Lecture: Design Principles

SIT320 - Advanced Algorithms

DEAKIN UNIVERSITY

Dr. Nayyar Zaidi

Object Oriented Design

Design

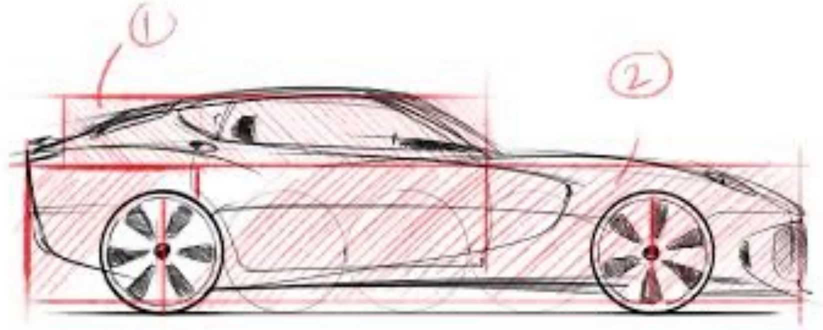
Product Development

Analysis

Design

Production

Lets build a car with
certain characteristics



Engine Specification

Brakes

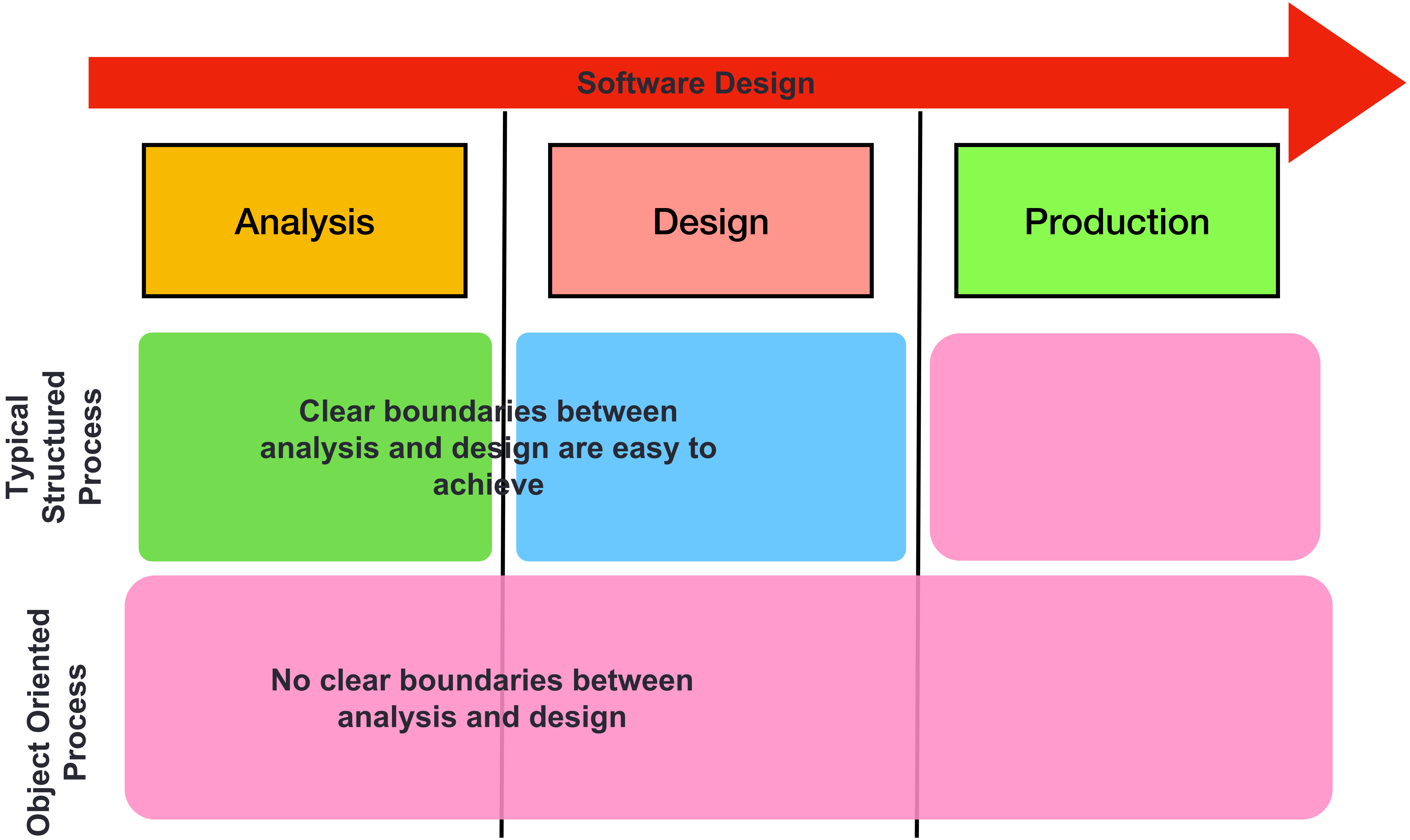
Body Shape

Interior

Electrical

- Unlike earlier techniques such as structured analysis and structured design, there is no sharp boundary between analysis and design
  - Mostly the same tools are used to document both design and analysis, e.g., UML diagrams
  - Class Diagrams (Conceptual, Specification, Implementation)

Design



Software Design

Analysis

Design

Production

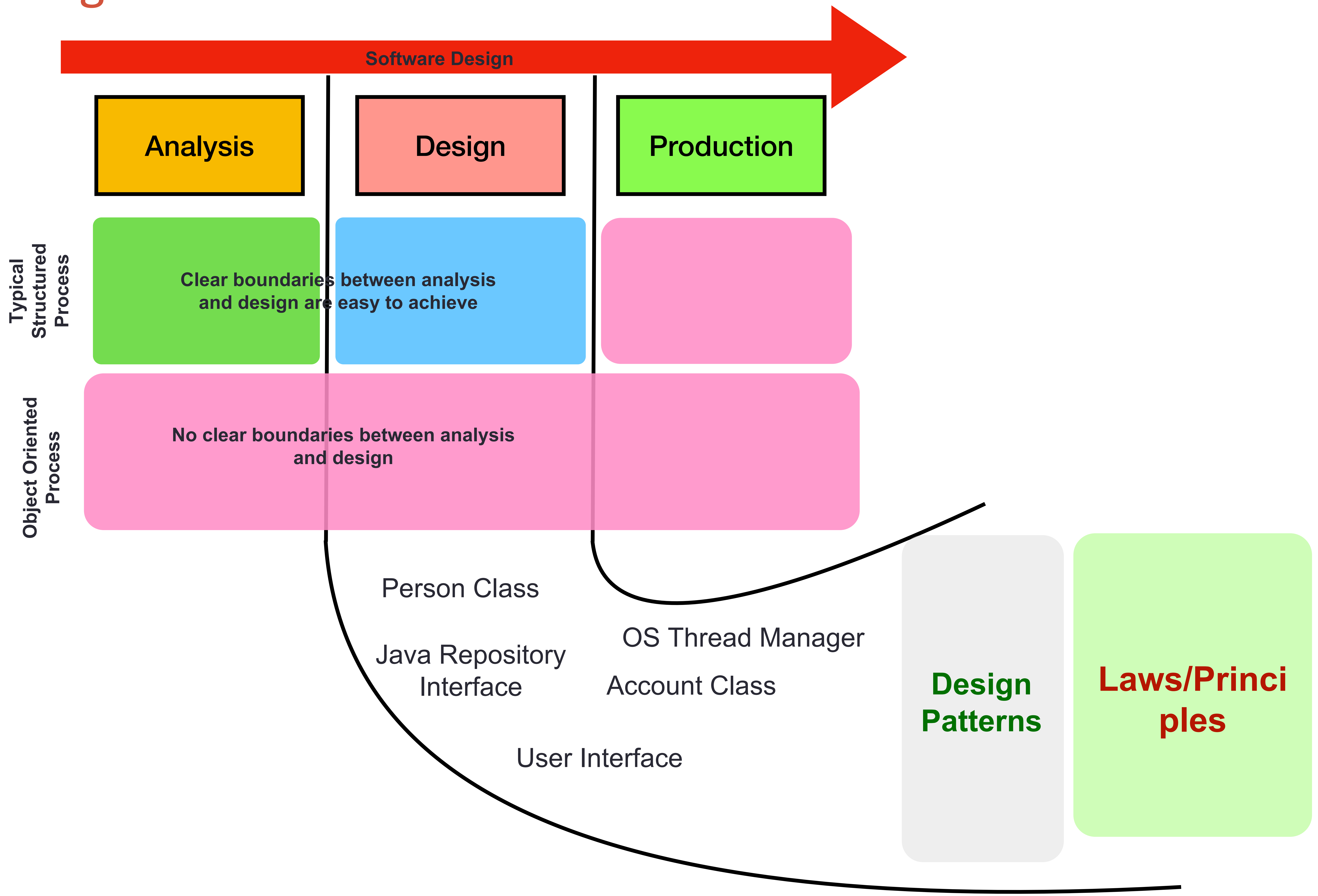
Typical Structured Process

Clear boundaries between analysis and design are easy to achieve

Object Oriented Process

No clear boundaries between analysis and design

Design


Software Design
Analysis
Design
Production
Typical Structured Process
Clear boundaries between analysis and design are easy to achieve
Object Oriented Process
No clear boundaries between analysis and design
Person Class
Java Repository Interface
OS Thread Manager
Account Class
User Interface
Design Patterns
Laws/Princi ples

Software Design

- Software design is the process by which an agent:
  - creates a specification of a software artifact,
  - intended to accomplish goals,
  - using a set of primitive components and subject to constraints
- Software design may refer to either "all the activity involved in conceptualizing, framing, implementing, commissioning, and ultimately modifying complex systems" or "the activity following requirements specification and before programming, as ... [in] a stylized software engineering process."
- Software design usually involves problem solving and planning a software solution
  - This includes both a low-level component and algorithm design and a high-level, architecture design

Key Ideas in OO Design

Reduction of Dependencies

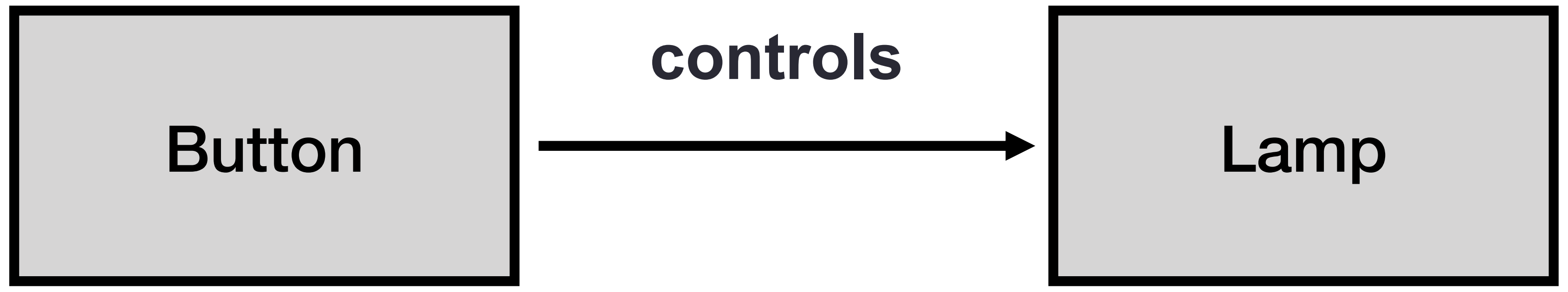
- A dependency exists when a change in some element of software may cause changes in another element of software
  - This is the major cause of maintenance problems
  - We aim to reduce dependencies as much as possible
    - Leads to Dependency Inversion Principle (DIP)

