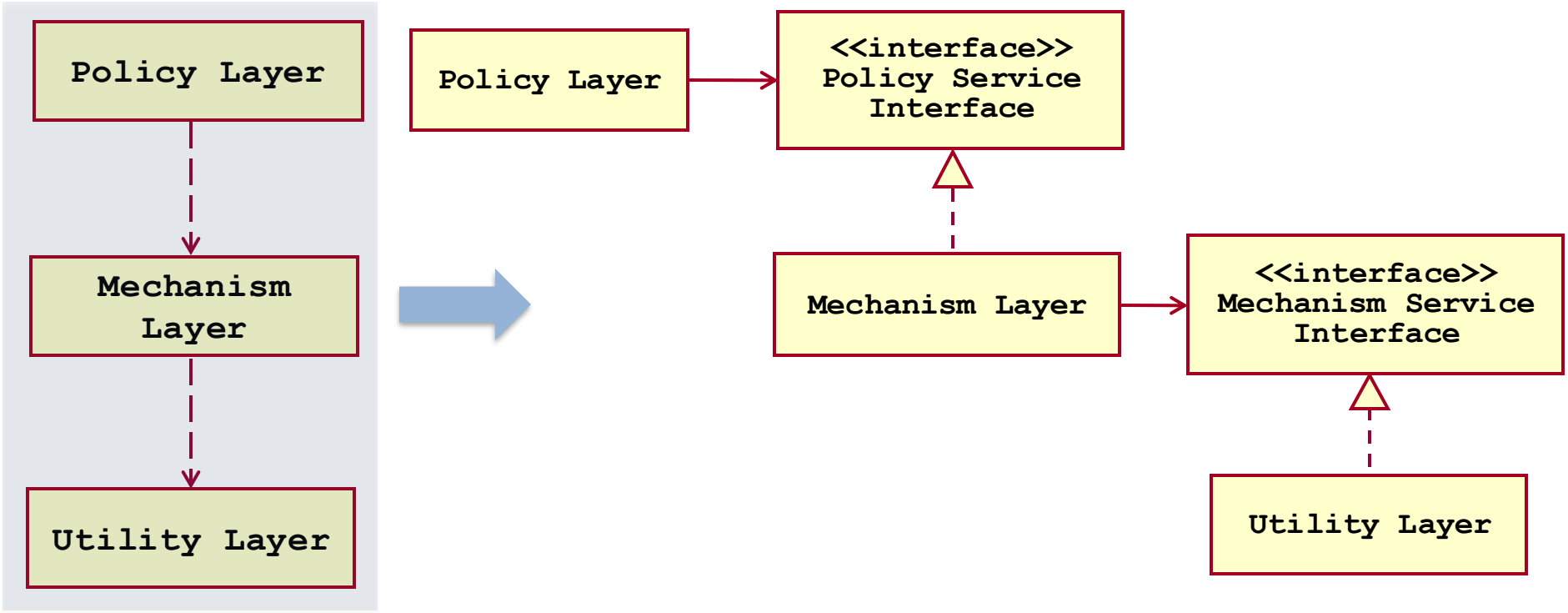


Dependency Inversion Principle

✧ The Dependency Inversion Principle is a principle that describes how to handle dependencies and write loosely coupled software.

> ***I. High-level modules should not depend on low-level modules. Both should depend on abstractions.***
> ***II. Abstractions should not depend on details. Details should depend on abstractions.***
>
> **R. Martin, 1996**

Dependency Inversion Principle

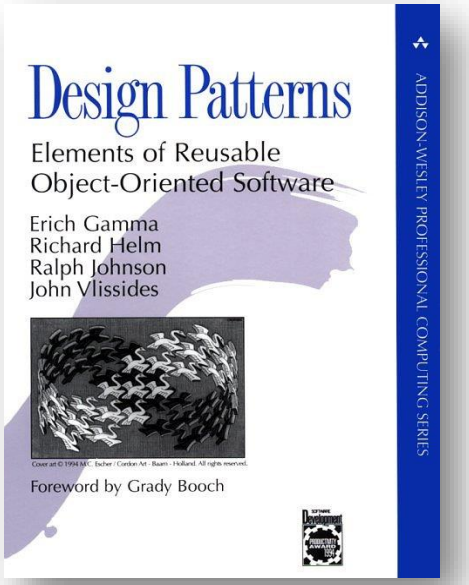


Design Patterns

Design Patterns

- A design pattern is a time-tested solution to a common software problem.
  - Each pattern is like a blueprint that you can customize to solve a particular design problem in your code.
- Also, patterns have become a vocabulary for talking about a design.

Gang of Four



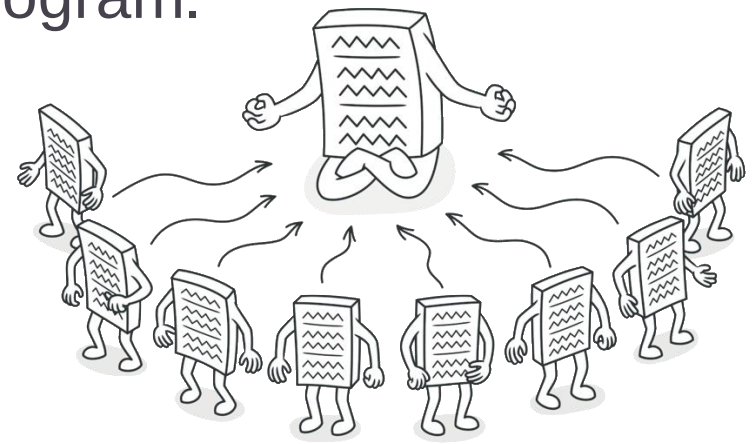
Classification of Patterns

- Purpose reflects what a pattern does.
  - Creational patterns concern the process of object creation.
  - Structural patterns deal with the composition of classes or objects.
  - Behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility.
- Scope specifies whether the pattern applies primarily to classes or to objects.

By Purpose				
		Creational	Structural	Behavioral
By Scope	Class	• Factory Method	• Adapter (class)	• Interpreter • Template Method
	Object	• Abstract Factory • Builder • Prototype • Singleton	• Adapter (object) • Bridge • Composite • Decorator • Façade • Flyweight • Proxy	• Chain of Responsibility • Command • Iterator • Mediator • Memento • Observer • State • Strategy • Visitor

Singleton

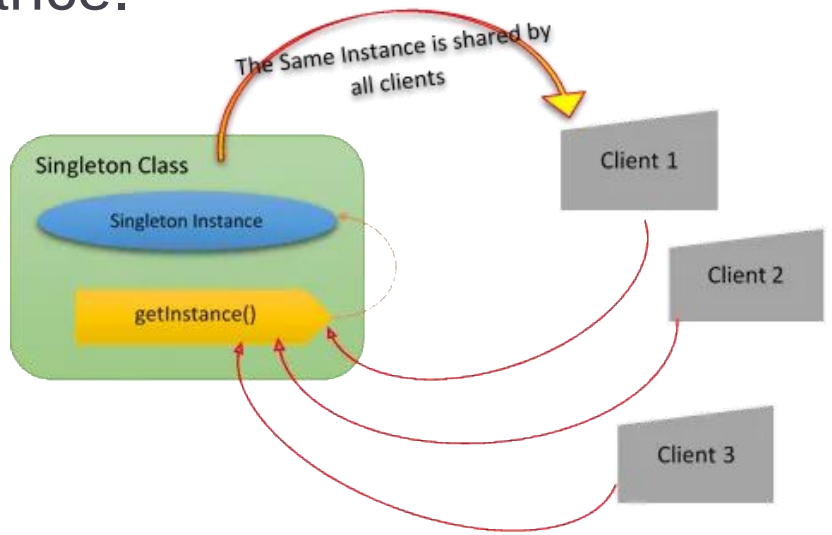
- Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.
- Use the Singleton pattern when a class in your program should have just a single instance available to all clients; for example, a single database object shared by different parts of the program.



