Universitatea Tehnica din Cluj-Napoca
Departament Calculatoare

Programming Techniques in Java

SOLID Principles, Inversion of Control & Frameworks

Sources: Design Principles and Design Patterns by Robert C. Martin, ref

T. Cioara, V. Chifu, C. Pop
2025

Rotting Design

Initial state-of-the-art design that everyone agrees to follow

The design starts to rot due to the hacks solving exceptions when new features must be added

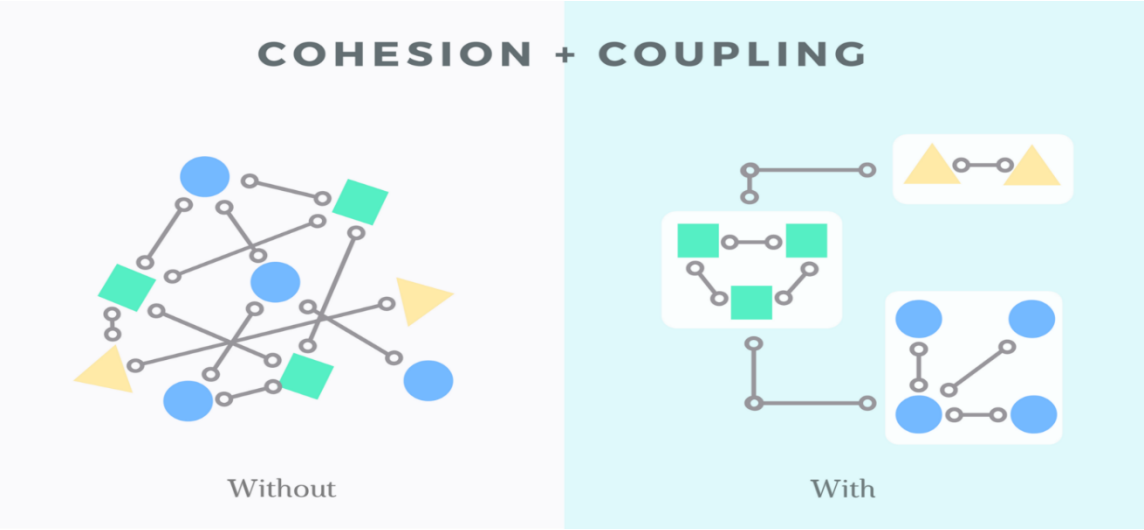
The design does not resemble the initial one, the code is hard to maintain, and we need to redesign and rewrite everything!

Rotting Design

Rigidity	**Difficult to change, even in simple ways.**
Fragility	**Break in strange places every time it is changed.**
Immobility	**Inability to reuse software from other projects**
Viscosity	**Design preserving methods are harder to employ than the hacks in addressing new changes** **Software is slow and inefficient**

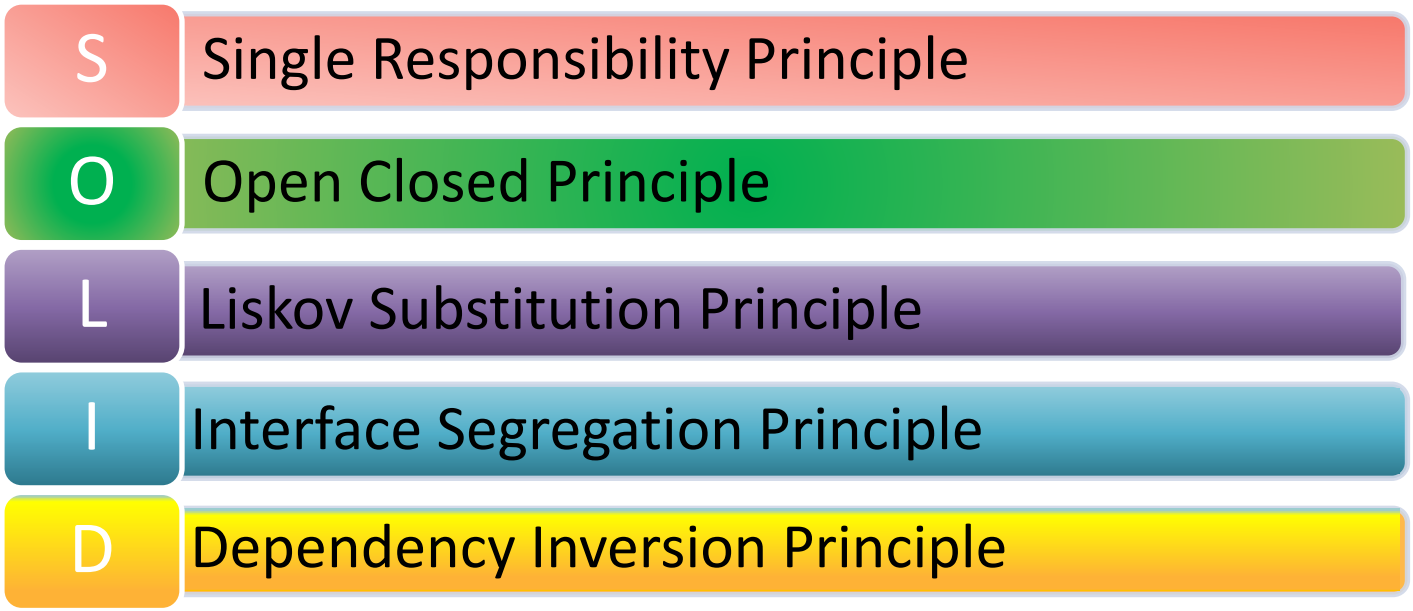
Rotting Design

- **Changing requirements**
  - Not anticipated by the initial design and made quickly
  - Works, but it violates the initial design
  - Accumulation effect
- Dependencies in design and implementation