Example

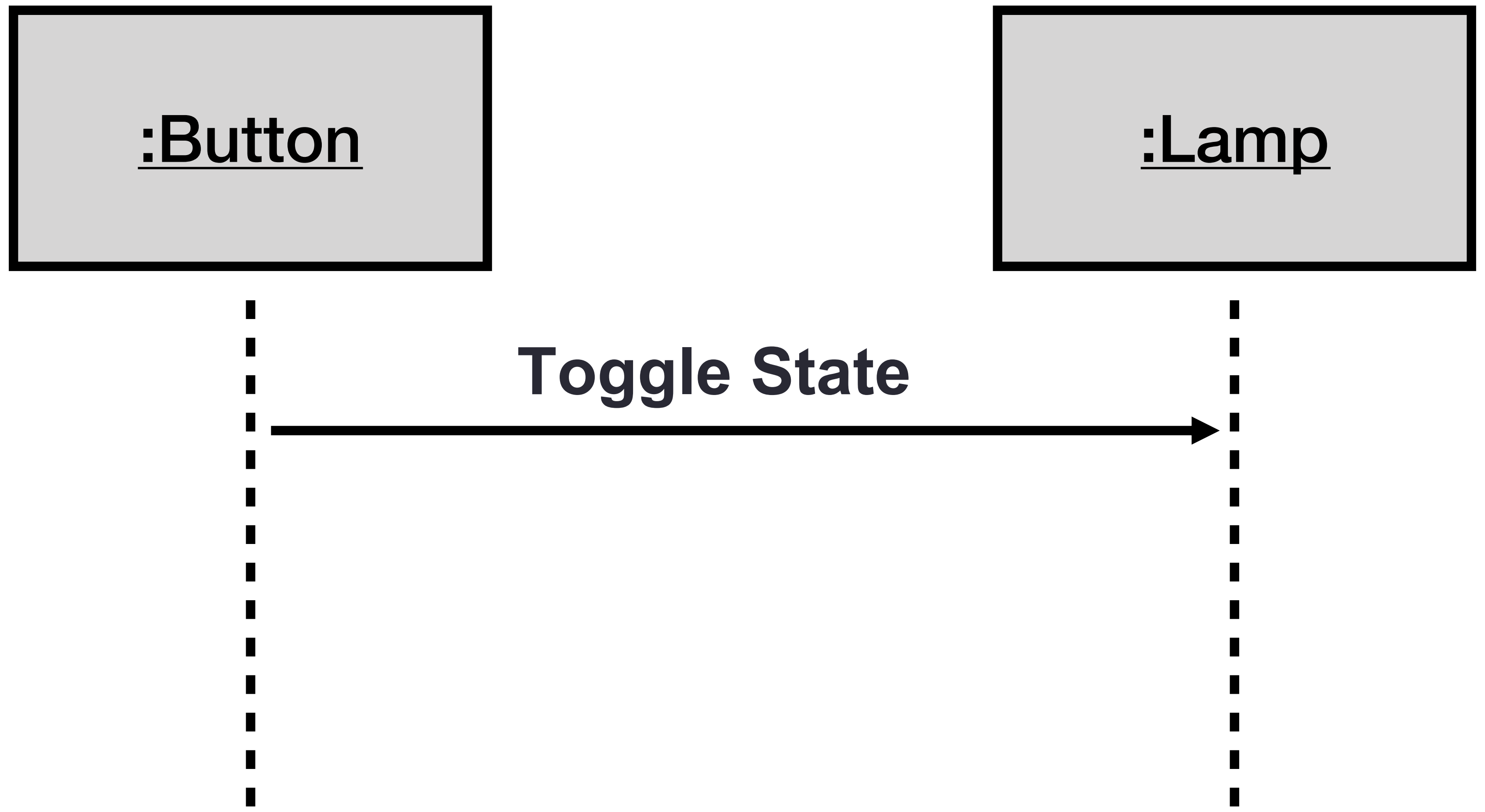
- The Use Case and an Object Model

Use Case	Turn lamp on an off
Actors	Button Pusher
Type	Primary
Description	When the ButtonPusher presses the button, the lamp goes on if it was off, and goes off if it was already on.

Class Diagram



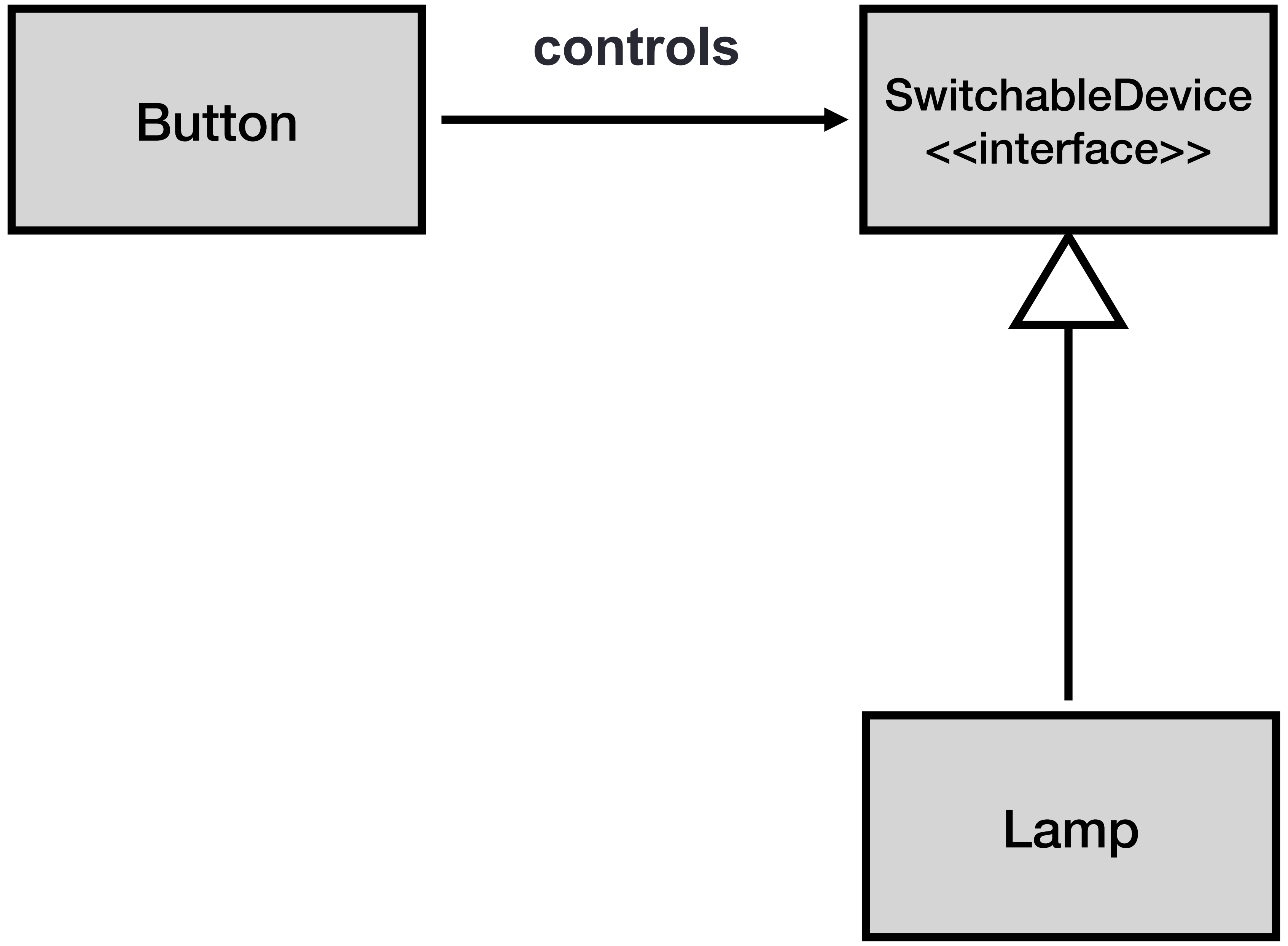
Sequence Diagram



- **Problems**
  - Any change to lamp will require a corresponding change to (or at least a recompilation of) button
  - It is very difficult to reuse either component alone, for example a button to start a motor and not to turn on a lamp
  - The code for button depends directly on the code for lamp

Example

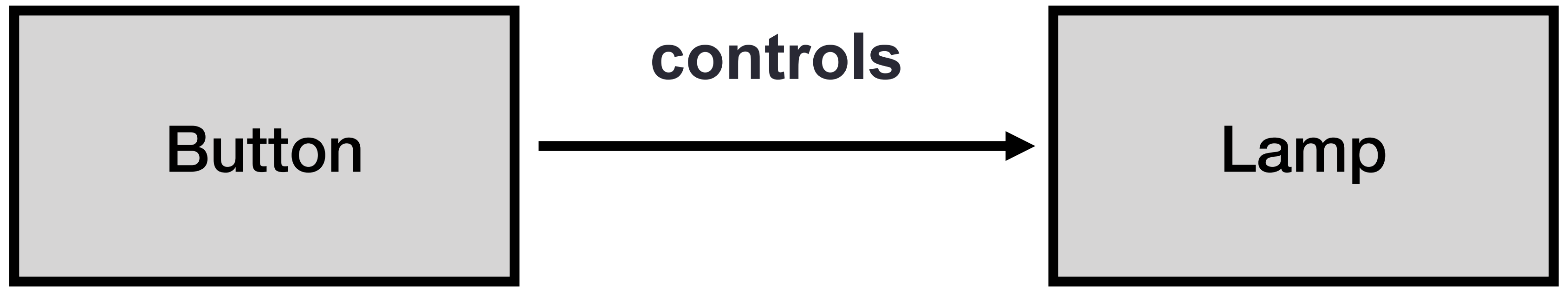
- A better solution
  - Illustrated with Class Diagram



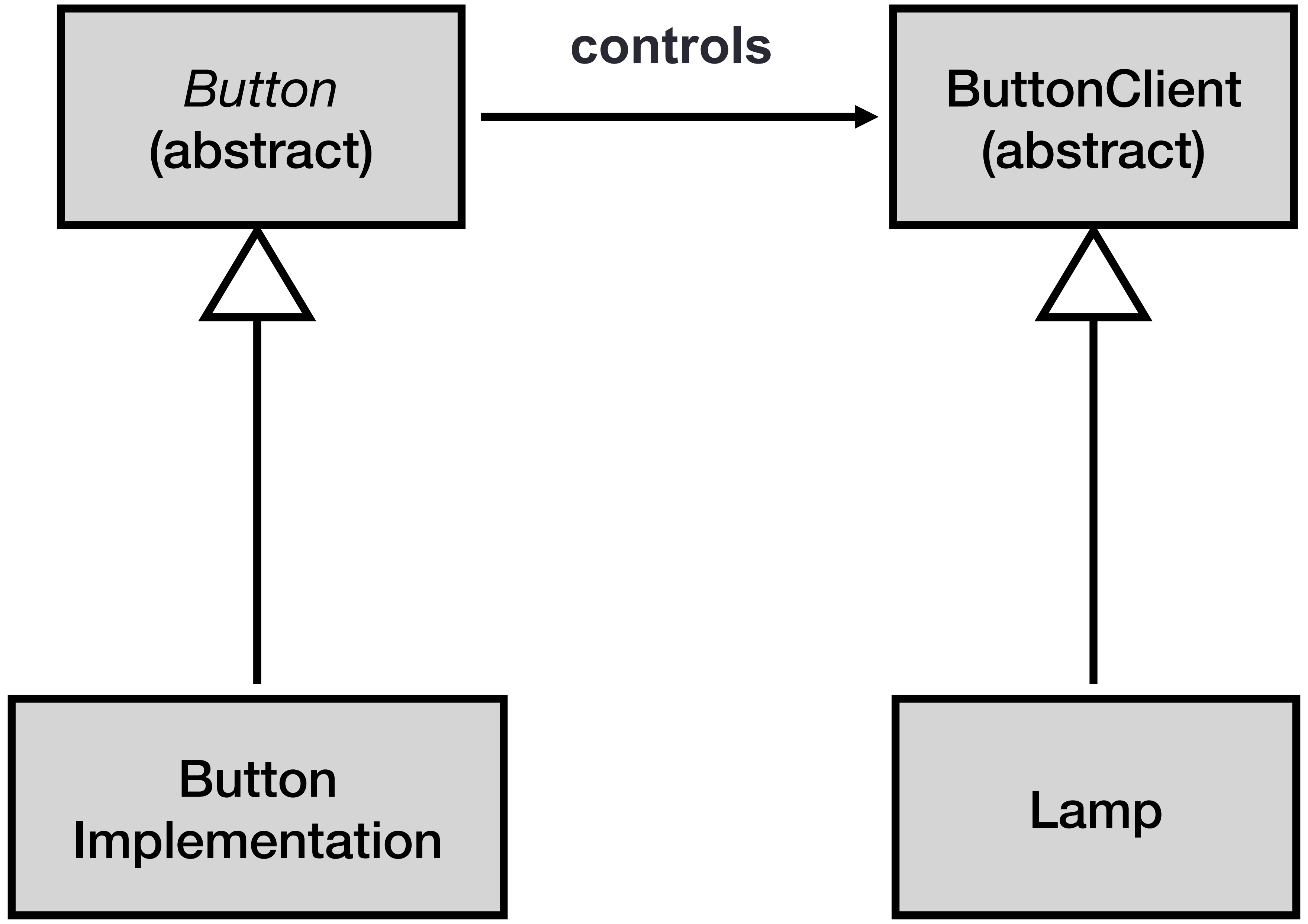
- Underlying abstraction is to relay an on/off gesture from a user to a target object
  - We should remove implementation from the abstraction
  - Button should know nothing about implementation of detecting the user gesture
  - Button should know nothing about Lamp

- Highly resistant to change, and high-level abstractions are easily reused elsewhere
- Use case requirements are still easily traceable in the design

Example



- A better solution
  - Illustrated with Class Diagram



- Underlying abstraction is to relay an on/off gesture from a user to a target object
  - We should removed implementation from the abstraction
  - Button should know nothing about implementation of detecting the user gesture
  - Button should know nothing about Lamp