Implementing Singleton

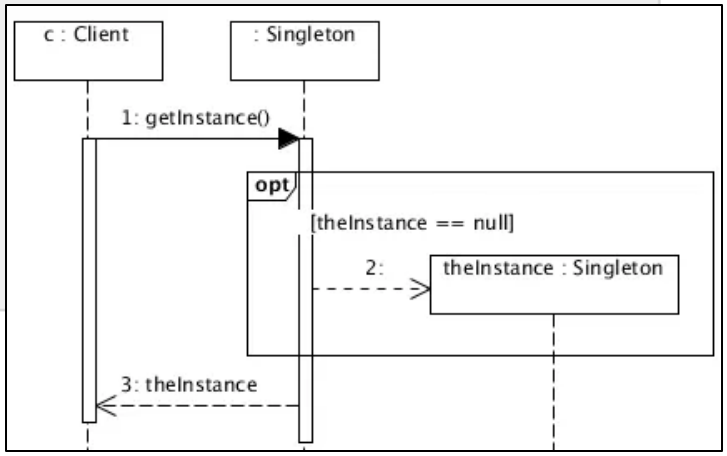
- ✧ Make constructor(s) private so that they can not be called from outside.
- ✧ Declare a single static private instance of the class.
- ✧ Write a public getInstance() or similar method that allows access to the single instance.

Singleton
- singleton : Singleton
- Singleton() + getInstance() : Singleton



Implementing Singleton

```
public class ClassicSingleton {

   private static ClassicSingleton instance = null;
   private ClassicSingleton() {
      // Exists only to defeat instantiation.
   }

   public static ClassicSingleton getInstance() {
      if(instance == null) {
         instance = new ClassicSingleton();
      }
      return instance;
   }
}
```

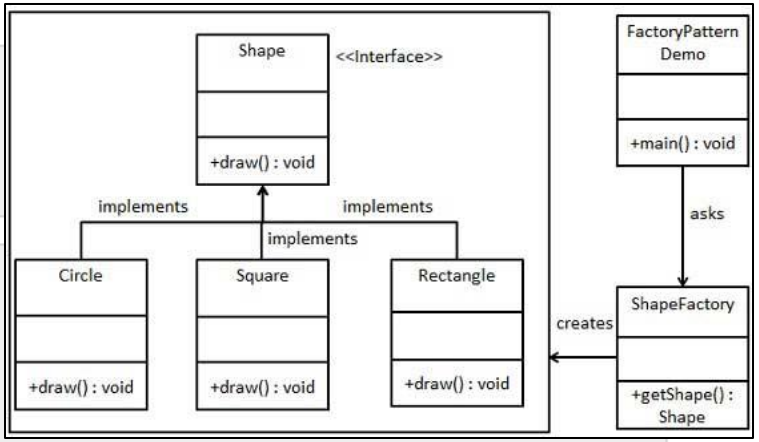


Factory Method

- 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.
  - Factory refers to a class whose job is to easily create and return instances of other classes.
- In Factory Method, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

Implementing Factory Method

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




```
public class Rectangle implements Shape {

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

```
public class Square implements Shape {

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

Implementing Factory Method

```
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;
   }
}
```

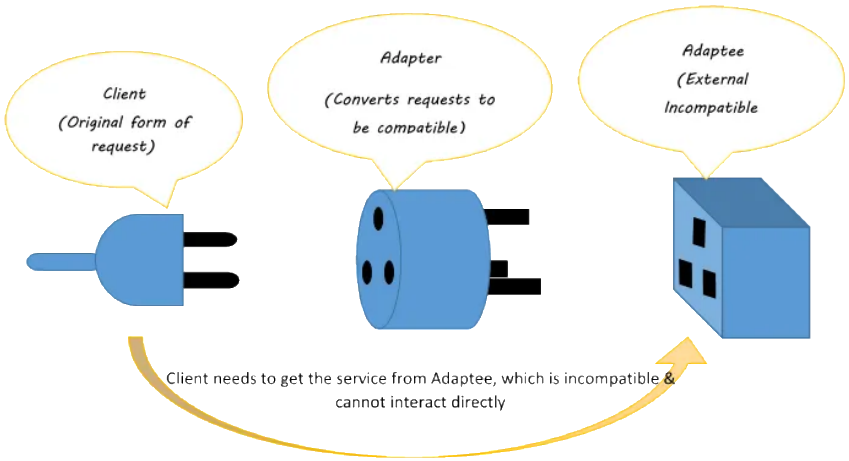
Implementing Factory Method

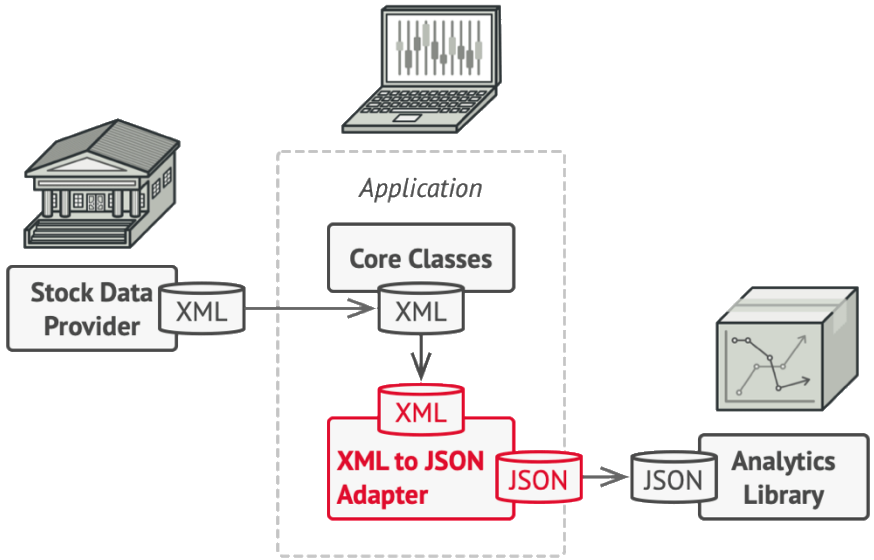
```
public class FactoryPatternDemo {

   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();
   }
}
```

```
Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.
```

Adapter

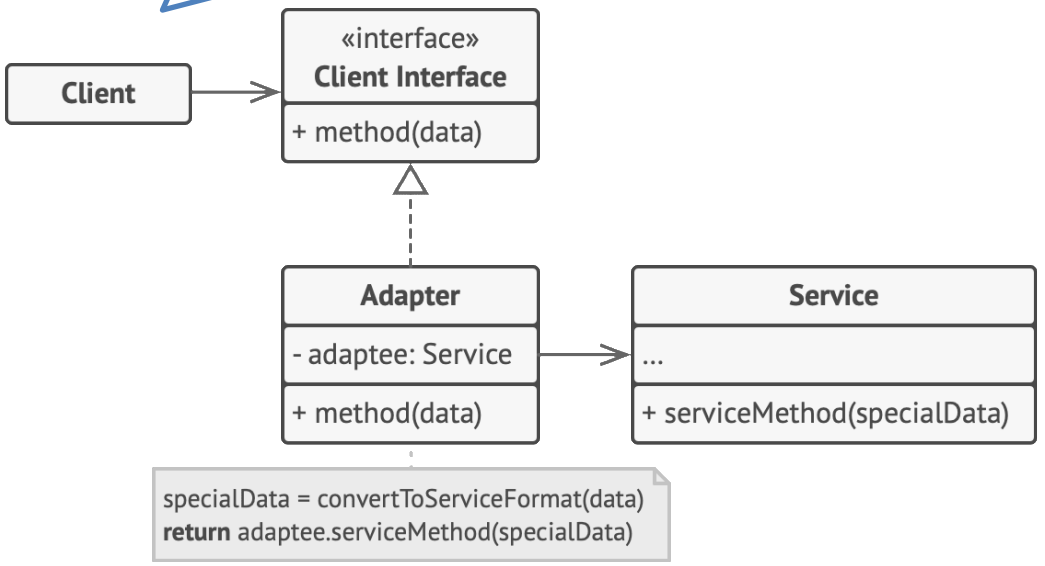
- Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate.
- Use the Adapter class when you want to use some existing class, but its interface isn't compatible with the rest of your code.