- **It is well known that requirements will change!**

SOLID Principles



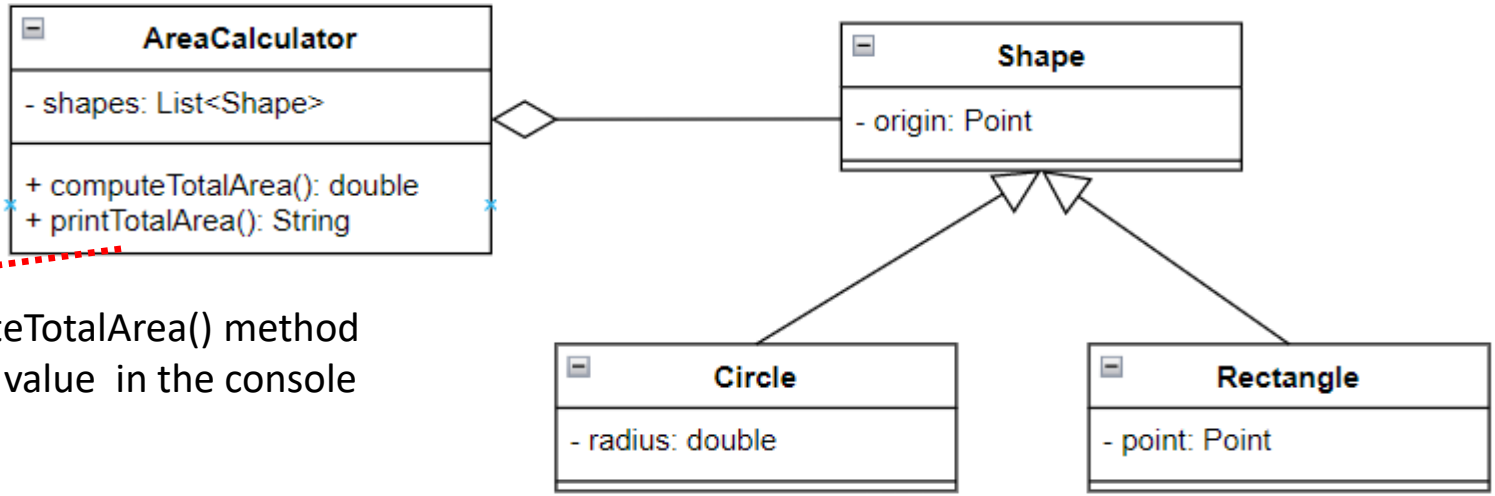
Design Principles and Design Patterns by
Robert C. Martin

Single Responsibility Principle

"A class should have only one responsibility, and it should have only one reason to change"

Advantages

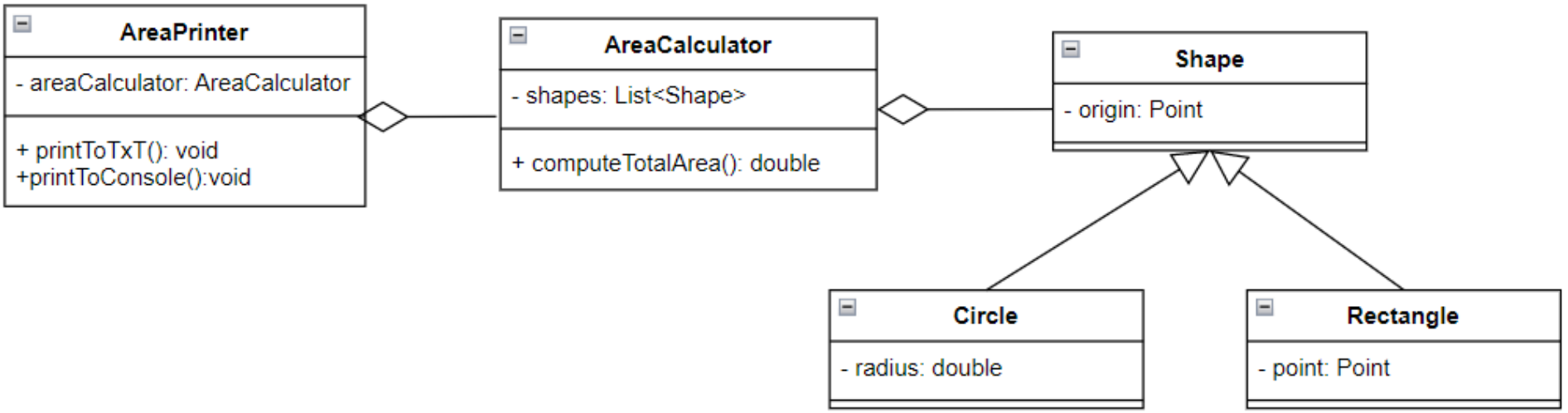
- Fewer test cases
- Fewer dependencies => loose coupling
- Smaller, well-organized classes are easier to search than monolithic ones