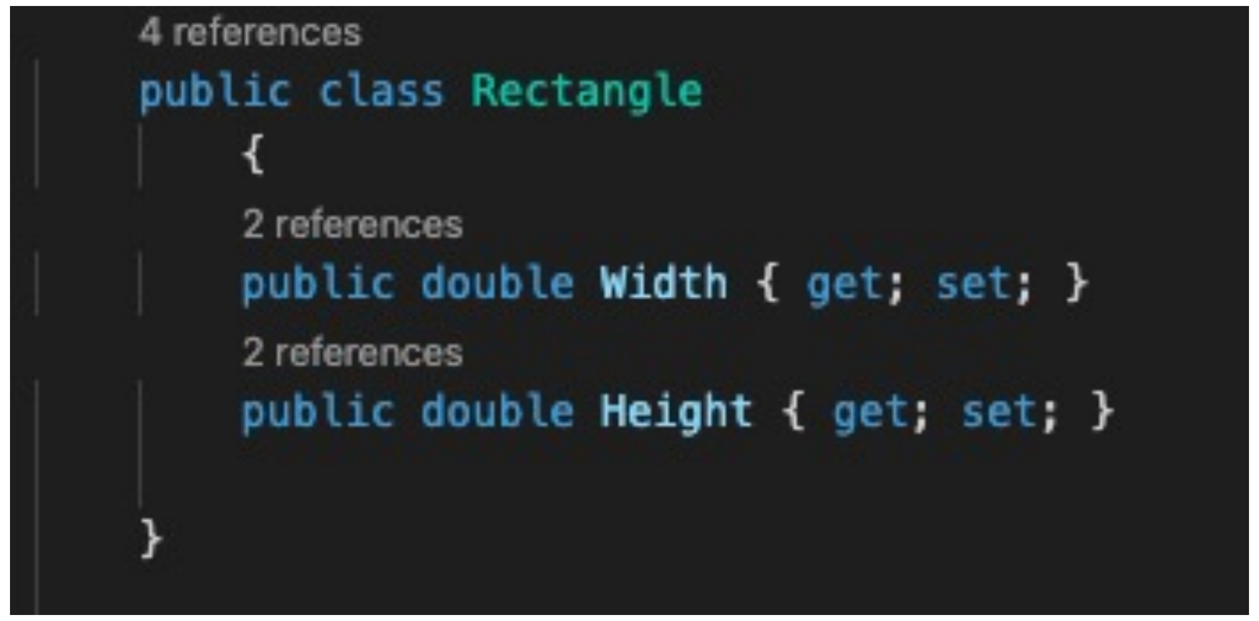
- Highly resistant to change, and high-level abstractions are easily reused elsewhere
- Use case requirements are still easily traceable in the design

Open-Closed Principle (OCP)

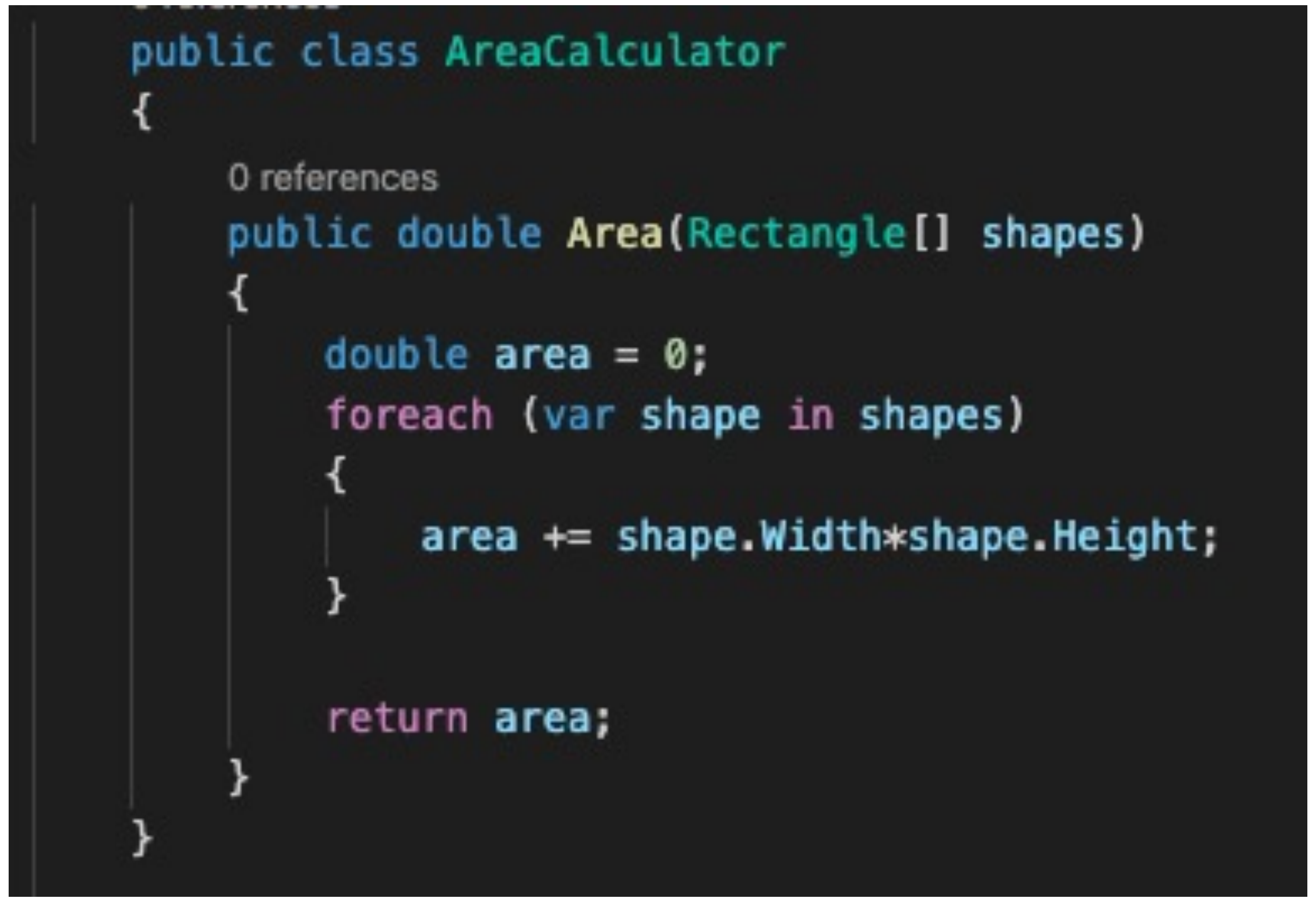
- **Classes should be open for extension, but closed for modification**
  - A class is open for extension if we can add functionality to it: e.g. expand the set of operations or add fields to its data structures
  - A class is closed for modification if it allow its behaviour to be extended without modifying its source-code
  - Write code in a way that does not needs to be changed anytime the requirement of your program changes

- This is normally implemented with inheritance/polymorphism in OO systems

Open-Closed Principle (OCP)

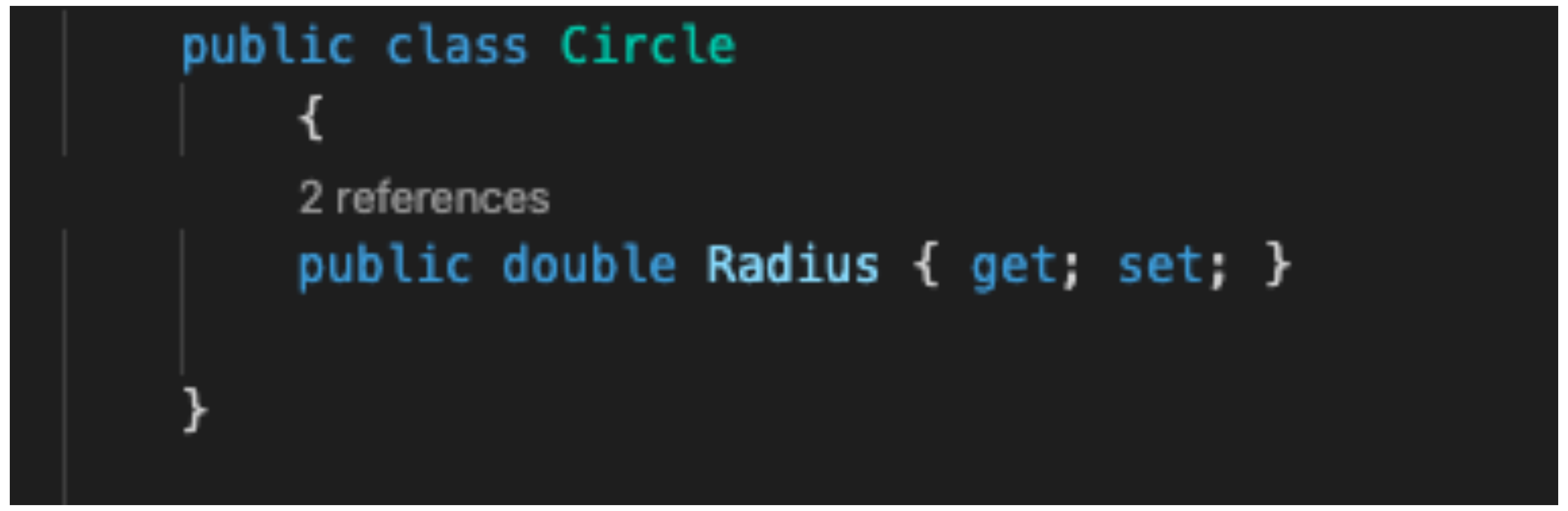


```
4 references
public class Rectangle
    {
        2 references
        public double Width { get; set; }
        2 references
        public double Height { get; set; }

}
```

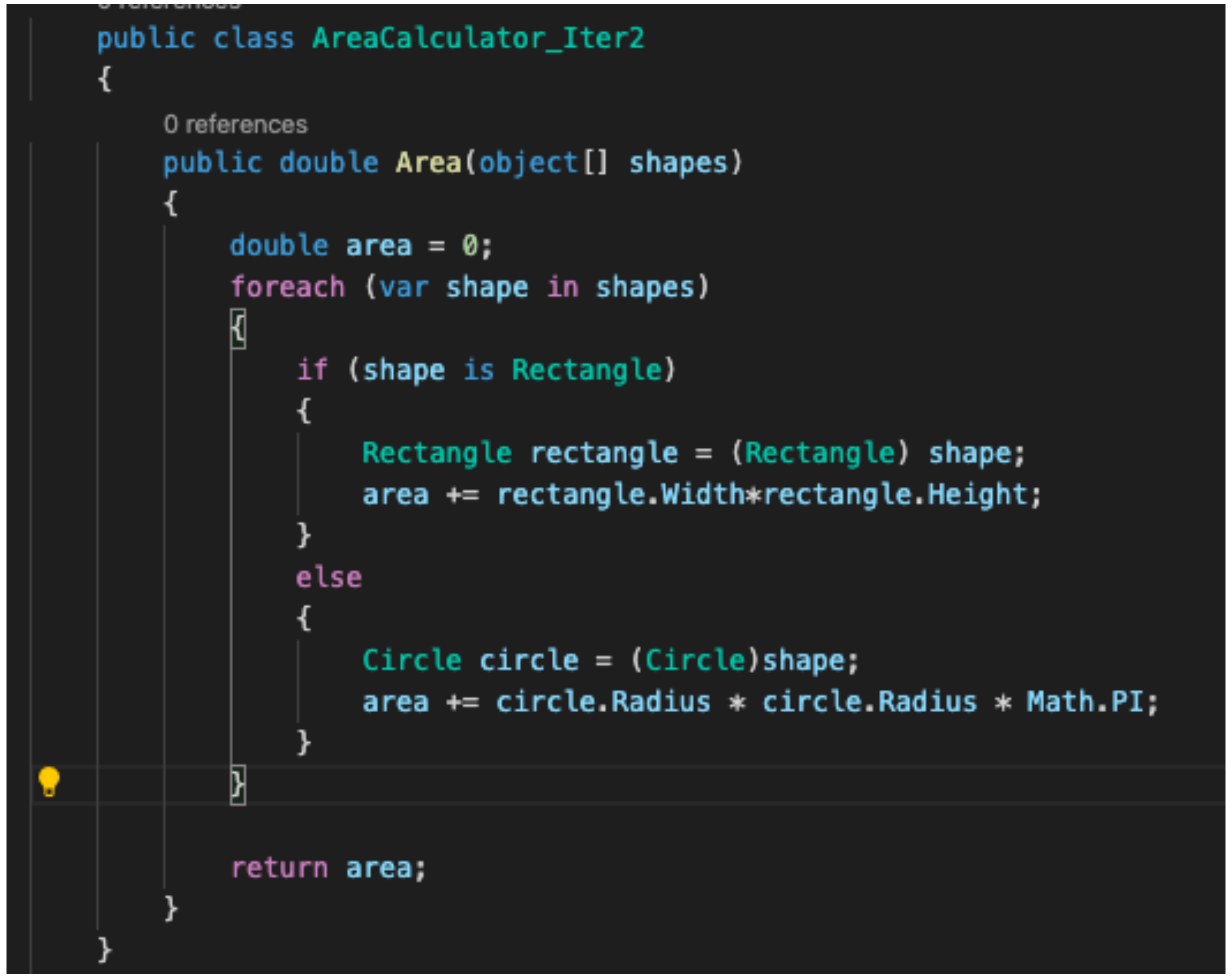


```
public class AreaCalculator
{
    0 references
    public double Area(Rectangle[] shapes)
    {
        double area = 0;
        foreach (var shape in shapes)
        {
            area += shape.Width*shape.Height;
        }

        return area;
    }
}
```

Iteration 1



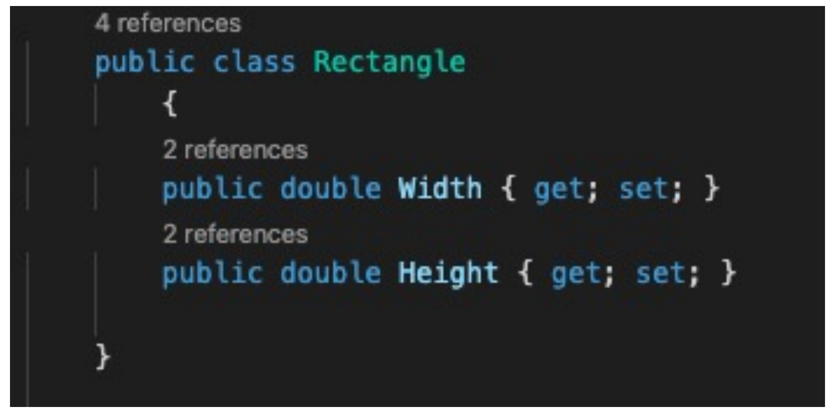
```
public class Circle
    {
        2 references
        public double Radius { get; set; }

    }
```