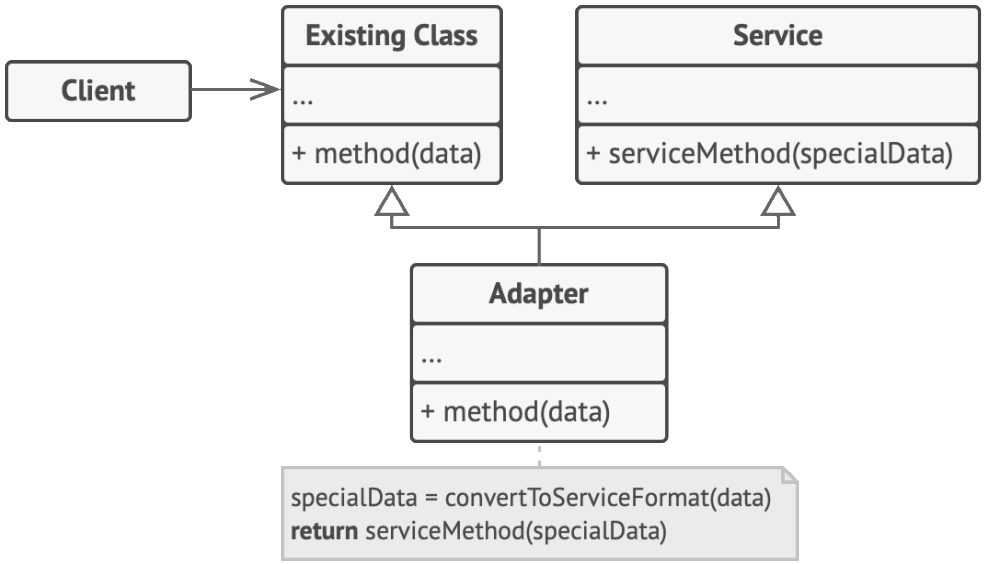




Implementing Adapter

Object adapter
This implementation uses the object composition principle: the adapter implements the interface of one object and wraps the other one.

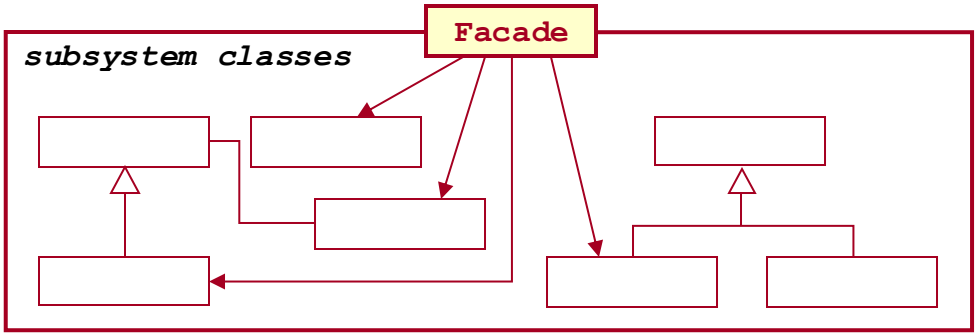


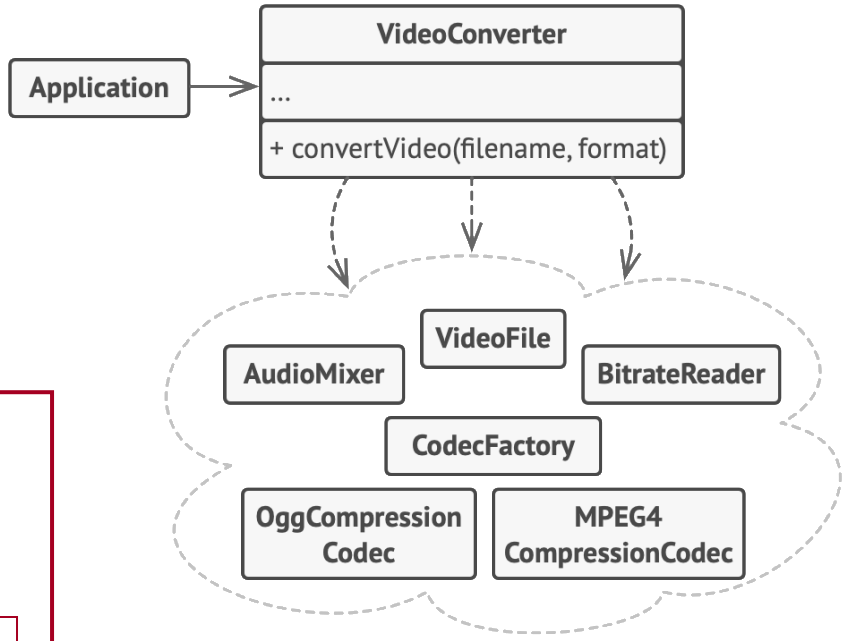


Class adapter
This implementation uses inheritance: the adapter inherits interfaces from both objects at the same time.

Facade

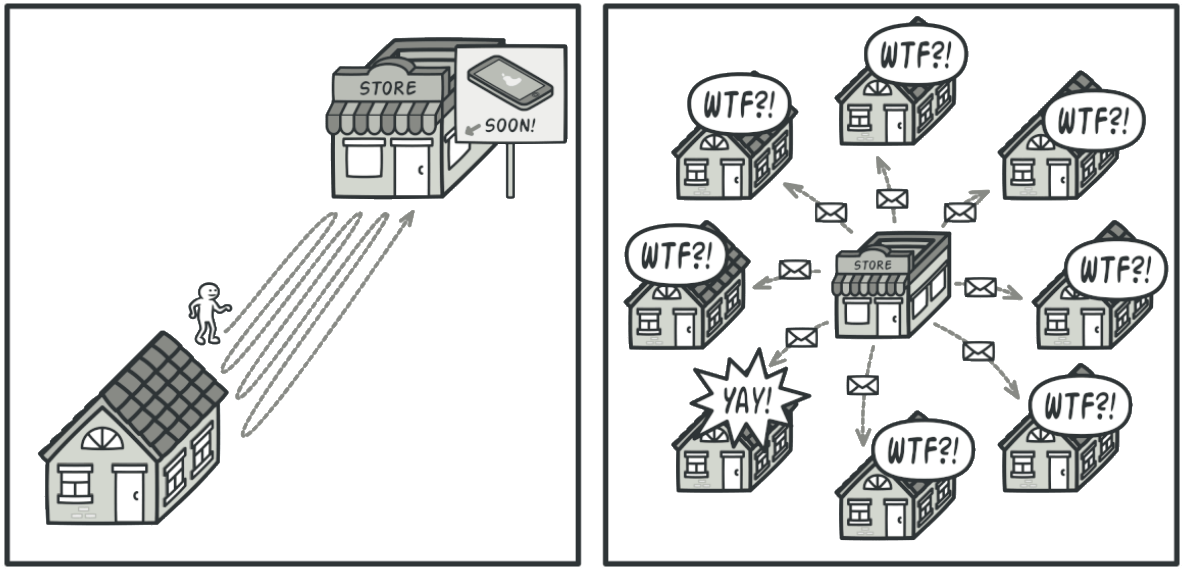
- Facade is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.
  - This pattern involves a single class which provides simplified methods required by client and delegates calls to methods of existing system classes.





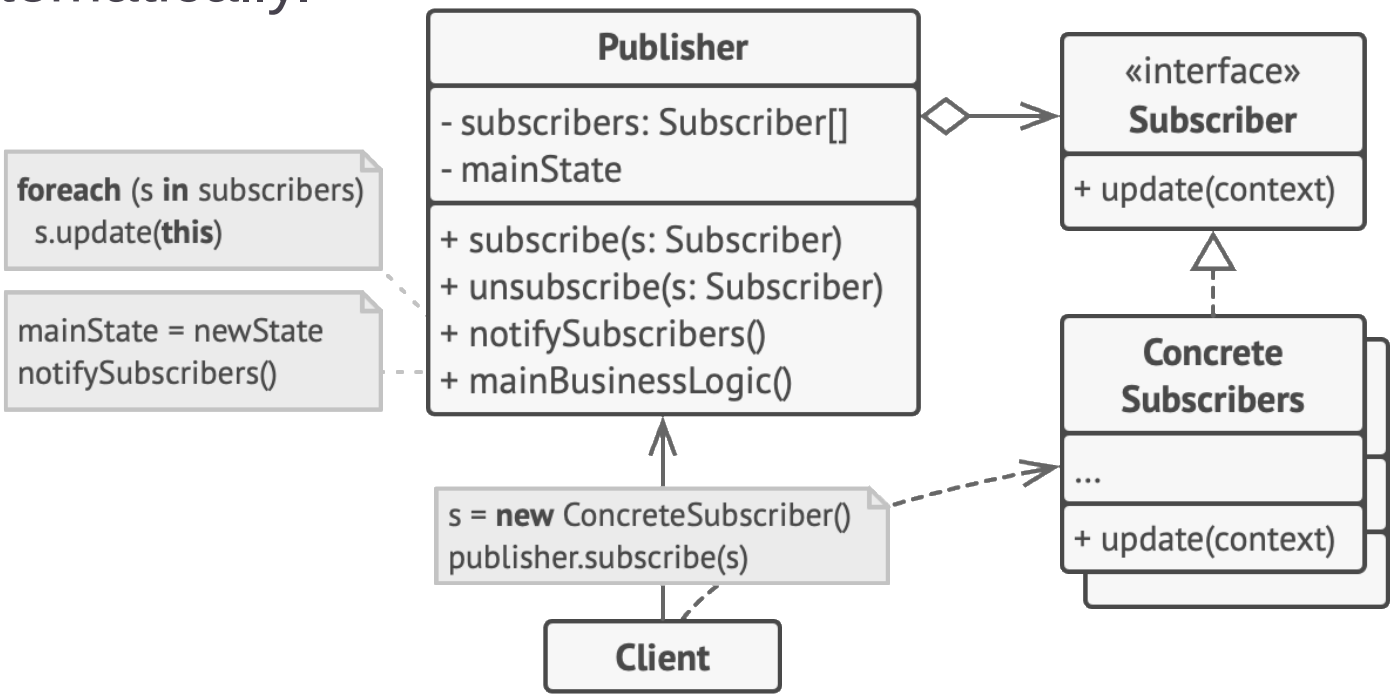
Observer

- Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.