Calls the computeTotalArea() method and displays the value in the console

Single Responsibility Principle

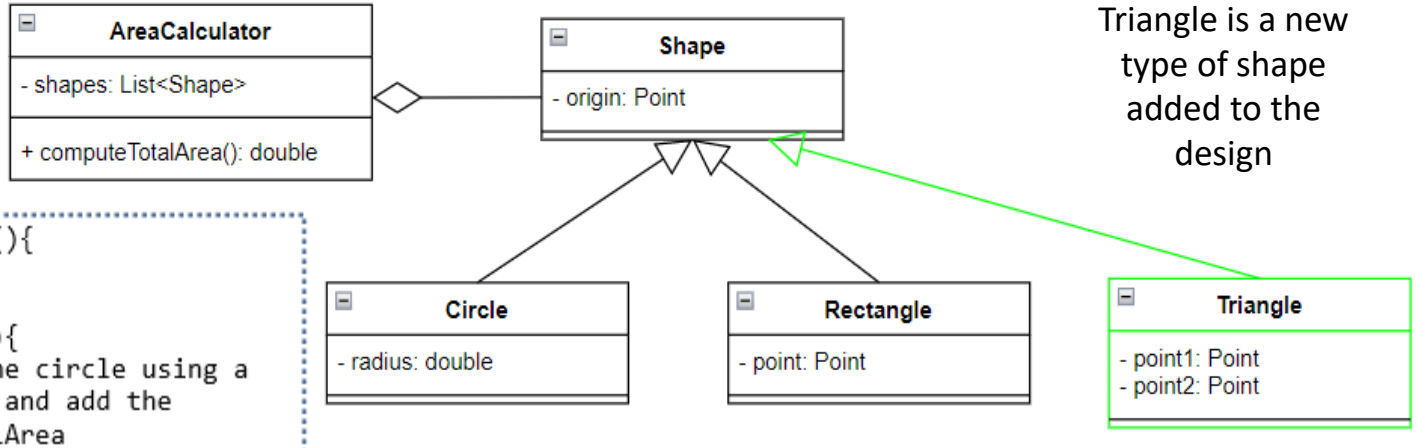
- **Problem**
  - AreaCalculator should be concerned with computing the total area and not with printing it in various formats!
  - Methods should deal only with fields
  - New class responsible for printing the total area in various formats



Open Closed Principle

"A module should be open for extension but closed for modification"

- A class should be extendable without modifying the class itself
- Implementation strategy: **Dynamic Polymorphism, Use Interfaces**

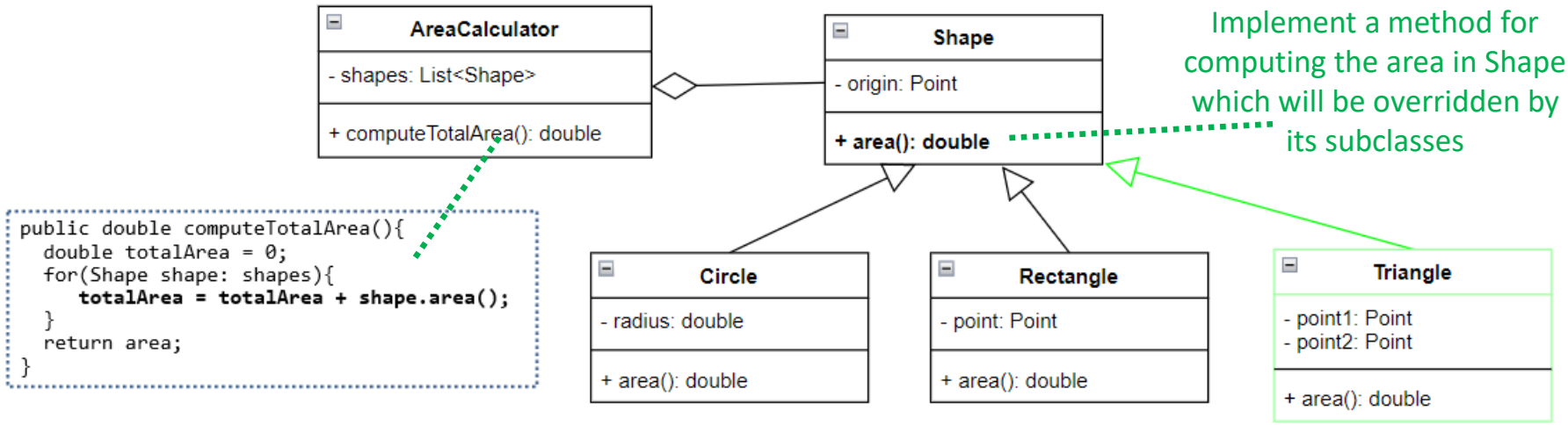


Triangle is a new type of shape added to the design

```
public double computeTotalArea(){
  double totalArea = 0;
  for(Shape shape: shapes){
    if(shape instanceof Circle){
      // compute the area of the circle using a
      // corresponding formula and add the
      // computed area to totalArea
    } else if(shape instanceof Rectangle){
        // compute the area of the circle using a
        // corresponding formula and add the
        // computed area to totalArea
       }
  }
  return area;
}
```

Open Closed Principle

- **Problem**
  - Modified to add new if/else blocks for the new shape



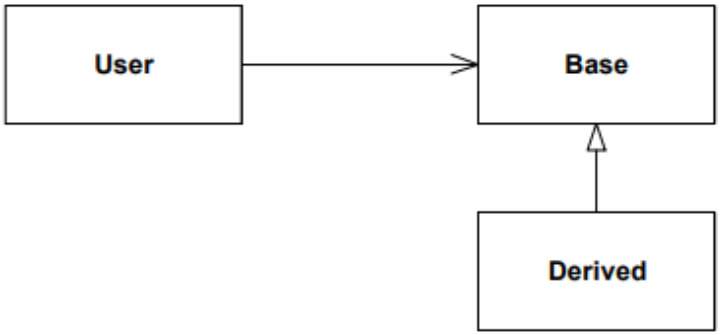
AreaCalculator has been closed for modification but open for extension!