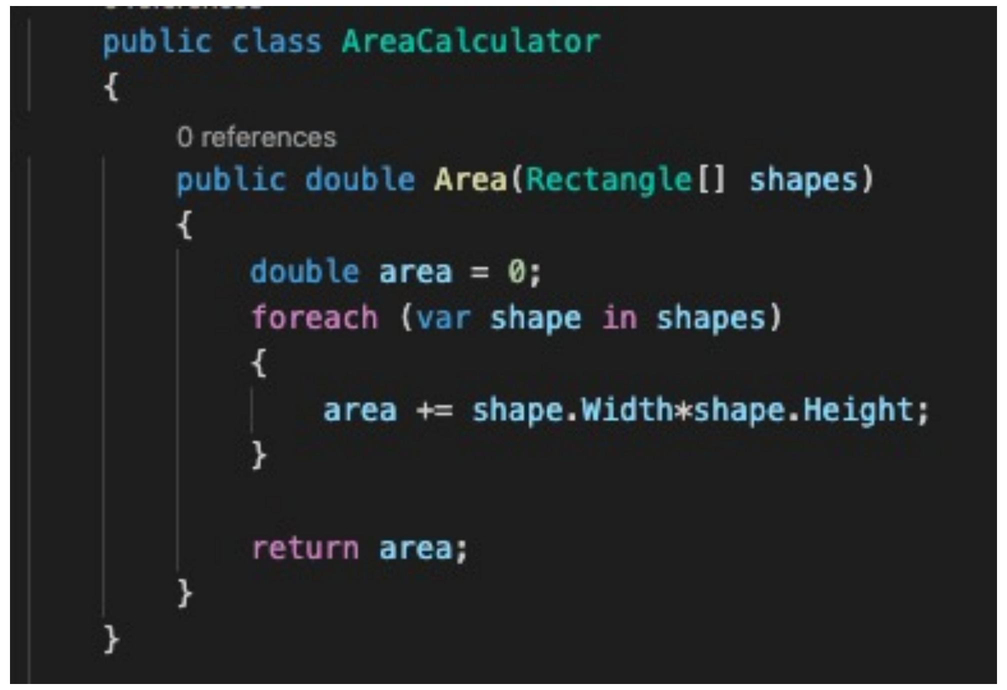
```
public class AreaCalculator_Iter2
{
    0 references
    public double Area(object[] shapes)
    {
        double area = 0;
        foreach (var shape in shapes)
        {
            if (shape is Rectangle)
            {
                Rectangle rectangle = (Rectangle) shape;
                area += rectangle.Width*rectangle.Height;
            }
            else
            {
                Circle circle = (Circle)shape;
                area += circle.Radius * circle.Radius * Math.PI;
            }
        }

        return area;
    }
}
```

Iteration 2

Open-Closed Principle (OCP)

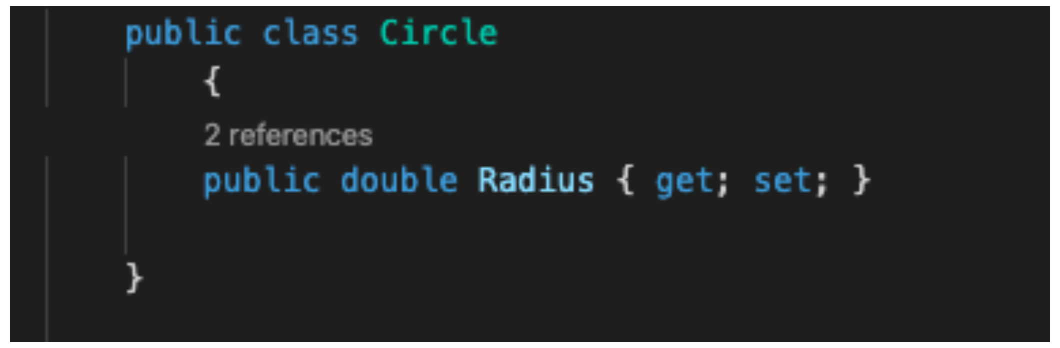


```
4 references
public class Rectangle
    {
        2 references
        public double Width { get; set; }
        2 references
        public double Height { get; set; }

    }
```

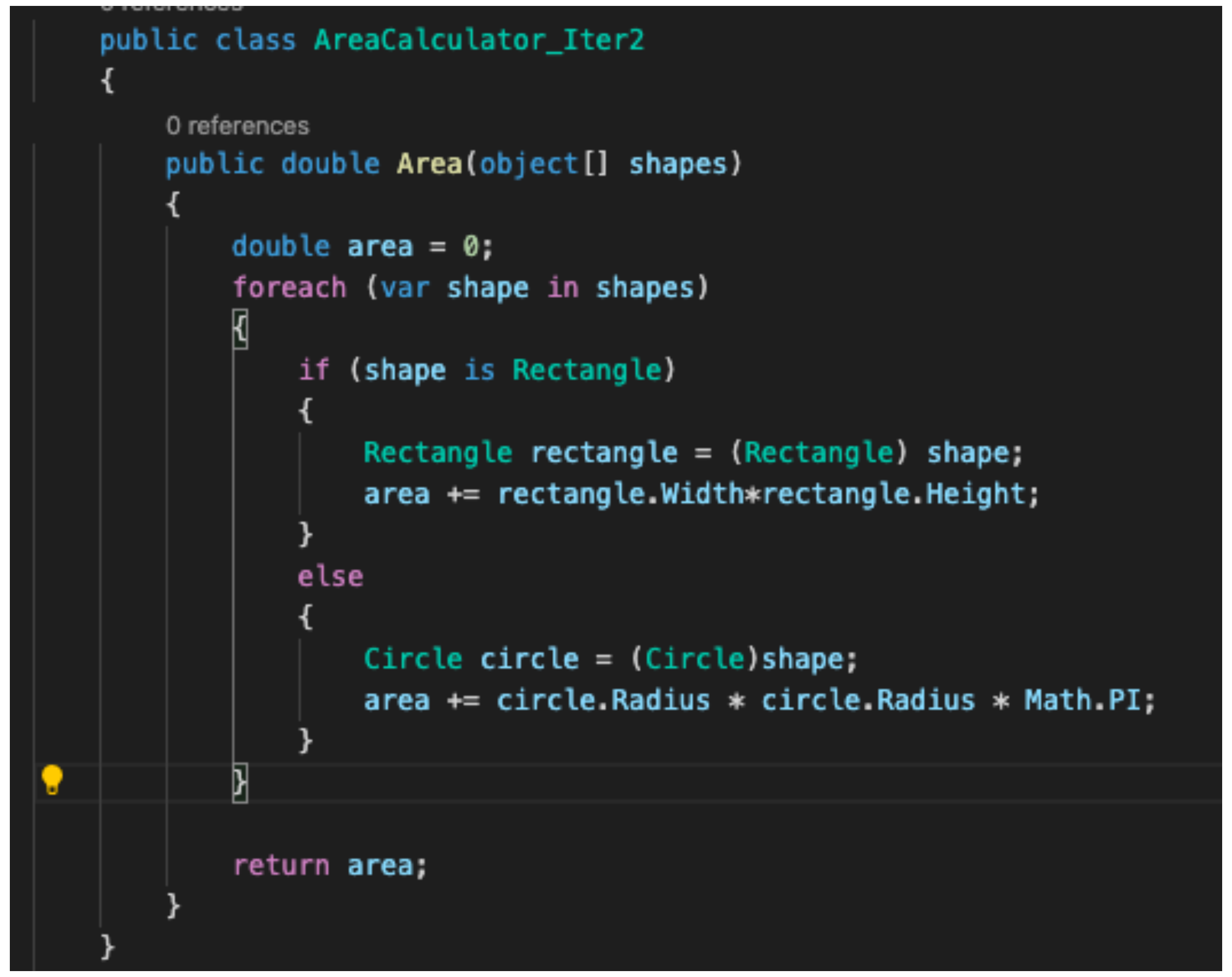


```
public class AreaCalculator
{
    0 references
    public double Area(Rectangle[] shapes)
    {
        double area = 0;
        foreach (var shape in shapes)
        {
            area += shape.Width*shape.Height;
        }

        return area;
    }
}
```

Iteration 1



```
public class Circle
    {
        2 references
        public double Radius { get; set; }

    }
```



```
public class AreaCalculator_Iter2
{
    0 references
    public double Area(object[] shapes)
    {
        double area = 0;
        foreach (var shape in shapes)
        {
            if (shape is Rectangle)
            {
                Rectangle rectangle = (Rectangle) shape;
                area += rectangle.Width*rectangle.Height;
            }
            else
            {
                Circle circle = (Circle)shape;
                area += circle.Radius * circle.Radius * Math.PI;
            }
        }

        return area;
    }
}
```

Iteration 2

Add a Triangle

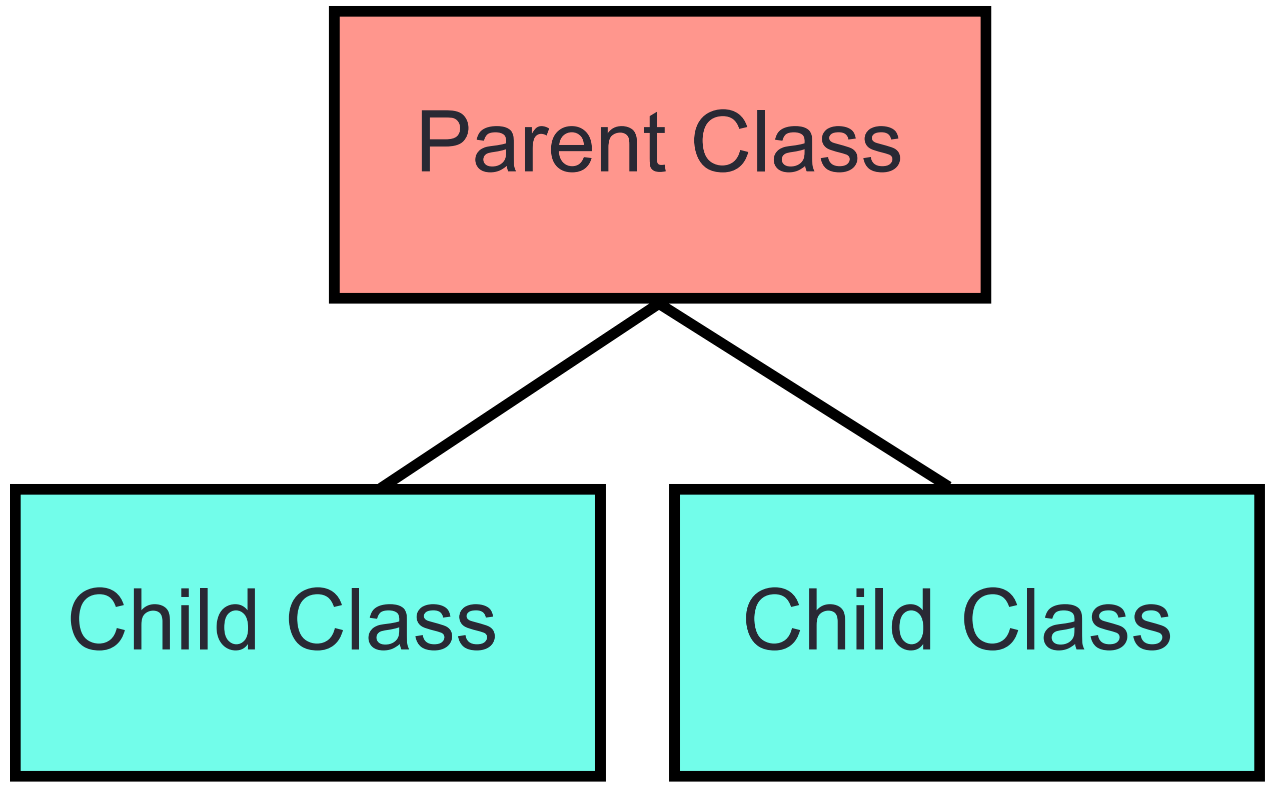
Iteration 3

- AreaCalculator is NOT open for extension and is closed for modifications
  - Solution:
    - Abstract class: Shape — Derived classes: Rectangle, Circle, Triangle each with its own Area functions

Principles of OO Design

Inheritance: The Idea

- Inheritance allows to keep:
  - common state and behaviour in one place (called the parent or the superclass) and
  - different state and behaviour in the individual classes that inherit from the parent (called child or subclass)



Inheritance

- Two important phrases to identify inheritance:
  - A Lecturer is a Person
  - A Lecturer is a kind of Person

Polymorphism Test

- SaleProduct can be replaced for Product any where we want, or
  - Lecturer can be replaced for Person