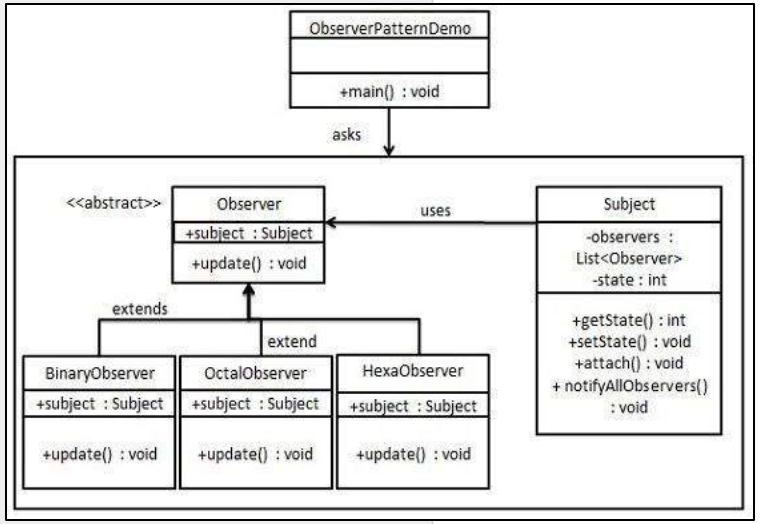
Implementing Observer

- Observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically.



Implementing Observer

```
import java.util.ArrayList;
import java.util.List;

public class Subject {

   private List<Observer> observers = new ArrayList<Observer>();
   private int state;

   public int getState() {
      return state;
   }

   public void setState(int state) {
      this.state = state;
      notifyAllObservers();
   }

   public void attach(Observer observer){
      observers.add(observer);
   }

   public void notifyAllObservers(){
      for (Observer observer : observers) {
         observer.update();
      }
   }
}
```



Implementing Observer

```
public abstract class Observer {
   protected Subject subject;
   public abstract void update();
}
```

```
public class BinaryObserver extends Observer{

   public BinaryObserver(Subject subject){
      this.subject = subject;
      this.subject.attach(this);
   }

   @Override
   public void update() {
      System.out.println( "Binary String: " + Integer.toBinaryString( subje
   }
}
```

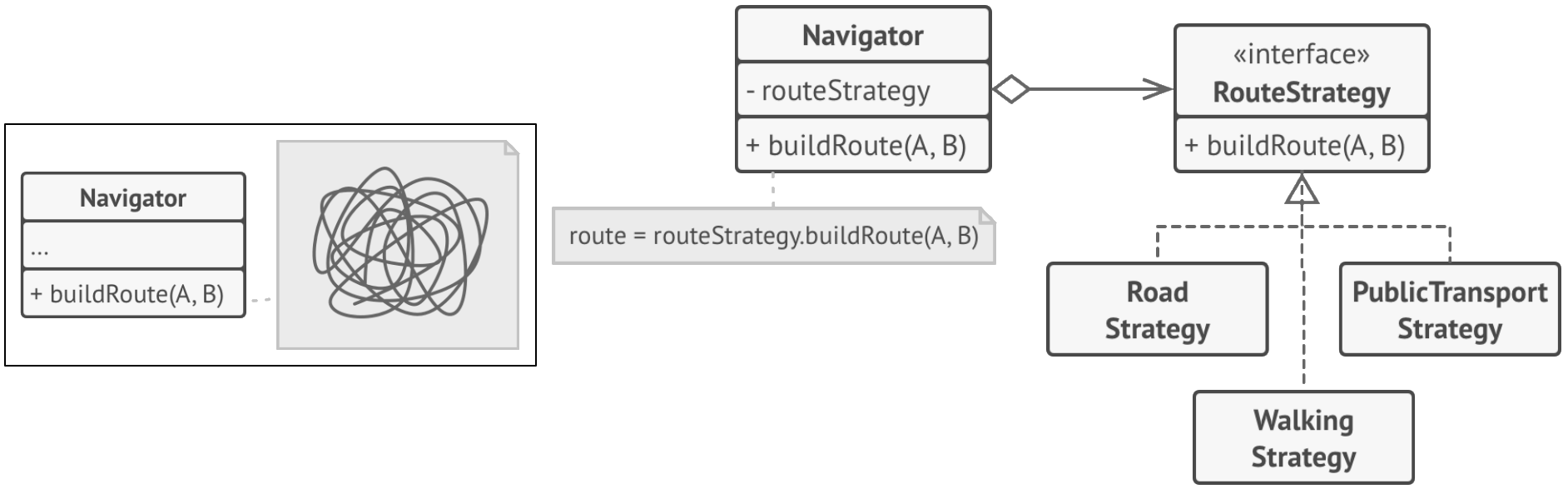
Implementing Observer

```
public class ObserverPatternDemo {
   public static void main(String[] args) {
      Subject subject = new Subject();

      new HexaObserver(subject);
      new OctalObserver(subject);
      new BinaryObserver(subject);

      System.out.println("First state change: 15");
      subject.setState(15);
      System.out.println("Second state change: 10");
      subject.setState(10);
   }
}
```

```
First state change: 15
Hex String: F
Octal String: 17
Binary String: 1111
Second state change: 10
Hex String: A
Octal String: 12
Binary String: 1010
```

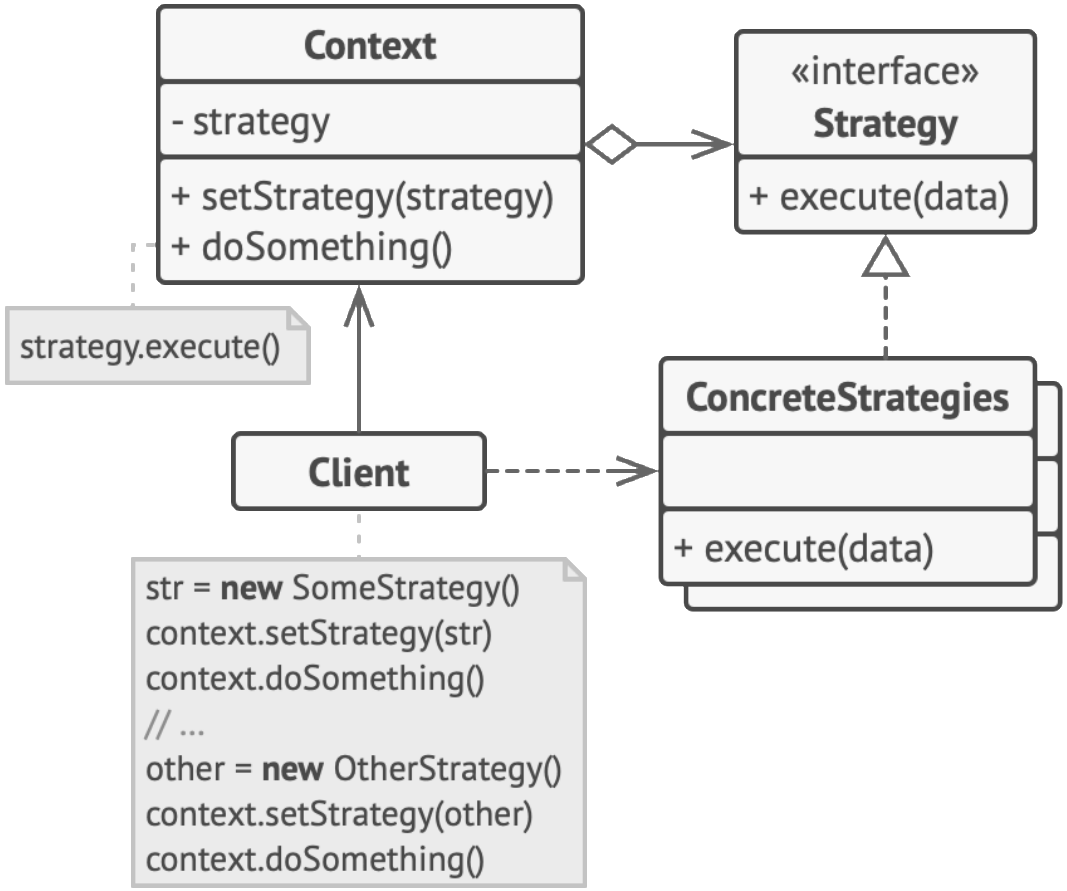
Strategy

- ✧ Strategy is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.