Liskov Substitution Principle

“Subclasses should be substitutable for their base classes”

```
public class Video{
   private String title;
   private int time;
   private int views
   public void hoursPlayed () {…}
   public void playRandomAD () {…}
}
```

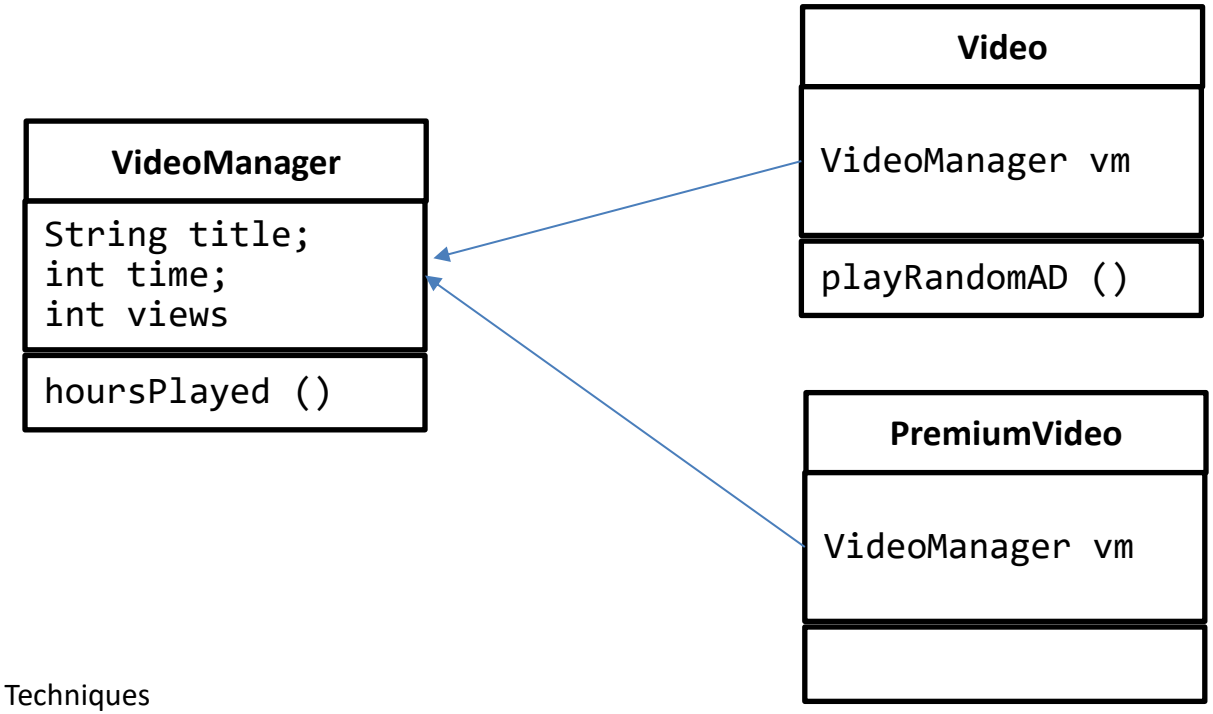
```
public class PremiumVideo extends Video  {

   public void playRandomAD () {
      throw Exception (“no ads in premium”)

  }
```




```
public class User {

   processVideos (List videos)
   {
       for (Video video: videos)
             video.playRandomAD

   }
}
```

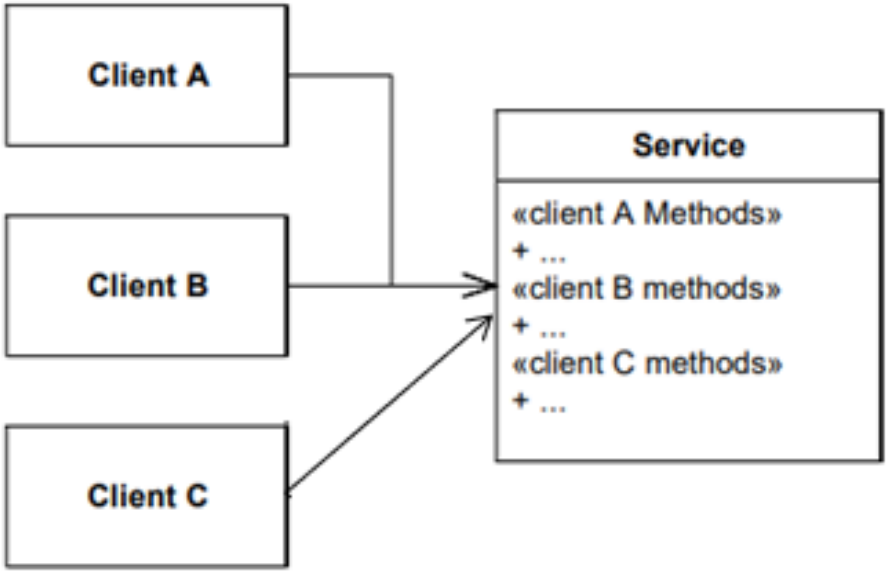
Liskov Substitution Principle

- **Problem**
  - Code may crash on iteration
  - COAD principles not respected
  - Solution use composition as the principle suggest



Interface Segregation Principle

"Many client specific interfaces are better than one general purpose interface"



Fat Service with integrated interfaces

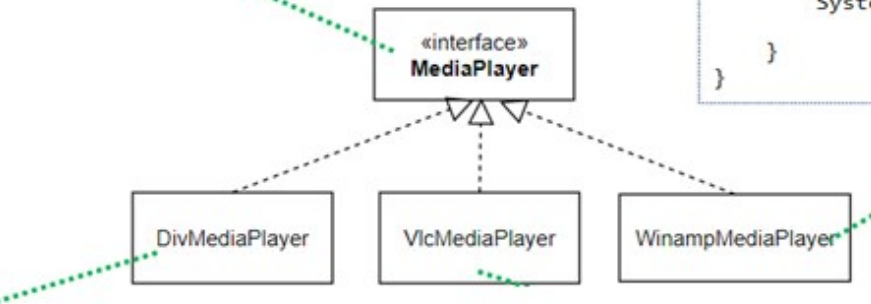
- Whenever a change is made to one of the methods that ClientA calls, ClientB and ClientC may be affected
- Use of interfaces in the previous example?