Implementation Inheritance

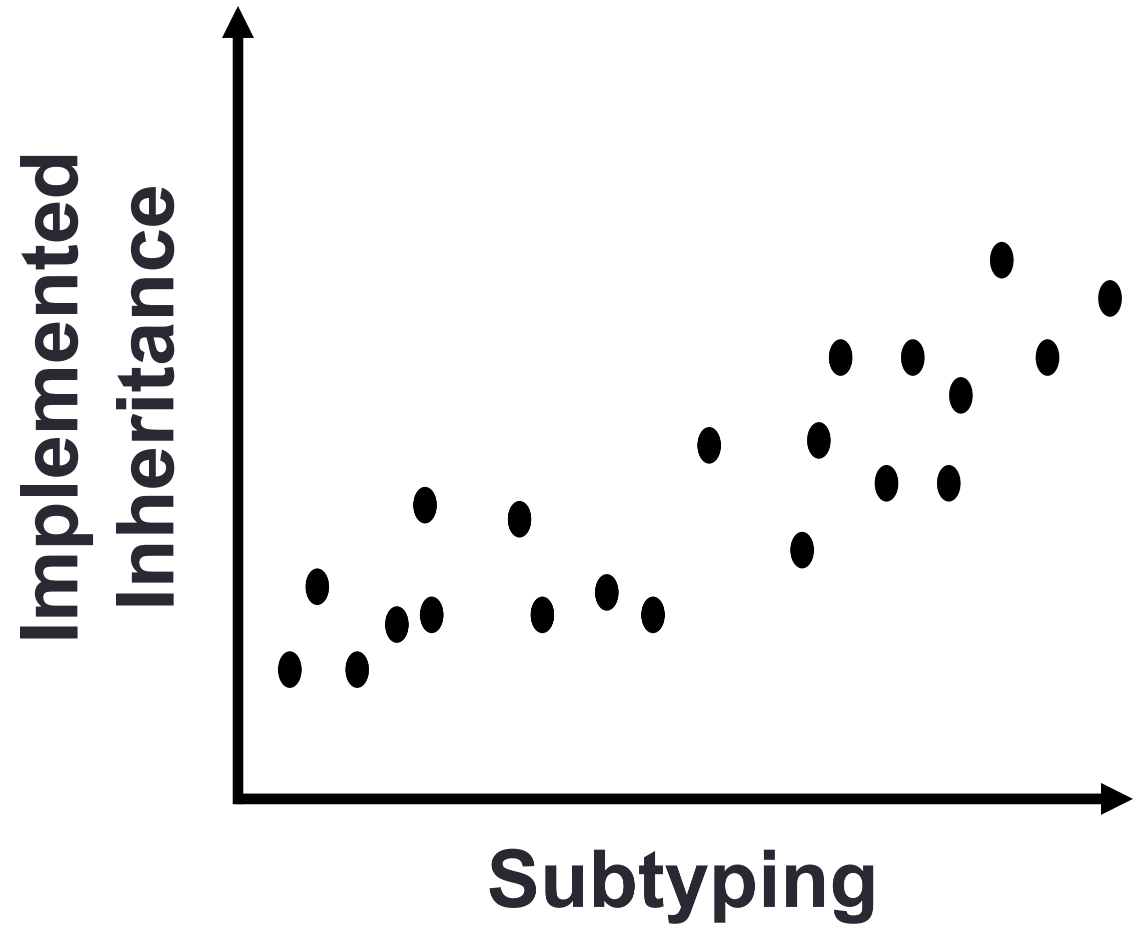
- Factor out repeated code
  - To create a new class, write on the difference
  - Code re-use
  - Adding more functions

```
class Product {
     private String title;
     private String description;
     private int price; // in cents
     public int getPrice() {
          return price;
      }
     public int getTax() {
          return (int)(getPrice() * 0.096);
     }
     …
}

class SaleProduct {
     private String title;
     private String description;
     private int price; // in cents
     private float factor;
     public int getPrice() {
        return (int)(price*factor);
     }
     public int getTax() {
          return (int)(getPrice() * 0.096);
     }
     …
}

class SaleProduct extends Product {
     private float factor;
     public int getPrice() {
       return (int)(super.getPrice()*factor);
      }
}
```

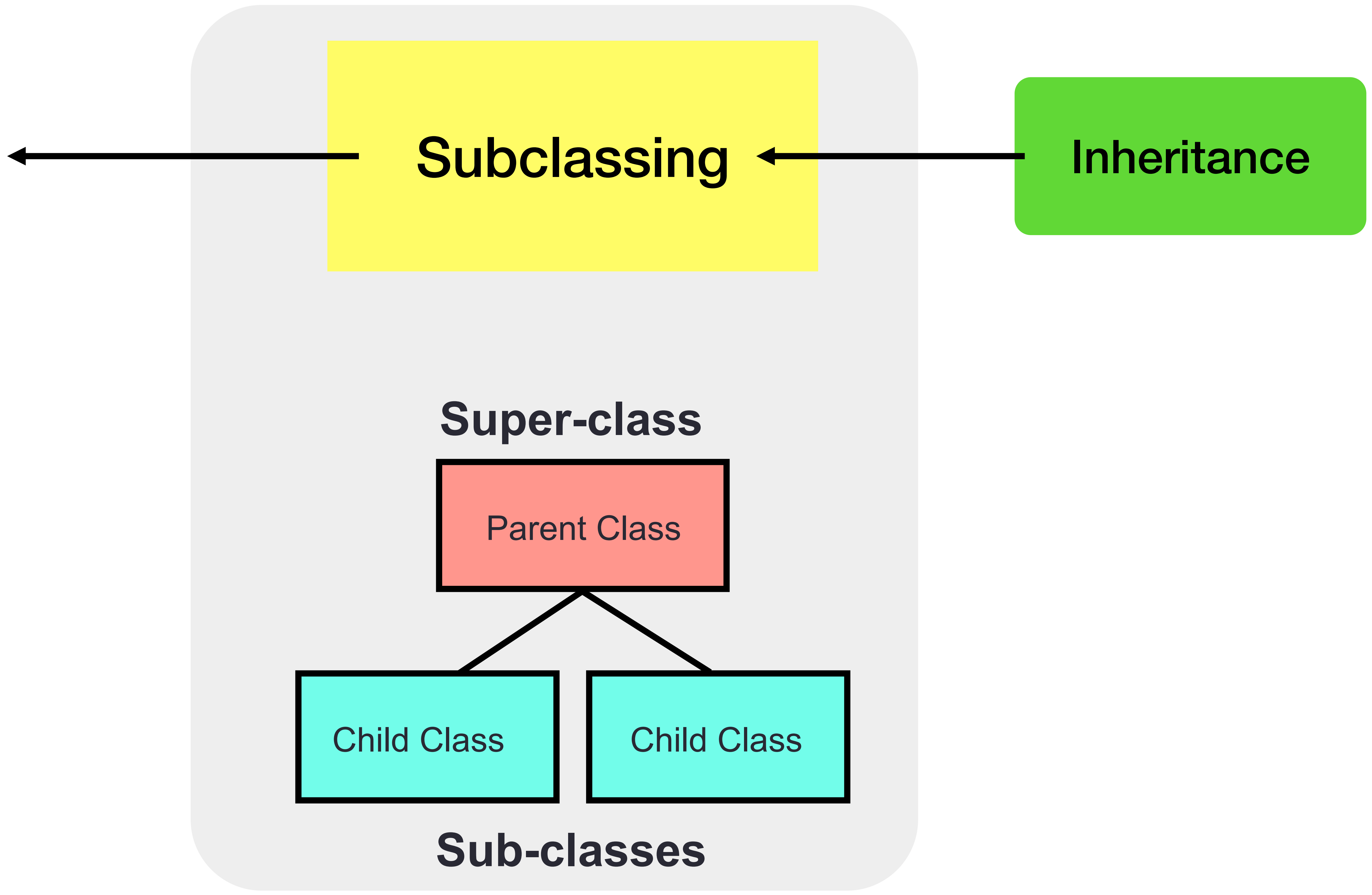
Inheritance (Sub-class)

- **What is a sub-class?**
- A class derived using inheritance
  - **By the use of sub-classes we implement inheritance**

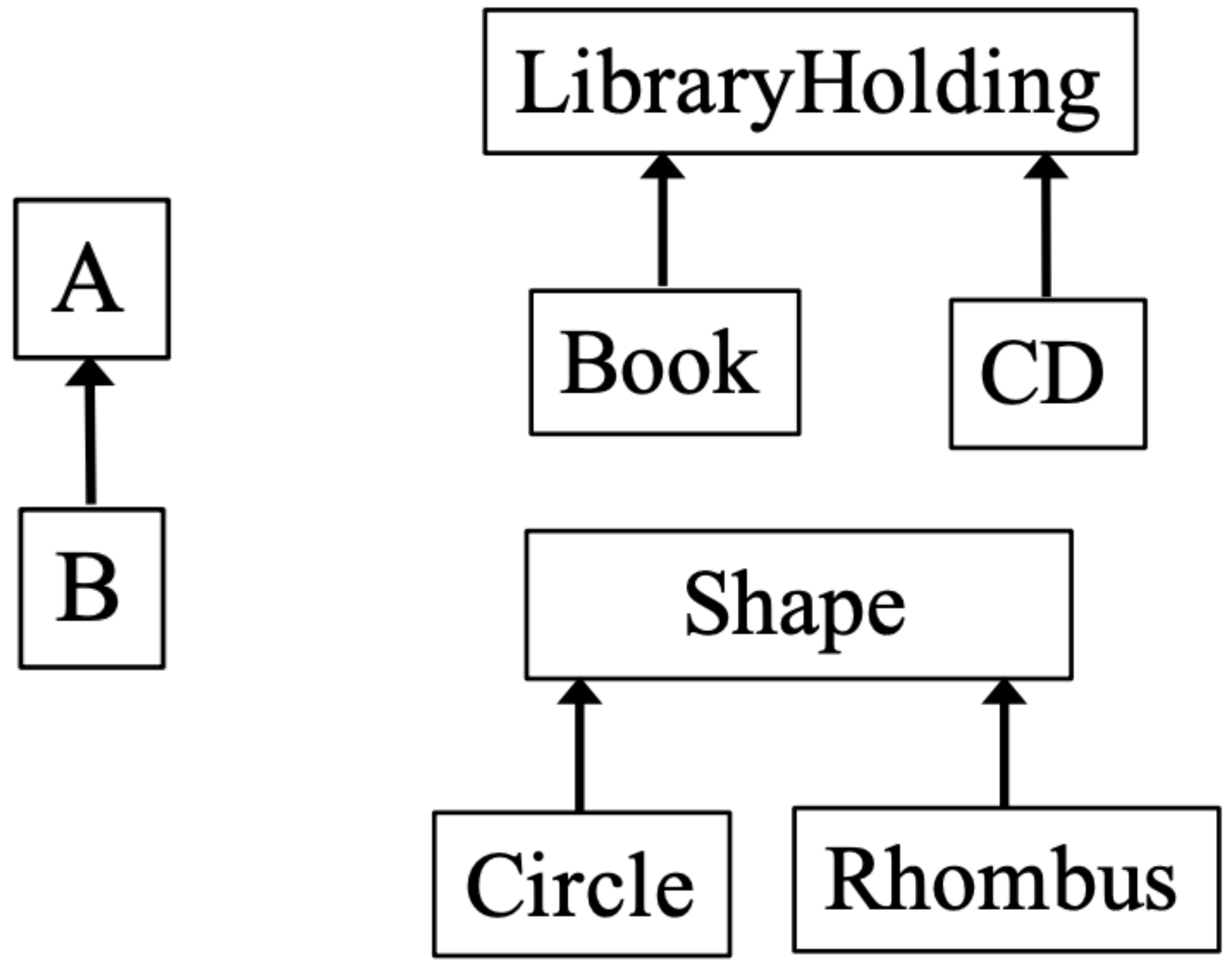
- **Code Inheritance**
- **Syntactic Relationship (is-a)**



- **Behavioural Sub-typing**
- **Polymorphism Test**
- **Semantic relationship**
- **Interface Inheritance**



Sub-types



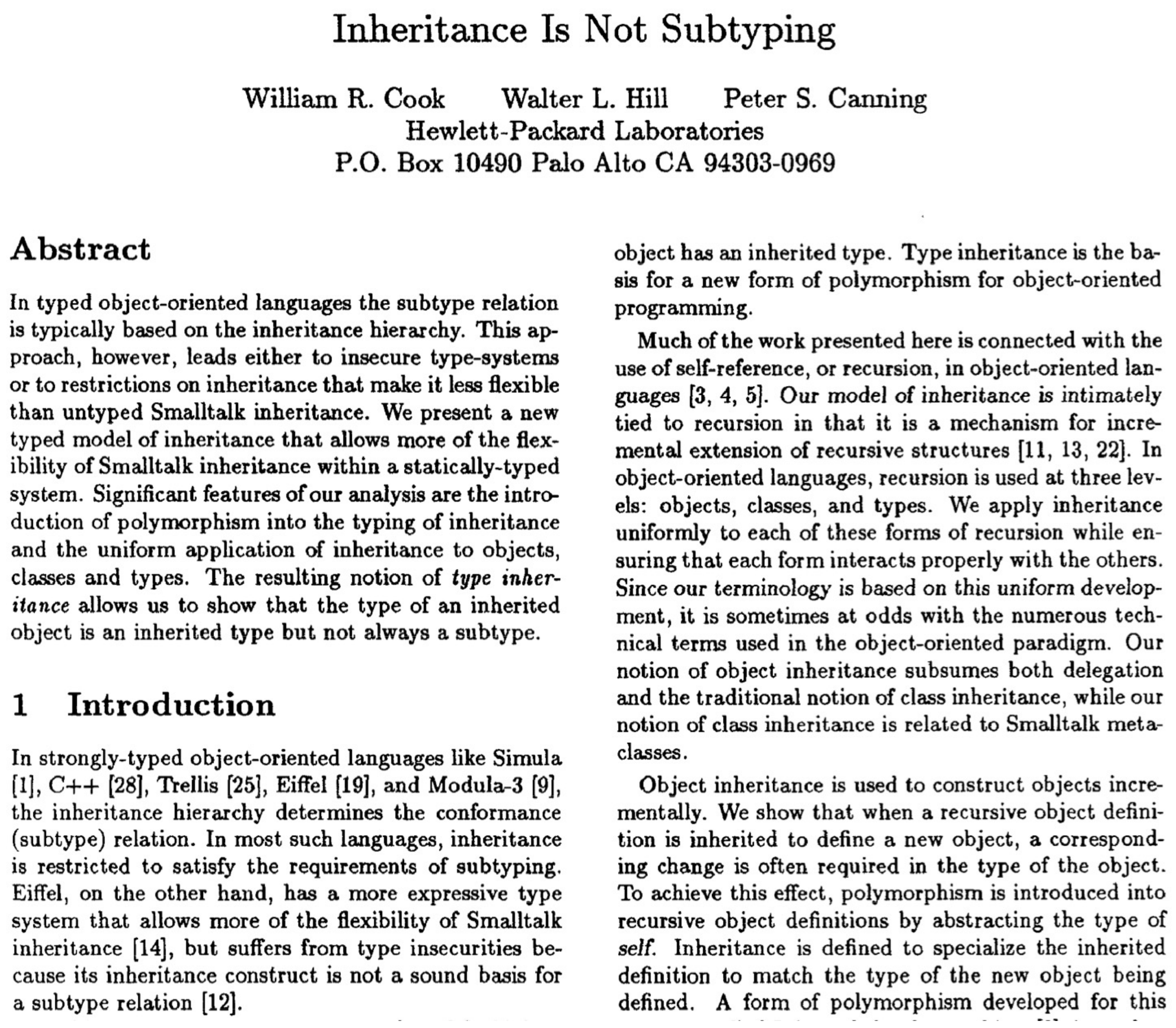
- Sometimes “every B is an A”
  - Example: In a library database:
    - Every book is a library holding
    - Every CD is a library holding
- Subtyping expresses this
  - “B is a subtype of A” means:
    - “every object that satisfies the rules for a B also satisfies the rules for an A”
- Goal: code written using A's specification operates correctly even if given a B
  - Plus: clarify design, share tests, (sometimes) share code