Implementing Strategy

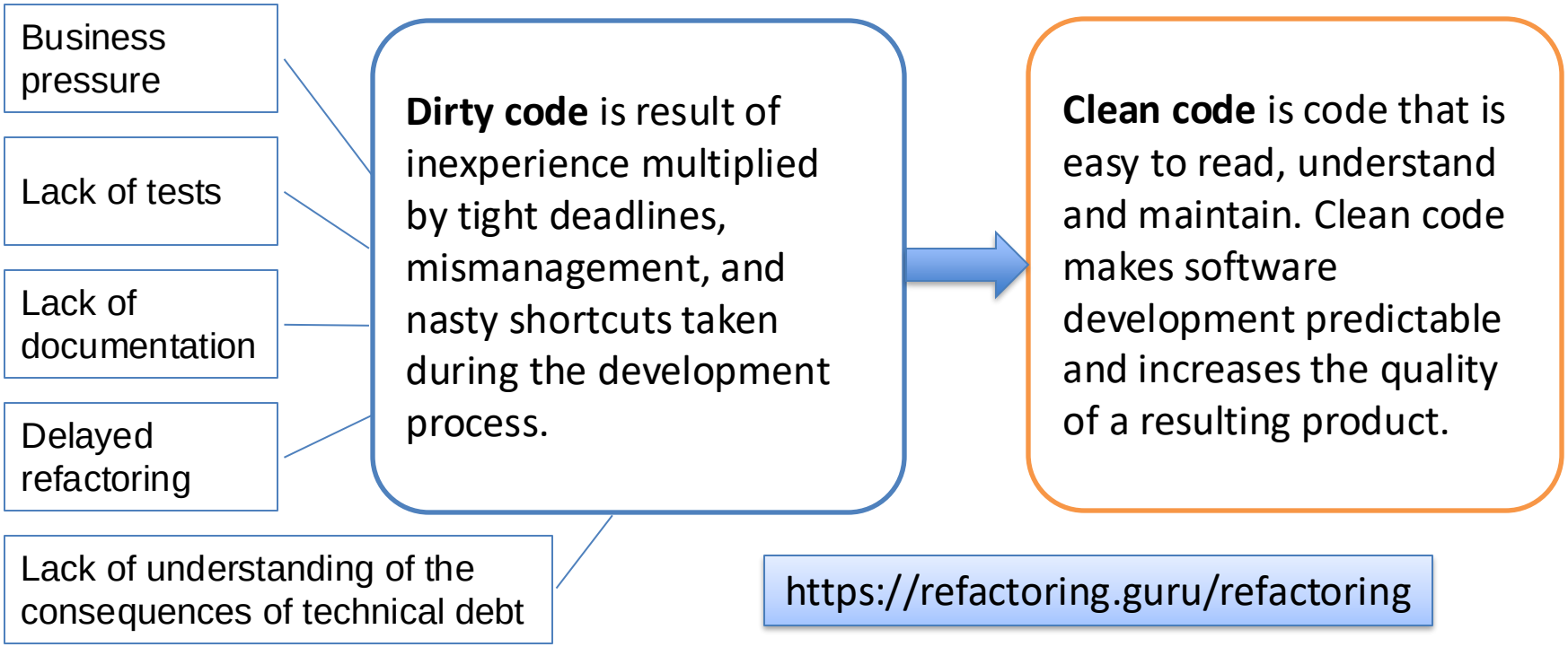
- In Strategy pattern, we create objects which represent various strategies and a context object whose behavior varies as per its strategy object. The strategy object changes the executing algorithm of the context object.



Refactoring

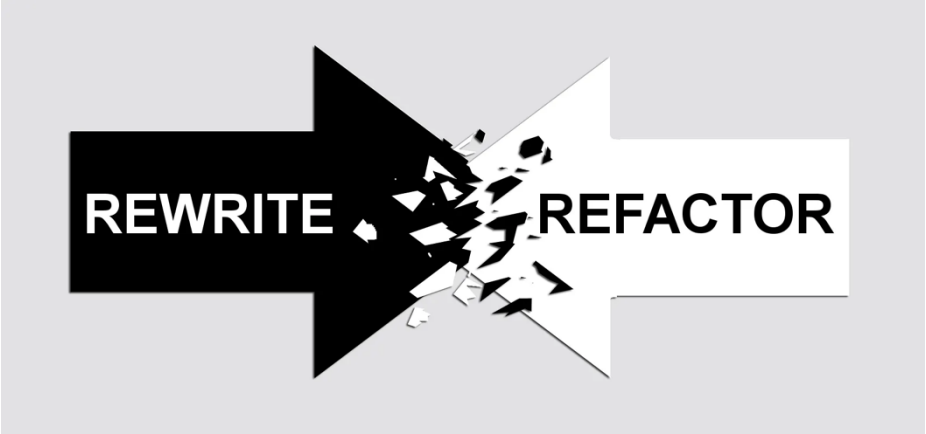
Refactoring

- Refactoring is a systematic process of improving code without creating new functionality that can transform a mess into clean code and simple design.



When to Refactor

- ✧ Rule of Three
  - When you're doing something for the first time, just get it done.
  - When you're doing something similar for the second time, cringe at having to repeat but do the same thing anyway.
  - When you're doing something for the third time, start refactoring.
- ✧ When adding a feature
- ✧ When fixing a bug
- ✧ During a code review



Code Smells

✧ Bloaters

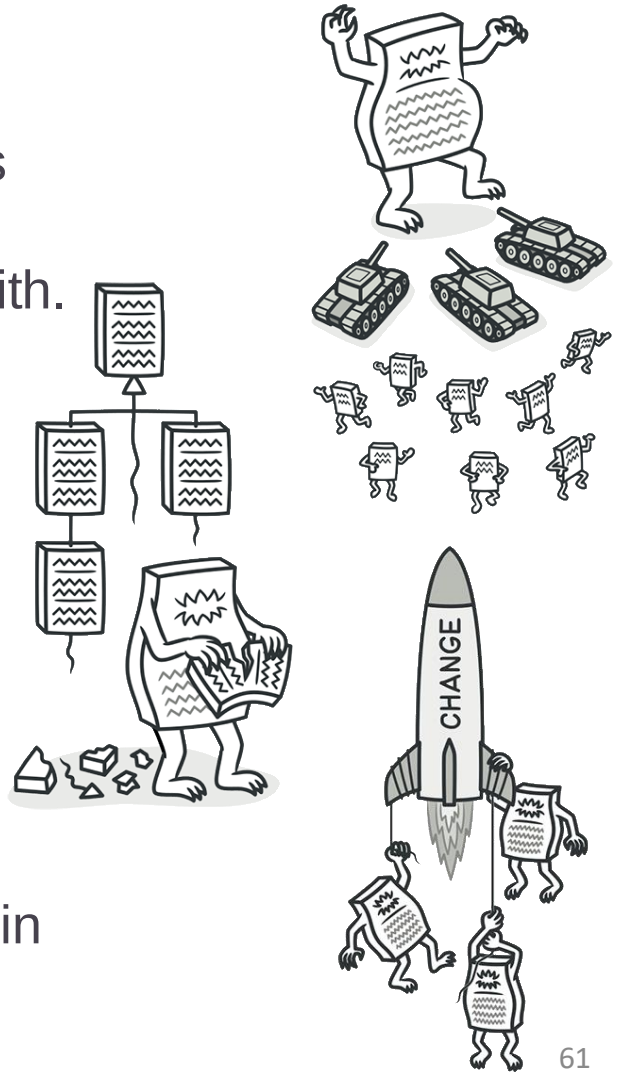
- Bloaters are code, methods and classes that have increased to such gargantuan proportions that they are hard to work with.

✧ Object-Orientation Abusers

- All these smells are incomplete or incorrect application of object-oriented programming principles.