Interface Segregation Principle

- **Problem**

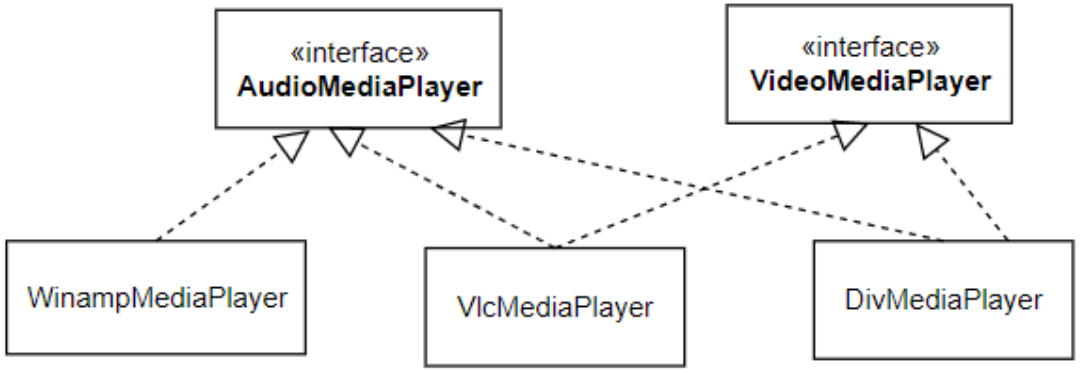
```
public interface MediaPlayer {
    public void playAudio();
    public void playVideo();
}
```

```
public class WinampMediaPlayer implements MediaPlayer {

    // Play video is not supported in Winamp player
    public void playVideo() {
        throw new VideoUnsupportedException();
    }

    @Override
    public void playAudio() {
        System.out.println("Playing audio...");

    }
}
```




```
public class DivMediaPlayer implements MediaPlayer {
    @Override
    public void playAudio() {
        System.out.println(" Playing audio…");
    }

    @Override
    public void playVideo() {
        System.out.println(" Playing video…");
    }
}
```

- **Solution**



Source

Dependency Inversion

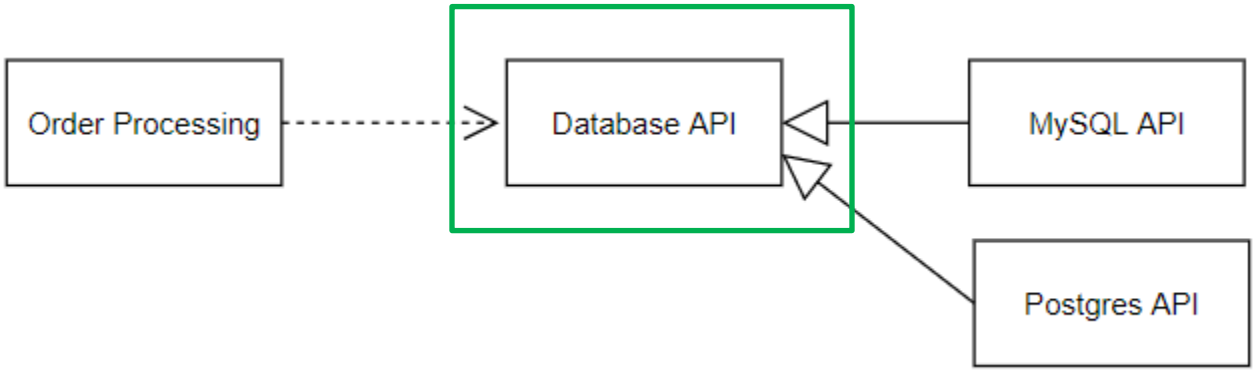
"Depend upon Abstractions. Do not depend upon concretions."

- Every dependency in the design should target an interface, or an abstract class.
- No dependency should target a concrete class.

- **Example** – Problem

What if we want to use another database API?

- Solution



Inversion of Control

IoC as a DP (others are calling it a programming principle)

- Is used to **invert different types of controls** in OO design
- Achieve **loose coupled classes** which make them **better testable, maintainable and extensible**

Types of controls

- Flow of an application,
- Flow of an object creation or
- Flow of dependent object creation and binding the dependencies

IoC alternatives

- Dependency Injection (DI) techniques
- Locator techniques
- Callback techniques

IoC through Dependency Injection

- Examples of having class A use class B

(Case 1) Class A is in control of creation of B instance

```
// Hardcoded dependency
public class A {
    private B  b = new B();
}
```