Sub-types

- Subtypes are **substitutable** for supertypes
  - Instances of subtype won't surprise client by failing to satisfy the supertype's specification
  - Instances of subtype won't surprise client by having more expectations than the supertype's specification

LISKOV SUBSTITUTION PRINCIPLE
If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

Root of Confusion



Inheritance Is Not Subtyping

William R. Cook Walter L. Hill Peter S. Canning
Hewlett-Packard Laboratories
P.O. Box 10490 Palo Alto CA 94303-0969

Abstract

In typed object-oriented languages the subtype relation is typically based on the inheritance hierarchy. This approach, however, leads either to insecure type-systems or to restrictions on inheritance that make it less flexible than untyped Smalltalk inheritance. We present a new typed model of inheritance that allows more of the flexibility of Smalltalk inheritance within a statically-typed system. Significant features of our analysis are the introduction of polymorphism into the typing of inheritance and the uniform application of inheritance to objects, classes and types. The resulting notion of *type inheritance* allows us to show that the type of an inherited object is an inherited type but not always a subtype.

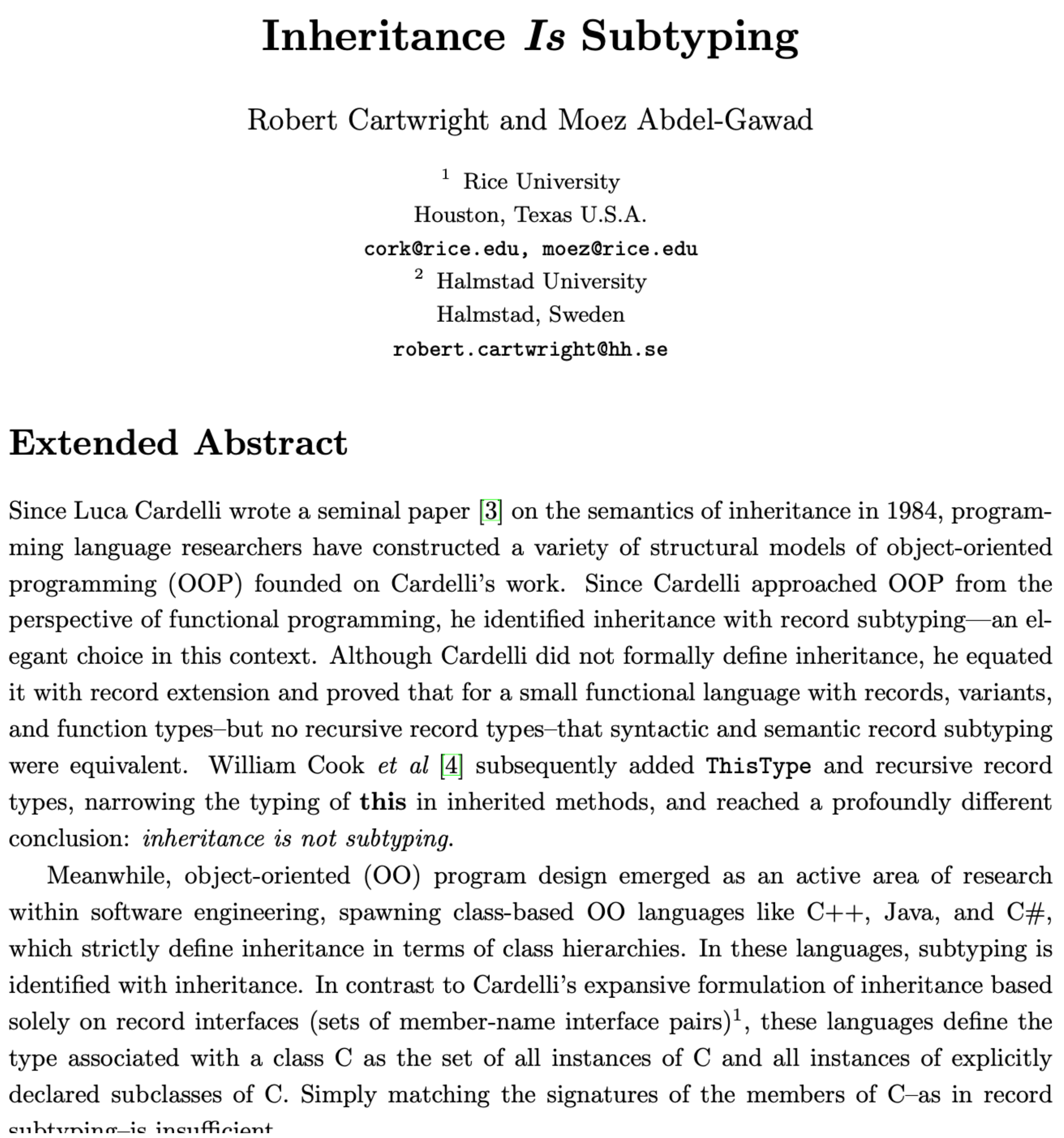
1 Introduction

In strongly-typed object-oriented languages like Simula [1], C++ [28], Trellis [25], Eiffel [19], and Modula-3 [9], the inheritance hierarchy determines the conformance (subtype) relation. In most such languages, inheritance is restricted to satisfy the requirements of subtyping. Eiffel, on the other hand, has a more expressive type system that allows more of the flexibility of Smalltalk inheritance [14], but suffers from type insecurities because its inheritance construct is not a sound basis for a subtype relation [12].

object has an inherited type. Type inheritance is the basis for a new form of polymorphism for object-oriented programming.

Much of the work presented here is connected with the use of self-reference, or recursion, in object-oriented languages [3, 4, 5]. Our model of inheritance is intimately tied to recursion in that it is a mechanism for incremental extension of recursive structures [11, 13, 22]. In object-oriented languages, recursion is used at three levels: objects, classes, and types. We apply inheritance uniformly to each of these forms of recursion while ensuring that each form interacts properly with the others. Since our terminology is based on this uniform development, it is sometimes at odds with the numerous technical terms used in the object-oriented paradigm. Our notion of object inheritance subsumes both delegation and the traditional notion of class inheritance, while our notion of class inheritance is related to Smalltalk metaclasses.

Object inheritance is used to construct objects incrementally. We show that when a recursive object definition is inherited to define a new object, a corresponding change is often required in the type of the object. To achieve this effect, polymorphism is introduced into recursive object definitions by abstracting the type of *self*. Inheritance is defined to specialize the inherited definition to match the type of the new object being defined. A form of polymorphism developed for this



Inheritance *Is* Subtyping

Robert Cartwright and Moez Abdel-Gawad

[1] Rice University
Houston, Texas U.S.A.
cork@rice.edu, moez@rice.edu
[2] Halmstad University
Halmstad, Sweden
robert.cartwright@hh.se

Extended Abstract

Since Luca Cardelli wrote a seminal paper [3] on the semantics of inheritance in 1984, programming language researchers have constructed a variety of structural models of object-oriented programming (OOP) founded on Cardelli's work. Since Cardelli approached OOP from the perspective of functional programming, he identified inheritance with record subtyping—an elegant choice in this context. Although Cardelli did not formally define inheritance, he equated it with record extension and proved that for a small functional language with records, variants, and function types–but no recursive record types–that syntactic and semantic record subtyping were equivalent. William Cook *et al* [4] subsequently added `ThisType` and recursive record types, narrowing the typing of **this** in inherited methods, and reached a profoundly different conclusion: *inheritance is not subtyping.*

Meanwhile, object-oriented (OO) program design emerged as an active area of research within software engineering, spawning class-based OO languages like C++, Java, and C#, which strictly define inheritance in terms of class hierarchies. In these languages, subtyping is identified with inheritance. In contrast to Cardelli's expansive formulation of inheritance based solely on record interfaces (sets of member-name interface pairs)[1], these languages define the type associated with a class C as the set of all instances of C and all instances of explicitly declared subclasses of C. Simply matching the signatures of the members of C–as in record subtyping–is insufficient

Substitution vs. (Implemented) Inheritance

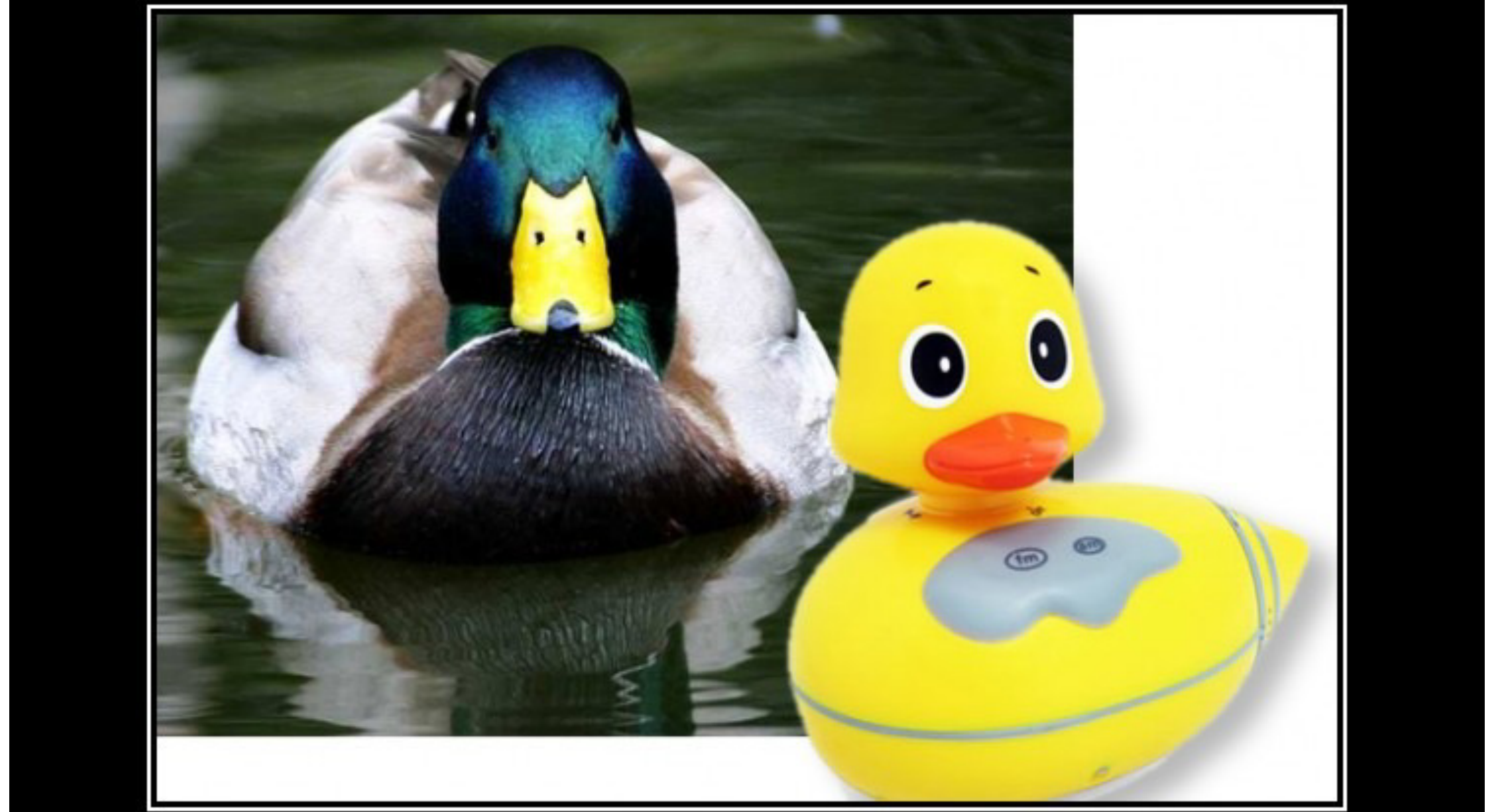
- **Substitution (subtype)** — a specification notion
  - B is a subtype of A iff an object of B can masquerade as an object of A in any context
- **Inheritance (subclass)** — an implementation notion
  - Factor out repeated code
  - To create a new class, write only the differences