✧ Change Preventers

- These smells mean that if you need to change something in one place in your code, you have to make many changes in other places too.



Code Smells

✧ Dispensables

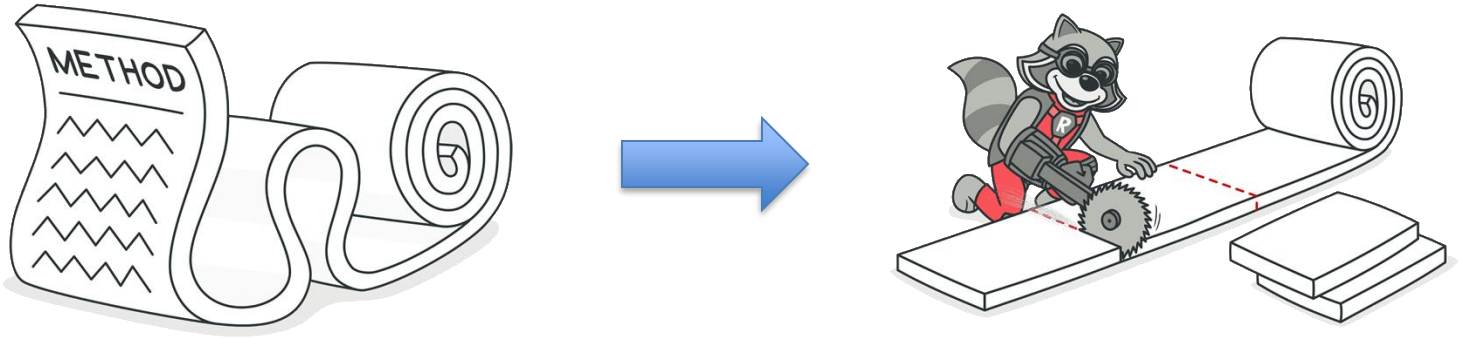
- A dispensable is something pointless and unneeded whose absence would make the code cleaner, more efficient and easier to understand.

✧ Couplers

- All the smells in this group contribute to excessive coupling between classes or show what happens if coupling is replaced by excessive delegation.

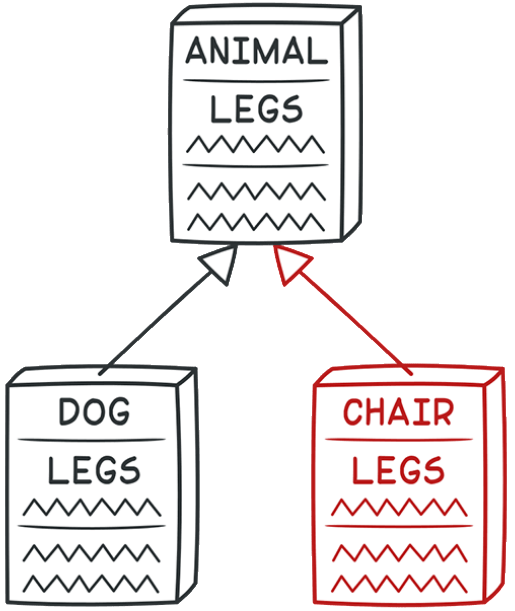
Bloaters - Long Method

- A method contains too many lines of code. Generally, any method longer than ten lines should make you start asking questions.
- Treatment
  - As a rule of thumb, if you feel the need to comment on something inside a method, you should take this code and put it in a new method. And if the method has a descriptive name, nobody will need to look at the code to see what it does.



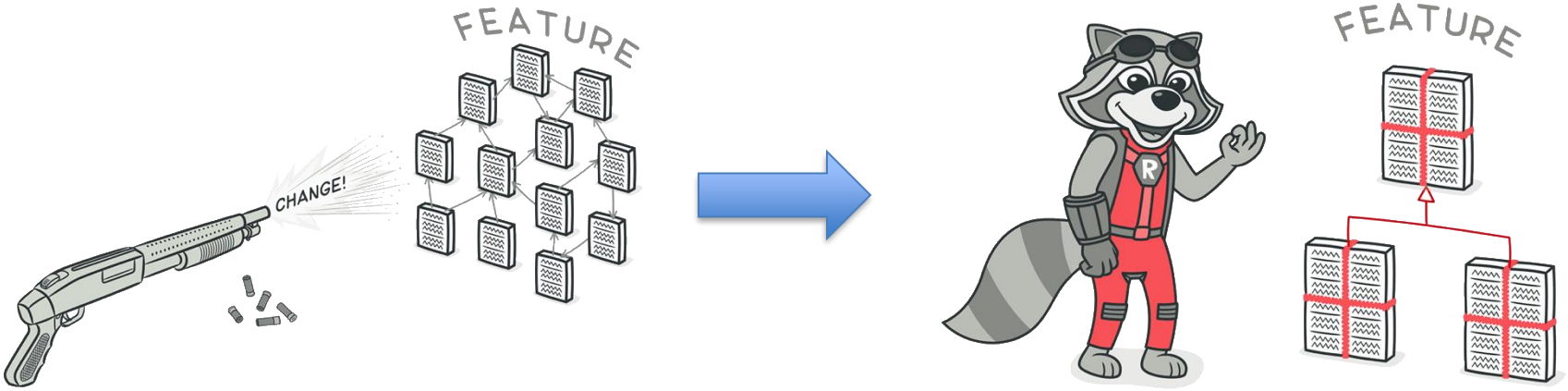
OO Abusers - Refused Bequest

- If a subclass uses only some of the methods and properties inherited from its parents, the hierarchy is off-kilter. The unneeded methods may simply go unused or be redefined and give off exceptions.
- Treatment
  - If inheritance makes no sense and the subclass really does have nothing in common with the superclass, eliminate inheritance.
  - If inheritance is appropriate, get rid of unneeded fields and methods in the subclass.



Change Preventers - Shotgun Surgery

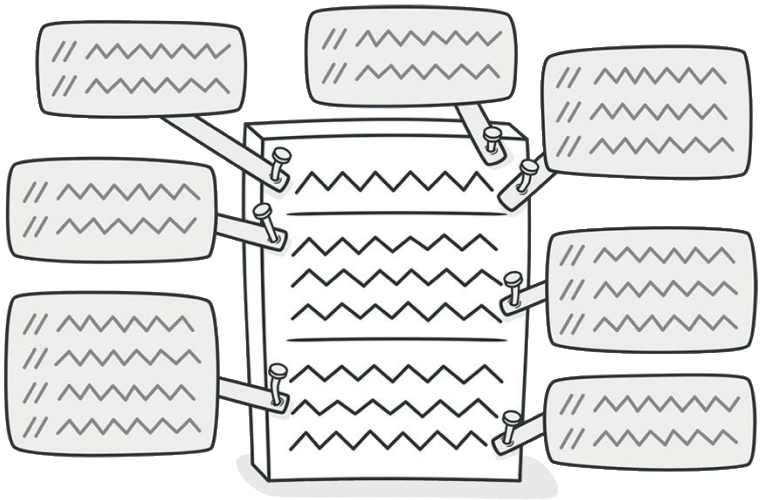
- ✧ Making any modifications requires that you make many small changes to many different classes.
- ✧ Treatment
  - Move existing class behaviors into a single class. If there's no class appropriate for this, create a new one.
  - If moving code to the same class leaves the original classes almost empty, try to get rid of these now-redundant classes.



Dispensables - Comments

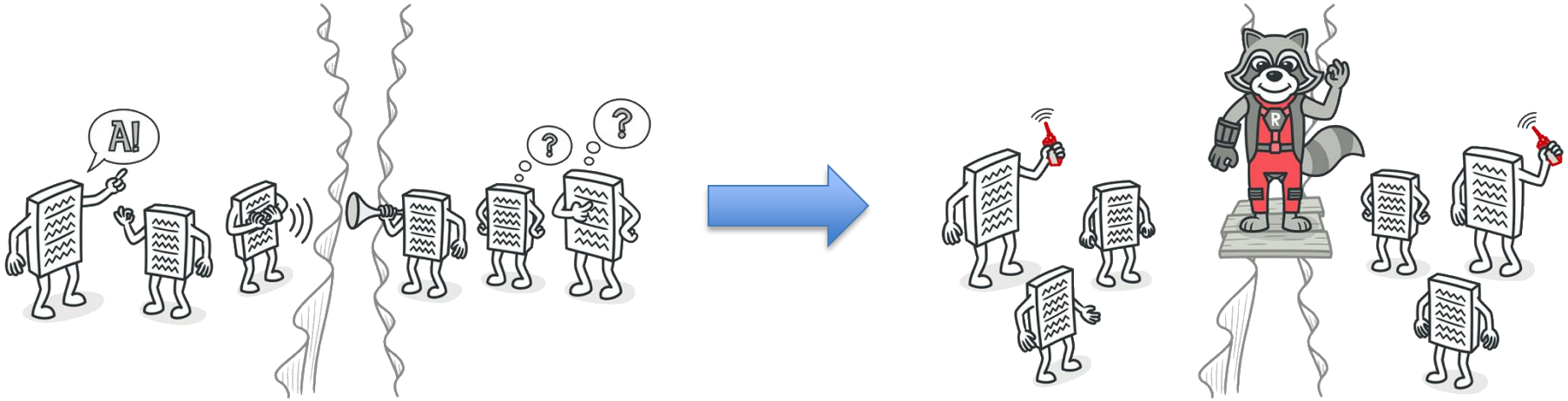
- A method is filled with explanatory comments.
- Comments are usually created with the best of intentions, when the author realizes that his or her code isn't intuitive or obvious. In such cases, comments are like a deodorant masking the smell of fishy code that could be improved.