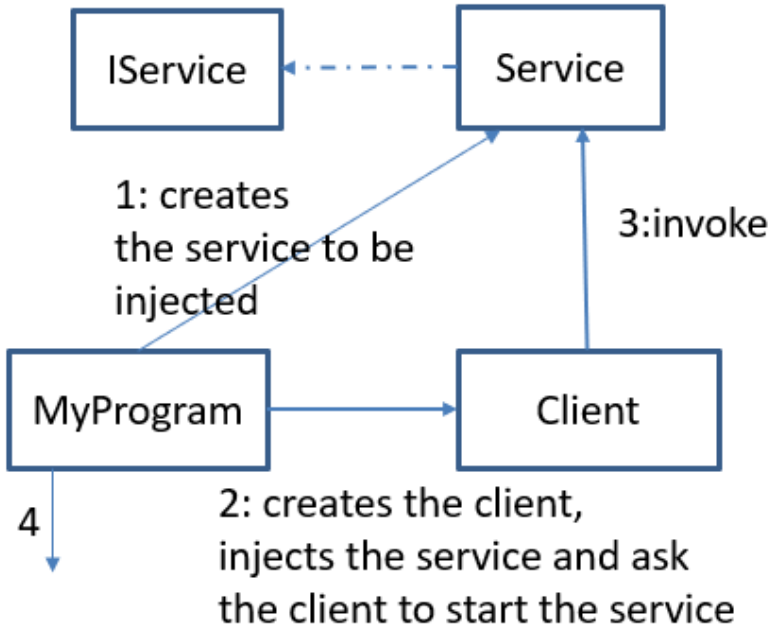
- Class A creates an instance of class B

(Case 2) Someone else is in control to create class B instance for class A

```
// Injected dependency
public class A {
    private B bObj;
    public A (B b) { this.bObj = b; }
}
```

- Class A orders to someone (a factory method) to create an instance of B for it
- A dependency is injected into class A from the outside of A. The dependency can be injected in several ways, like
  - a parameter in the constructor or
  - using “set” type methods

IoC through Dependency Injection

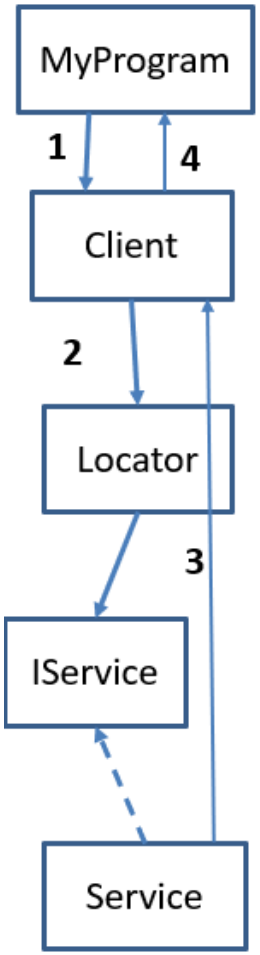



```
public interface IService { void serve(); }

public class Service implements IService {
 public void serve() {
  System.out.println("Service Invoked");
 }}

public class Client {
 private IService service;
 public Client(IService s) {
   this.service = s;
 }
 public void startService() {
   this.service.serve();
 } }

public class MyProgram {
  public static void main(String[] args) {
    var client = new Client(new Service(…));
    client.start();
 }}
```

- The Injection happens in Client, done by MyProgram by passing the Service that implements the IService-Interface into Client's constructor
- All dependencies are assembled by MyProgram (MyProgram should know the types of each IService)

IoC through Service Locator

- Defines a locator for resolving the dependencies within a class

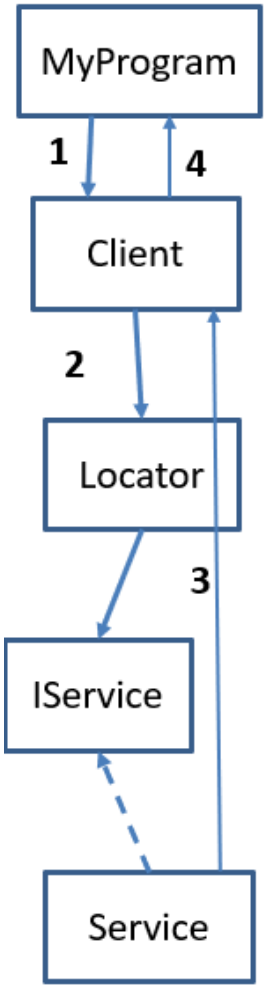



```
public class Client {
  private IService service;
  public Client(…) { this.service = Locator.getService(…);
                     // 2 call locator to find the service
  }
  public void start() {
    this.service.serve();
  }
}
public class MyProgram {
  public static void main(String[] args) {
  // 1 - calls the client indicating certain needs
  var client = new Client(…);
  client.start();
 } }
```

```
public interface IService { void serve(); }
public class Service implements IService {
  public void serve() { System.out.println("Service Invoked"); } }
public class Locator {
   public static IService getService(…) {
      // … 3 - looking for the service and getting it
      return service;  } }
```