😟

- In modern OO PLs — each subclass is a subtype automatically
  - But not necessarily a true sub-type (remember behavioural sub-typing)
  - Reason for a lot of misunderstanding around inheritance
  - E.g., inheritance misuse etc. (IMO, sub-classing can be misused not inheritance)

Liskov Substitution Principle

- **For class S to be a true subtype of class T, then S must conform to T:**
  - A class S conforms to class T, if an object of class S can be provided in any context where an object of class T is expected and correctness is still preserved when any accessor method is executed



LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

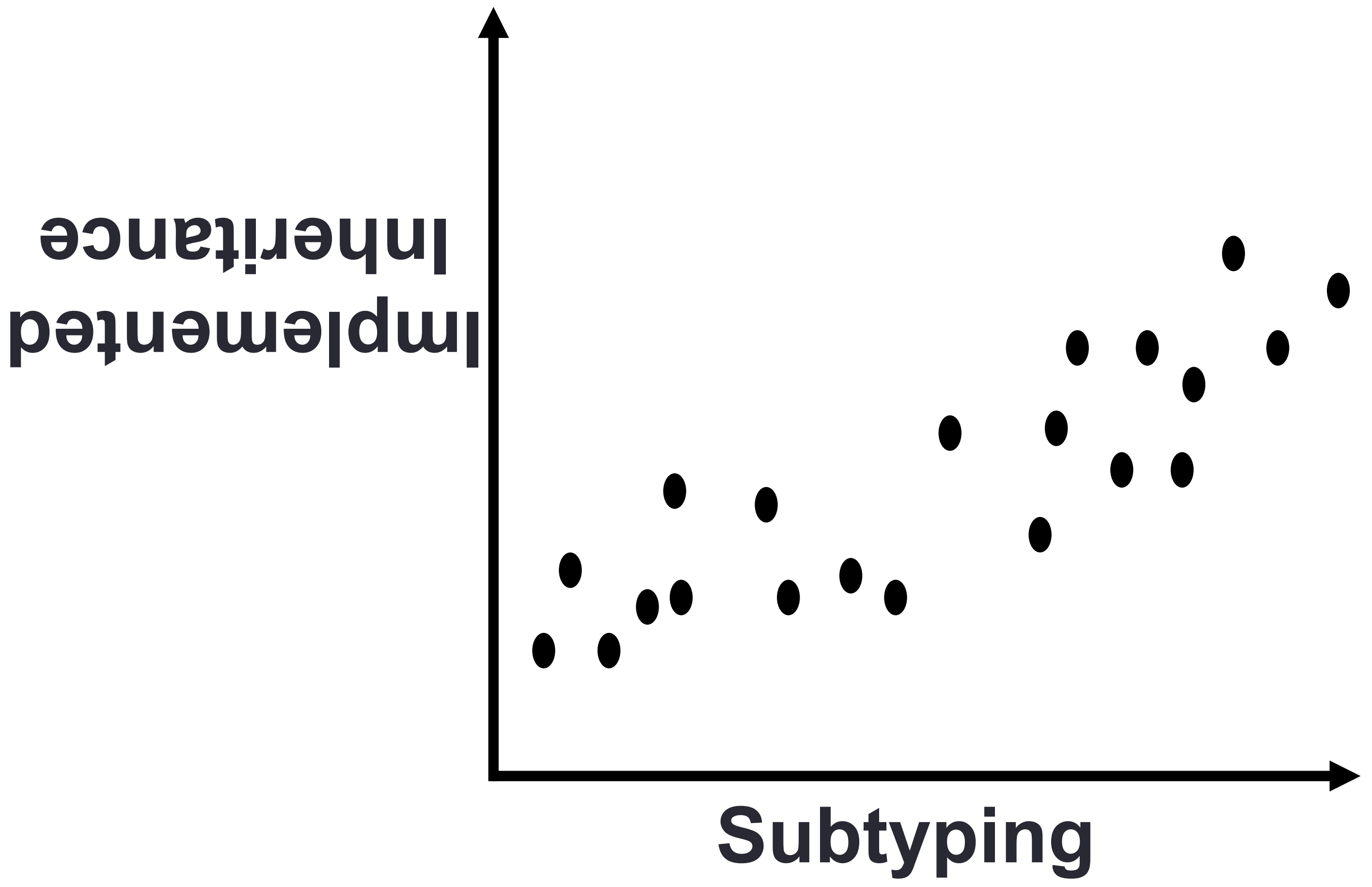
Solution

Subtyping without Inheritance

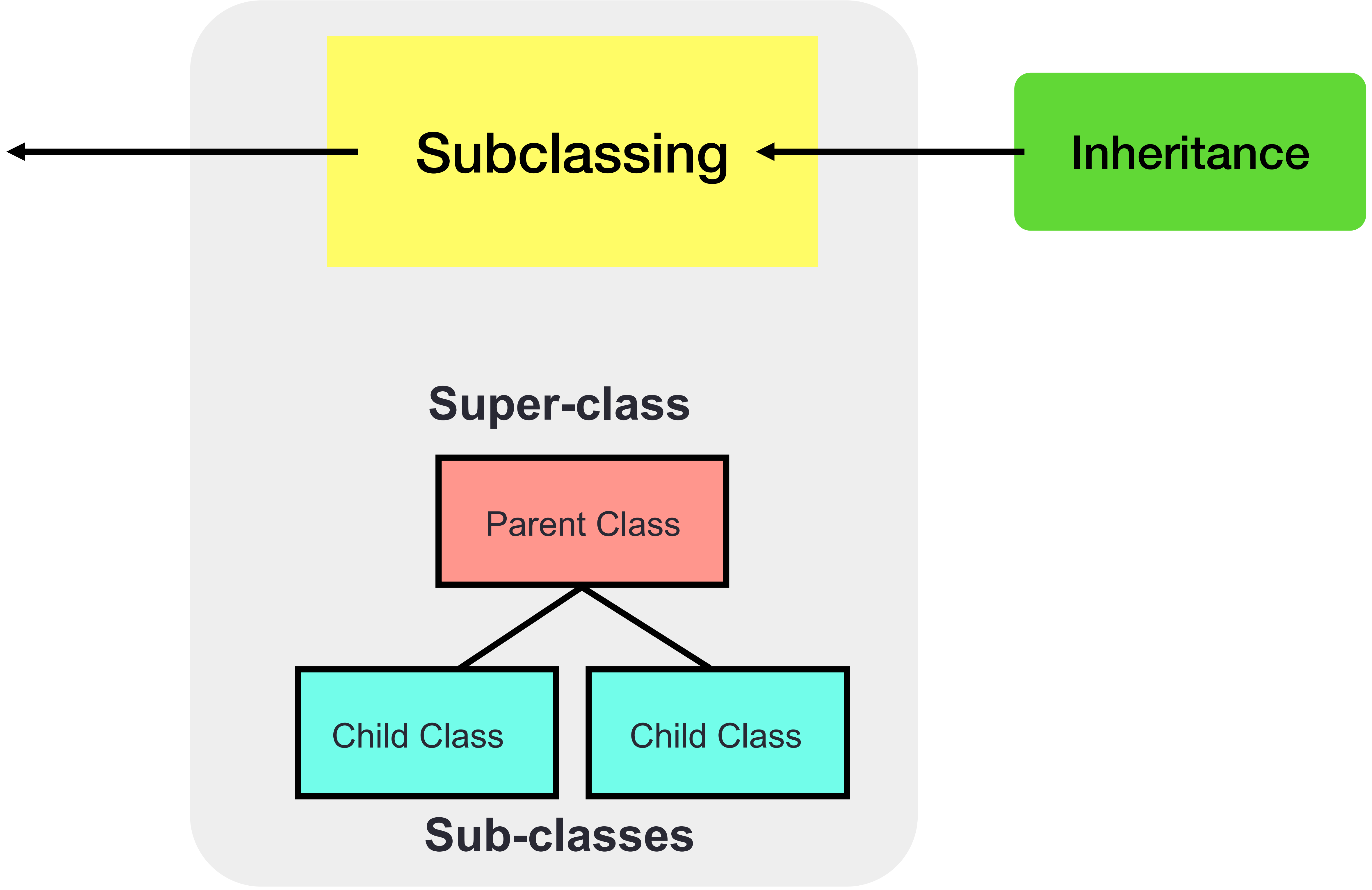
Interfaces

Inheritance without Subtyping

Composition



- **Behavioural Sub-typing**
- **Polymorphism Test**


Subclassing

Inheritance

Super-class

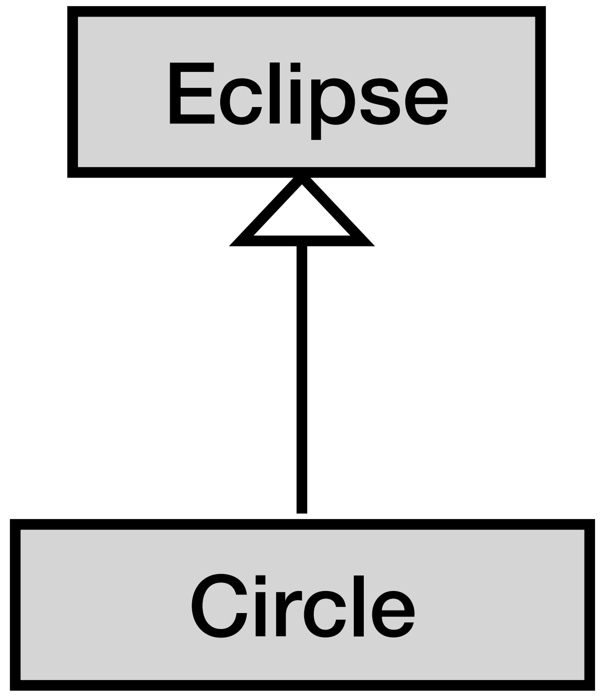
Parent Class

Child Class

Child Class

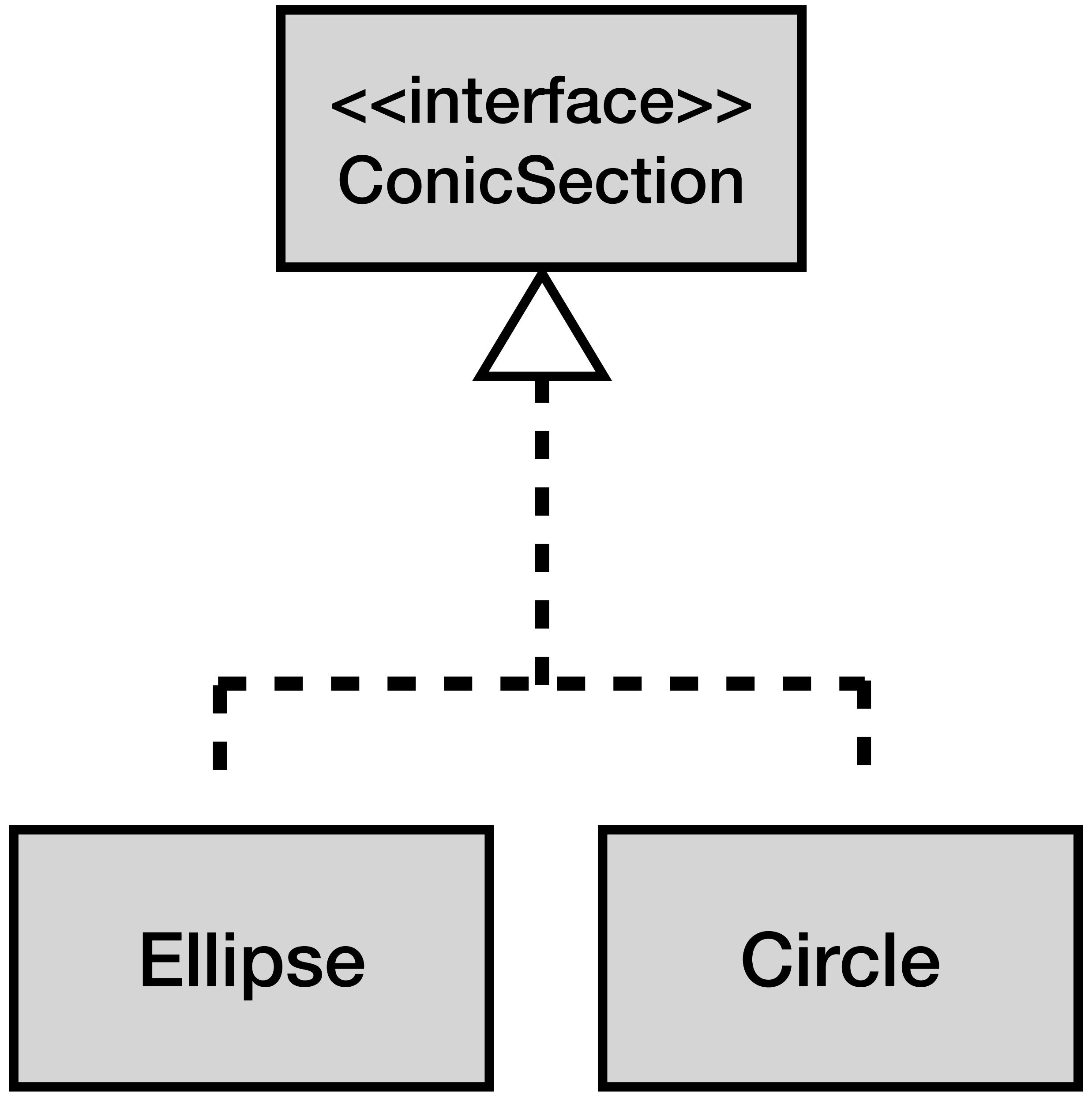
Sub-classes

Subtyping without (implemented) Inheritance



But what happens when a circle gets a stretch_along_x_axis message?