The best comment is a good name for a method or class.



Couplers - Message Chains

- A message chain occurs when a client requests another object, that object requests yet another one, and so on.
- These chains mean that the client is dependent on navigation along the class structure. Any changes in these relationships require modifying the client.



Refactoring Techniques

- Refactoring techniques describe actual refactoring steps. Most refactoring techniques have their pros and cons.
  - Composing Methods
  - Moving Features between Objects
  - Organizing Data
  - Simplifying Conditional Expressions
  - Simplifying Method Calls
  - Dealing with Generalization

Extract Method

✧ Problem

- You have a code fragment that can be grouped together.

```
void printOwing() {
  printBanner();

  // Print details.
  System.out.println("name: " + name);
  System.out.println("amount: " + getOutstanding());
}
```

Extract Method

✧ Solution

- Move this code to a separate new method (or function) and replace the old code with a call to the method.

```
void printOwing() {
  printBanner();
  printDetails(getOutstanding());
}

void printDetails(double outstanding) {
  System.out.println("name: " + name);
  System.out.println("amount: " + outstanding);
}
```

Extract Variable

✧ Problem

- You have an expression that's hard to understand.

```
void renderBanner() {
  if ((platform.toUpperCase().indexOf("MAC") > -1) &&
       (browser.toUpperCase().indexOf("IE") > -1) &&
        wasInitialized() && resize > 0 )
  {
    // do something
  }
}
```

Extract Variable

✧ Solution

- Place the result of the expression or its parts in separate variables that are self-explanatory.

```
void renderBanner() {
  final boolean isMacOs = platform.toUpperCase().indexOf("MAC")
  final boolean isIE = browser.toUpperCase().indexOf("IE") > -1;
  final boolean wasResized = resize > 0;

  if (isMacOs && isIE && wasInitialized() && wasResized) {
    // do something
  }
}
```

Replace Magic Number with Symbolic Constant

- ✧ Problem
  - Your code uses a number that has a certain meaning to it.
- ✧ Solution
  - Replace this number with a constant that has a human-readable name explaining the meaning of the number.

```
double potentialEnergy(double mass, double height) {
  return mass * height * 9.81;
}
```

```
static final double GRAVITATIONAL_CONSTANT = 9.81;

double potentialEnergy(double mass, double height) {
  return mass * height * GRAVITATIONAL_CONSTANT;
}
```

Replace Nested Conditional with Guard Clauses

✧ Problem

- You have a group of nested conditionals and it's hard to determine the normal flow of code execution.

✧ Solution

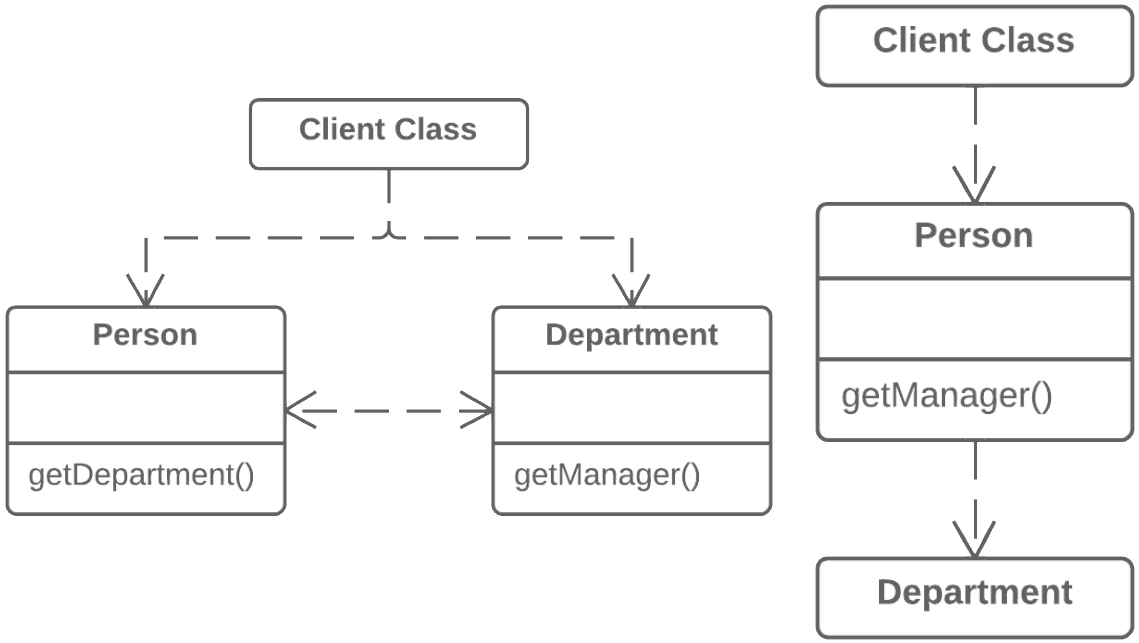
- Isolate all special checks and edge cases into separate clauses and place them before the main checks. Ideally, you should have a "flat" list of conditionals, one after the other.

```
public double getPayAmount() {
  double result;
  if (isDead){
    result = deadAmount();
  }
  else {
    if (isSeparated){
      result = separatedAmount();
    }
    else {
      if (isRetired){
        result = retiredAmount();
      }
      else{
        result = normalPayAmount();
      }
    }
  }
  return result;
}
```

```
public double getPayAmount() {
  if (isDead){
    return deadAmount();
  }
  if (isSeparated){
    return separatedAmount();
  }
  if (isRetired){
    return retiredAmount();
  }
  return normalPayAmount();
}
```

Hide Delegate

- ✧ Problem
  - The client gets object B from a field or method of object A. Then the client calls a method of object B.
- ✧ Solution
  - Create a new method in class A that delegates the call to object B. Now the client doesn't know about, or depend on, class B.