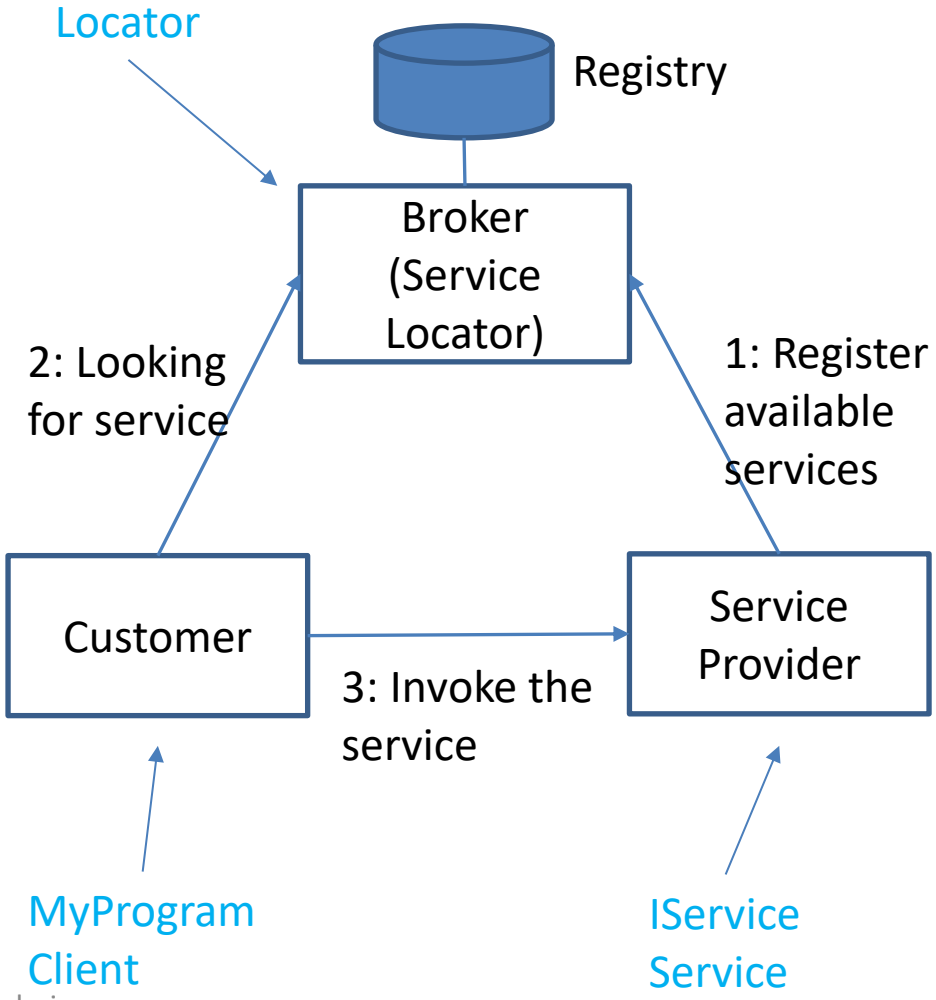
IoC through Service Locator

- Example
  - MyProgram requires its Client to solve a problem for it
    - Find a certain service (local or distributed) to replace a faulty one
    - Find a better performing service to replace a bad one
  - The Client invokes the Locator (acting as a Broker) to identify a specified service
    - The service is specified by parameters that are passed in the getService(…) call
  - The locator finds the proper service and injects it (IoC) in the Client (in the instance variable service)



IoC through Service Locator

- Example – Service Oriented Architecture (SOA)



Disadvantages of DI and SL

- DI and SL solutions are rather hardcoded
  - All the dependent types for the objects in the instantiation process (factory) must be known at compile time
  - Manual management of dependencies is difficult considering the large number of dependent classes

```
public class C3 {
    public void doTask(){ }
}

// C2 depends on C3
public class C2 {
    private C3 oC3;
    public C2(C3 c3){ this.oC3 = c3; }
    public void doTask(){ oC3.doTask(); }
}

// C1 depends on C2
public class C1 {
    private C2 oC2;
    public C1(C2 c2){ this.oC2 = c2; }
    public void doTask(){ oC2.doTask(); }
}
```

```
public class MyProgram {
    public static void main(String[] args){
    // A programmer must manage all
    dependencies
        C3 myC3 = new C3();
        C2 myC2 = new C2(myC3);
        C1 myC1 = new C1(myC2);
        myC1.doTask();
    }
}
```

Disadvantages of DI and SL

- Example – Assume that C2 needs a new dependency: C4

```
public class C2 {
    private C3 oC3;
    private C4 oC4;
    public C2(C3 o3, C4 o4){
        this.oC3 = o3;
        this.oC4 = o4;
    }
    public void doTask(){
        oC3.doTask();
        oC4.doTask();
    }
}
```

```
public class myProgram {
    public static void main(String[]
    args) {
        C4 myC4 = new C4();
        C3 myC3 = new C3();
        C2 myC2 = new C2(myC3, myC4);
        C1 myC1 = new C1(myC2);
        myC1.doTask();
    }
}
```

- The programmer needs to redefine/recompile these dependencies
- Think of real programs with hundreds of dependencies … and the management effort needed to enforce them !!!
- The need to automate the dependencies using IoC (DI) containers
  - E.g.: Spring IoC Container,

IoC through Callback

- Callback function (method)
  - a function (method) f that is passed as argument to another function (method) g
  - f is expected to be executed after a certain event has happened
- Callbacks are needed in event driven programming
  - Objective: inform a class that some work/event has been done/happened in another class
  - Very useful in asynchronous tasks
    - In event handling for processing an event (for example after clicking a button)
- In non-functional Java (pre-Java 8) functions cannot be passed as arguments
  - Callbacks are implemented in Java by using interfaces and lambda expressions

IoC through Callback

```
public interface IF {
  public void f();
}

// callback handler
public class Listener implements IF
    {
  @Override
  public void f() {
    System.out.println(“Executes f
    (callback)");
     // ... do work
  }
}
```

```
// The application - event generator
public class App {
  private IF refIF;
  public void registerCallback(IF ref) {
      refIF = ref;
  }

  public void doWork() {
    // if a certain event happens … execute callback
    System.out.println(“event has happened”);
    refIF.f();
  }

  public static void main(String[] args) {
      Listener lis = new Listener();
      App app = new App();
      app.registerCallback(lis);
      app.doWork();
  }
}
```

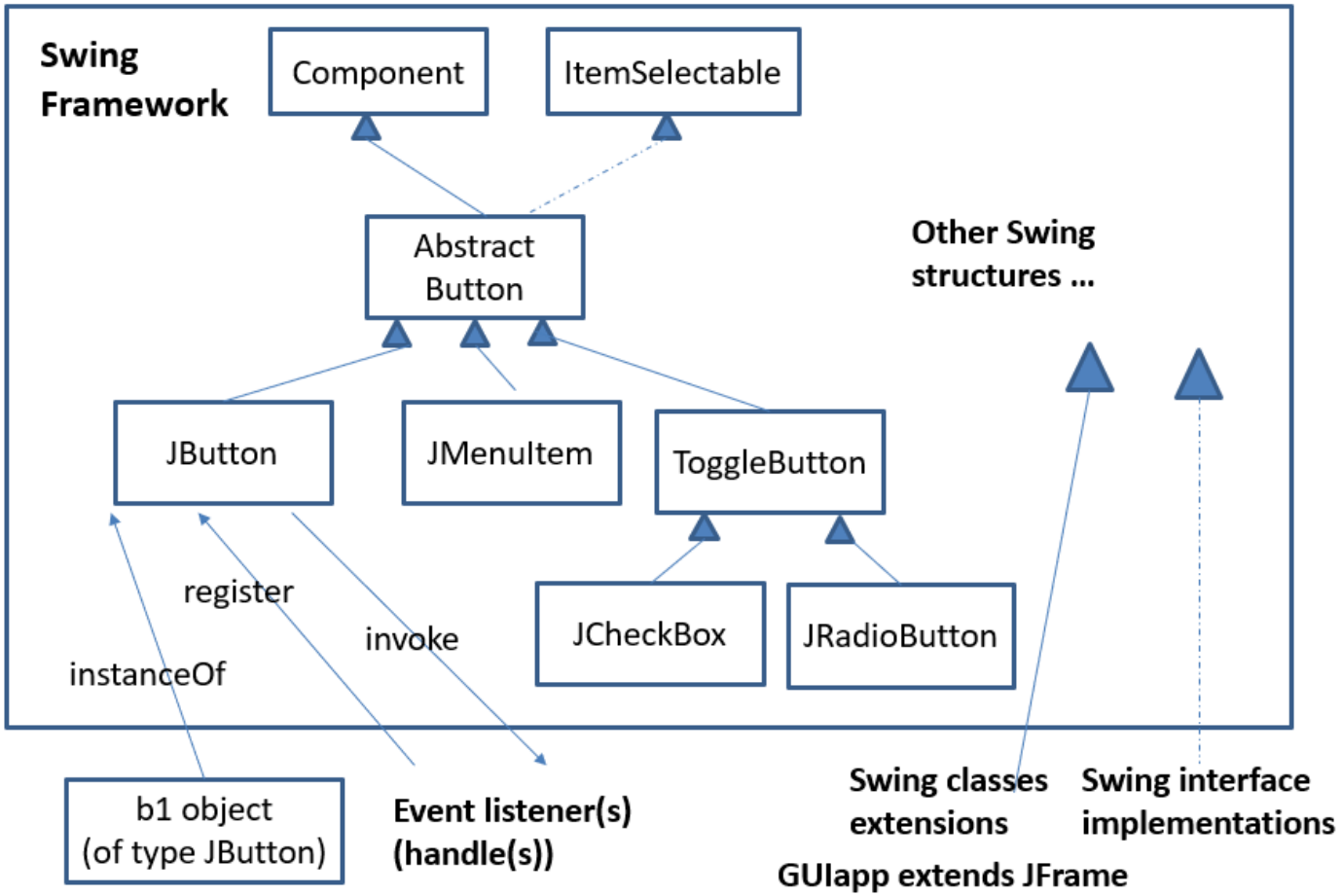
Callback is common in event handling in Java GUI programming

Frameworks

- Frameworks: semi-complete apps
  - Complete applications are developed by inheriting from, and instantiating parameterized framework components
  - Domain-specific functionality
  - Set of cooperative classes, interfaces, DPs
- Exhibit IoC at run-time,
  - Framework determines which objects and methods to invoke in response to events
- Generic and reusable software
  - Captures design decisions common for App domains
  - High level reuse of app domains solutions
  - Leverages on experience of developers and designers
  - Extensibility
    - Uses hook methods that are overwritten by the application to extend the framework

Frameworks

- Example - Domain app using framework Java Swing



Frameworks

- Solves a problem in a sufficient generic way
  - The developer will nor reinvent a solution of his own
  - Focus on the specific aspects of the application
- Influences the application architecture
  - Application's global structure
  - Cooperation between classes and objects
  - (largely) Predefines the control flow
- Layers of cooperative frameworks

Advantages

- Rapid Application Development
- Well designed and well tested generic solutions
- All framework-based applications => same structure => easier to maintain
- High reuse level

Disadvantages

- Not many design decisions to the specific App dev
- Design decisions - taken by the framework designers
  - App designer loses design creativity

Frameworks vs. Design Patterns vs. Library

- Frameworks and DPs
  - DPs - more abstract and must be implemented each time
  - DPs - smaller architectural elements than frameworks
  - DPs - less specialized than frameworks
- Frameworks and libraries
  - Library - code written 3-rd parties to help developers
  - A framework inverts the control of the program
  - The developers call the library where and when they need it (developers are in control)