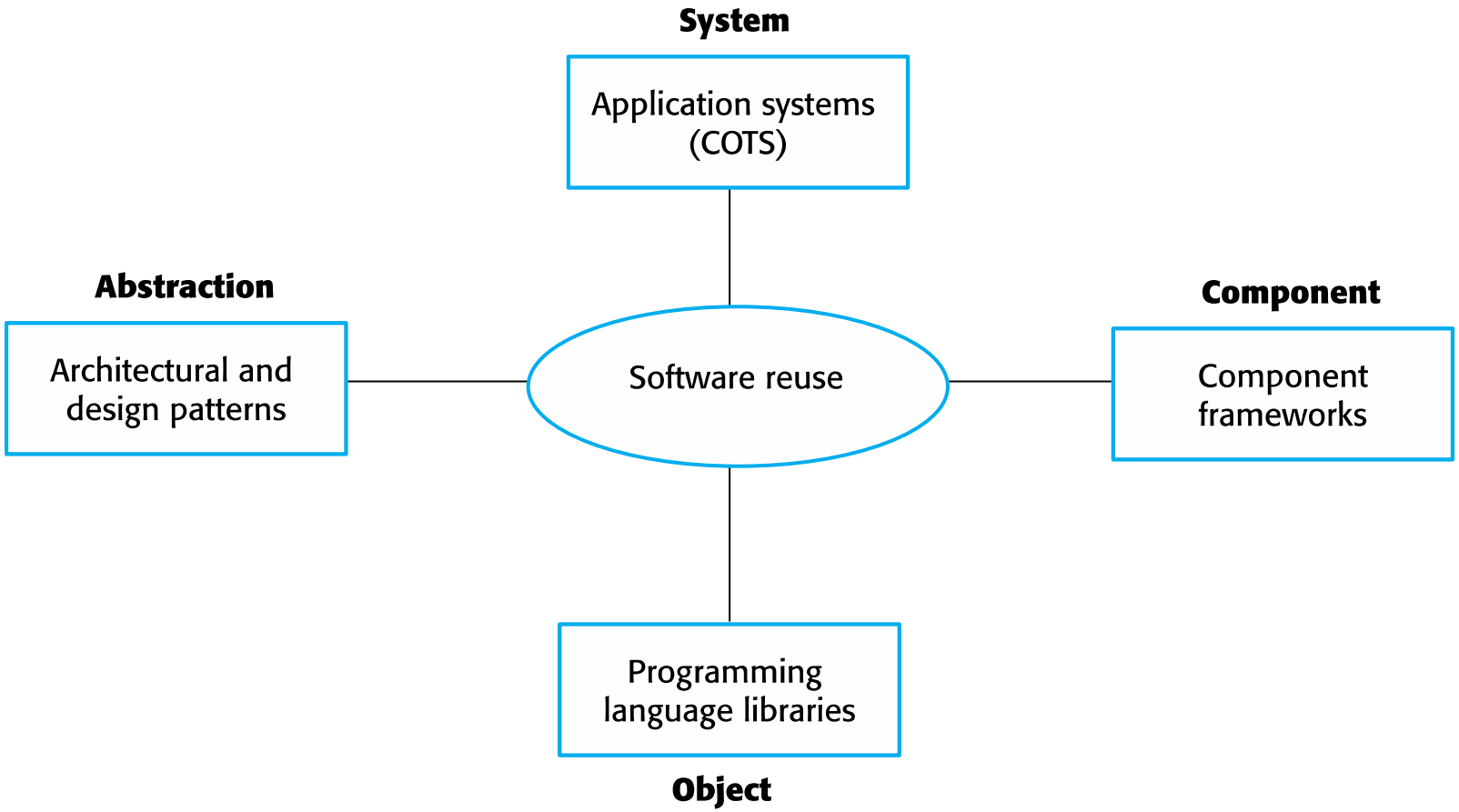


Implementation Issues

Implementation Issues

- Focus here is not on programming, although this is obviously important, but on other implementation issues that are often not covered in programming texts:
  - Reuse Most modern software is constructed by reusing existing components or systems. When you are developing software, you should make as much use as possible of existing code.
  - Configuration management During the development process, you have to keep track of the many different versions of each software component in a configuration management system.
  - Open source development Open-source development is an approach to software development in which the source code of a software system is published and volunteers are invited to participate in the development process.

Reuse


System
Application systems (COTS)
Abstraction
Architectural and design patterns
Software reuse
Component
Component frameworks
Programming language libraries
Object

Reuse Levels

四个等级要背

- The abstraction level
  - At this level, you don't reuse software directly but use knowledge of successful abstractions in the design of your software.
- The object level
  - At this level, you directly reuse objects from a library rather than writing the code yourself.
- The component level
  - Components are collections of objects and object classes that you reuse in application systems.
- The system level
  - At this level, you reuse entire application systems.

Reuse Costs

- The costs of the time spent in looking for software to reuse and assessing whether or not it meets your needs.
- Where applicable, the costs of buying the reusable software. For large off-the-shelf systems, these costs can be very high.
- The costs of adapting and configuring the reusable software components or systems to reflect the requirements of the system that you are developing.
- The costs of integrating reusable software elements with each other (if you are using software from different sources) and with the new code that you have developed.

Configuration Management

- Configuration management is the name given to the general process of managing a changing software system.
- The aim of configuration management is to support the system integration process so that all developers can access the project code and documents in a controlled way, find out what changes have been made, and compile and link components to create a system.

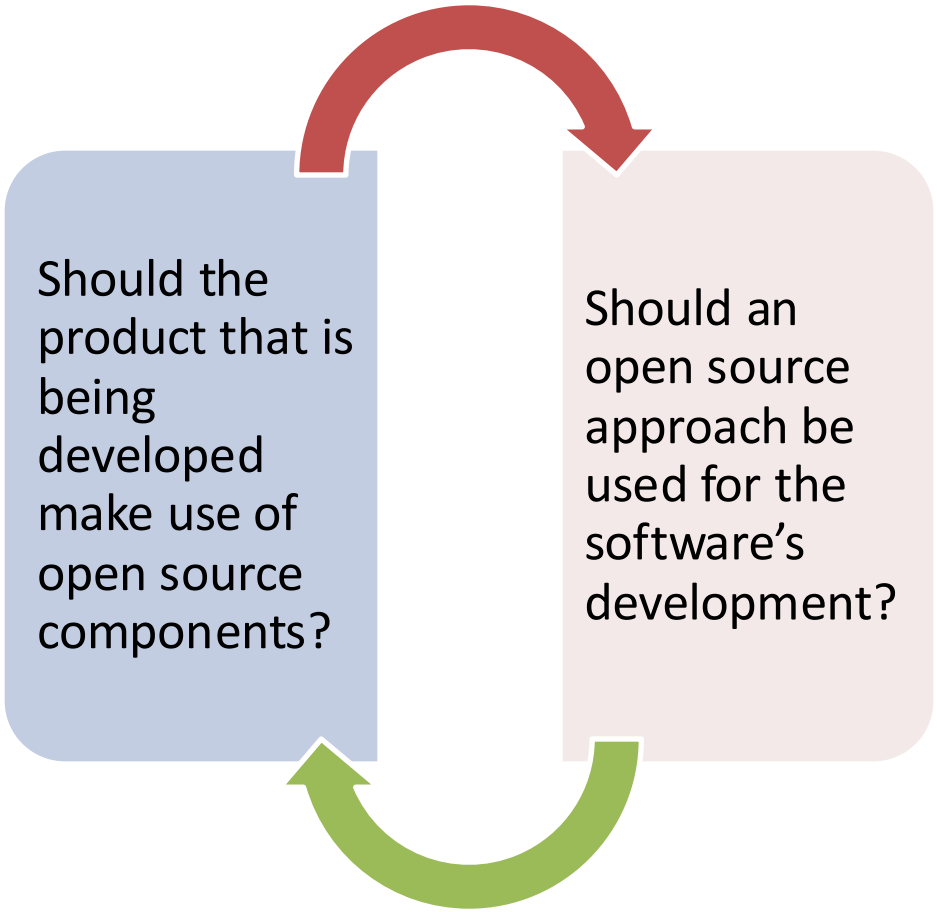
Configuration Management Activities

- Version management: where support is provided to keep track of the different versions of software components. Version management systems include facilities to coordinate development by several programmers.
  - Subversion
  - Git
- Problem tracking: where support is provided to allow users to report bugs and other problems, and to allow all developers to see who is working on these problems and when they are fixed.
  - Bugzilla

Open Source Development

- Open source development is an approach to software development in which the source code of a software system is published and volunteers are invited to participate in the development process.
- Its roots are in the Free Software Foundation (www.fsf.org), which advocates that source code should not be proprietary but rather should always be available for users to examine and modify as they wish.
  - Linux
  - Firefox
  - MySQL

Open Source Issues



Open Source Licensing

- A fundamental principle of open-source development is that source code should be freely available, this does not mean that anyone can do as they wish with that code.
  - The GNU General Public License (GPL). This is a so-called 'reciprocal' license that means that if you use open source software that is licensed under the GPL license, then you must make that software open source.
  - The GNU Lesser General Public License (LGPL) is a variant of the GPL license where you can write components that link to open source code without having to publish the source of these components.
  - The Berkley Standard Distribution (BSD) License. This is a non-reciprocal license, which means you are not obliged to re-publish any changes or modifications made to open source code. You can include the code in proprietary systems that are sold.

Summary

Key Points

- Software design and implementation are inter-leaved activities. The level of detail in the design depends on the type of system and whether you are using a plan-driven or agile approach.
- The process of object-oriented design includes activities to design the system architecture, identify objects in the system, describe the design using different object models and document the component interfaces.
- SOLID is an acronym that represents five separate software design principles.
- Refactoring is a systematic process of improving code without creating new functionality that can transform a mess into clean code and simple design.

Key Points

- Design patterns are typical solutions to common problems in software design. Each pattern is like a blueprint that you can customize to solve a particular design problem in your code.
- When developing software, you should always consider the possibility of reusing existing software, either as components, services or complete systems.
- Configuration management is the process of managing changes to an evolving software system. It is essential when a team of people are cooperating to develop software.
- Open source development involves making the source code of a system publicly available. This means that many people can propose changes and improvements to the software.