Better



(implemented) Inheritance without Subtyping

```
class Fruit {
    // Return int number of pieces of peel that
    // resulted from the peeling activity.

    public int Peel() {
        Console.WriteLine("Peeling is appealing.");
        return 1;
    }
}

class Apple : Fruit { }

class Example1 {

    public static void Main() {
        Apple apple = new Apple();
        int pieces = apple.Peel();
   }
}
```

```
class Fruit {
    // Return int number of pieces of peel that
    // resulted from the peeling activity.

    public int Peel() {
            Console.WriteLine("Peeling is appealing.");
            return 1;
    }
}

class Apple {
    private Fruit fruit = new Fruit();
    public int Peel() { return fruit.Peel(); }
}

class Example1 {

    public static void Main() {
           Apple apple = new Apple();
            int pieces = apple.Peel();
   }
}
```

On the use of Inheritance

- There are many ways to work out whether you should be using inheritance or not:
  - Liskov’s substitution principle
  - Or, if you nullify or override a lot of behaviour from the parent class, why are you inheriting in the first place

Containment

- Simpler structure
- No ‘surprise’ encapsulation breaking.
- Can be dynamic run-time mechanism. (This is good and bad.)
- You define exactly what this class does.

Inheritance

- Complicated hierarchies
- New method in the parent can add (unwanted) behavior in the child.
- Static (compile-time) mechanism.
- The child is influenced by the parent (unless you totally ignore the parent – if so, why inherit?)

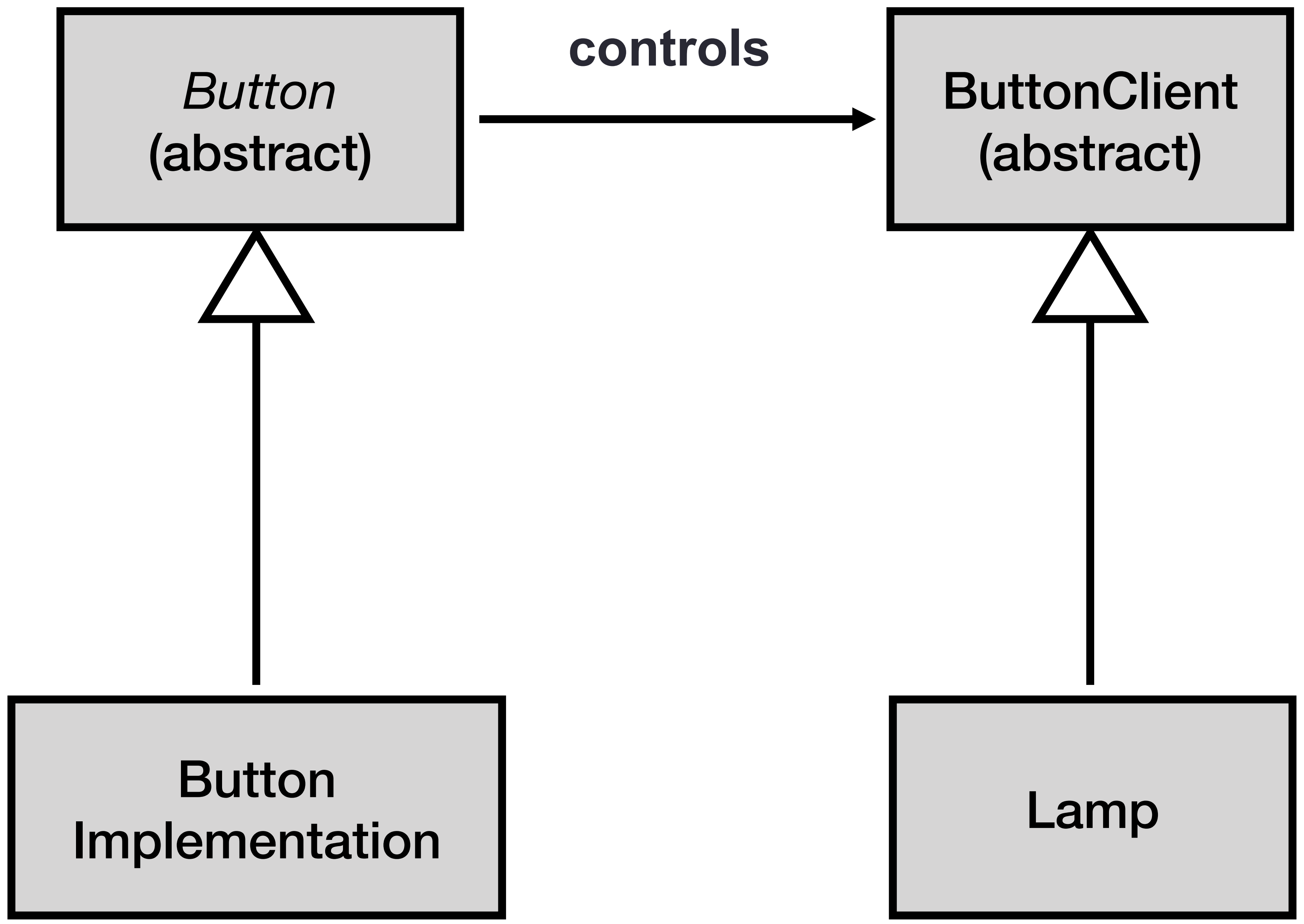
Principles

Dependency Inversion Principle (DIP)

- **The DIP requires that high level modules should not depend on low level modules, both should depend on abstraction. Also, abstraction should not depend on details, details should depend on abstractions.**

- Making a class Button associate to another class Lamp (because a Lamp has a Button) is a violation of DIP

- A better design will be associate an AbstractButton with an AbstractButtonClient, and define Button as a subclass of the AbstractButton and a Lamp a subclass of the AbstractButtonClient



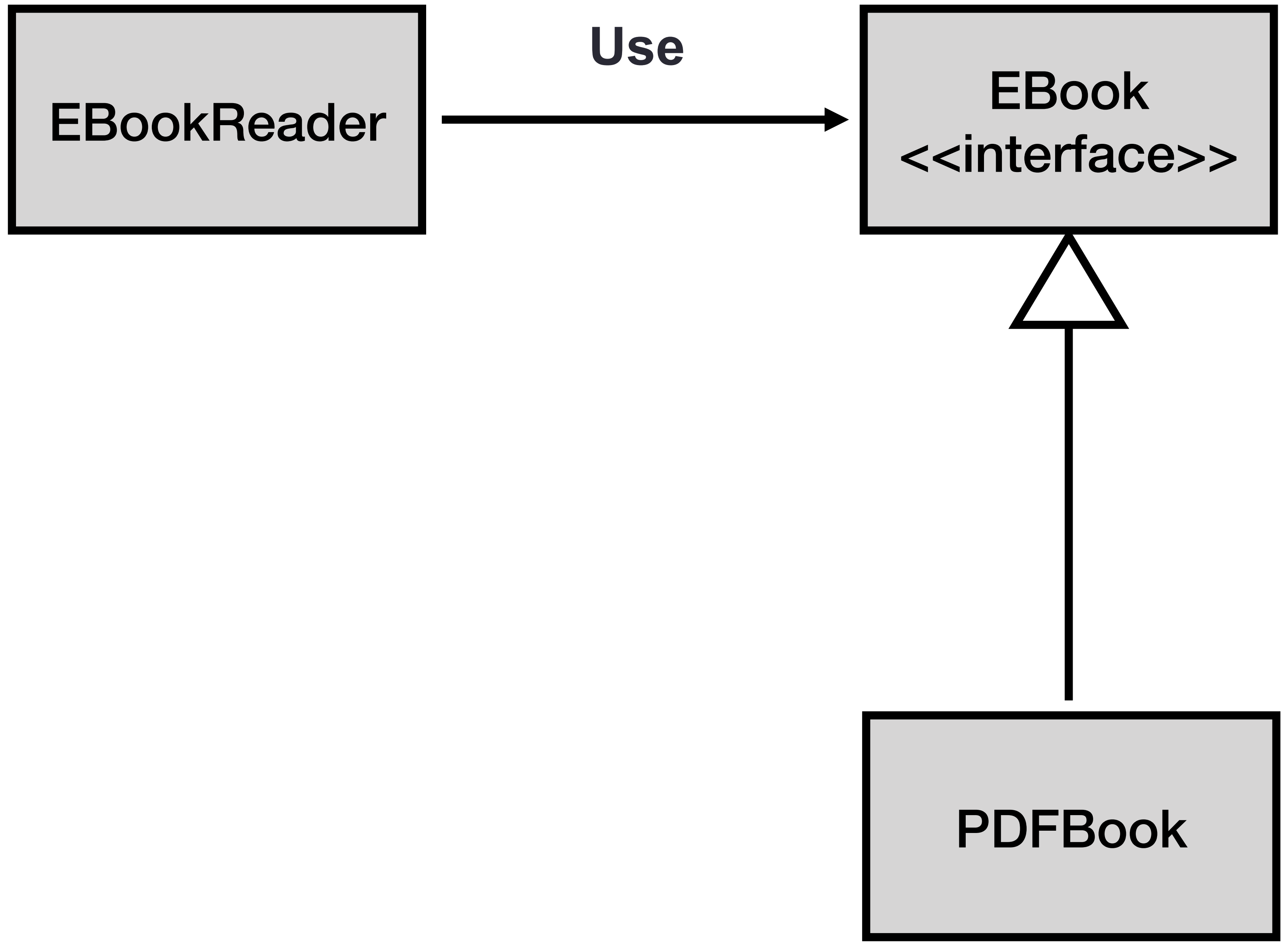
Dependency Inversion Principle (DIP)

- **The DIP requires that high level modules should not depend on low level modules, both should depend on abstraction. Also, abstraction should not depend on details, details should depend on abstractions.**

- Making an EBookReader class to use PDFBook class is a violation of DIP because it requires to change the EBookReader class to read other types of e-books.

- A better design is to let EBookReader use an interface EBook and let PDFBook and other types of e-book classes implement EBook.

- Now adding or changing e-book classes will not require any change to EBookReader class.

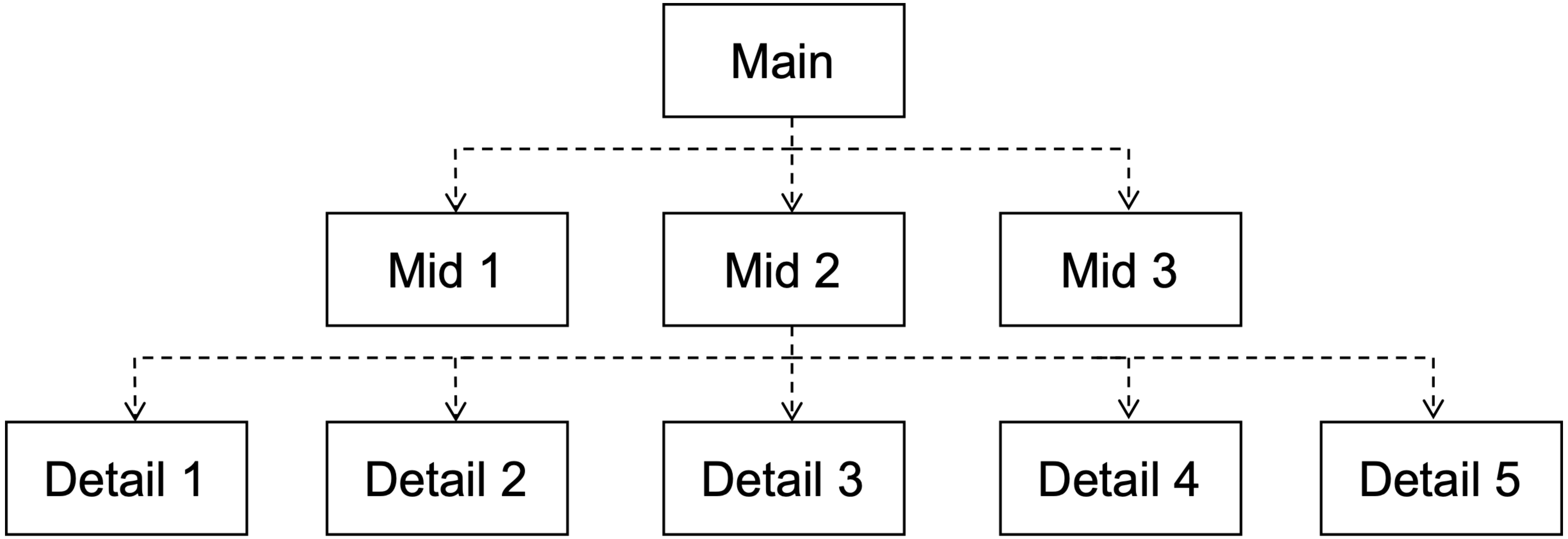


Dependency Inversion Principle (DIP)

- **The DIP requires that high level modules should not depend on low level modules, both should depend on abstraction. Also, abstraction should not depend on details, details should depend on abstractions.**

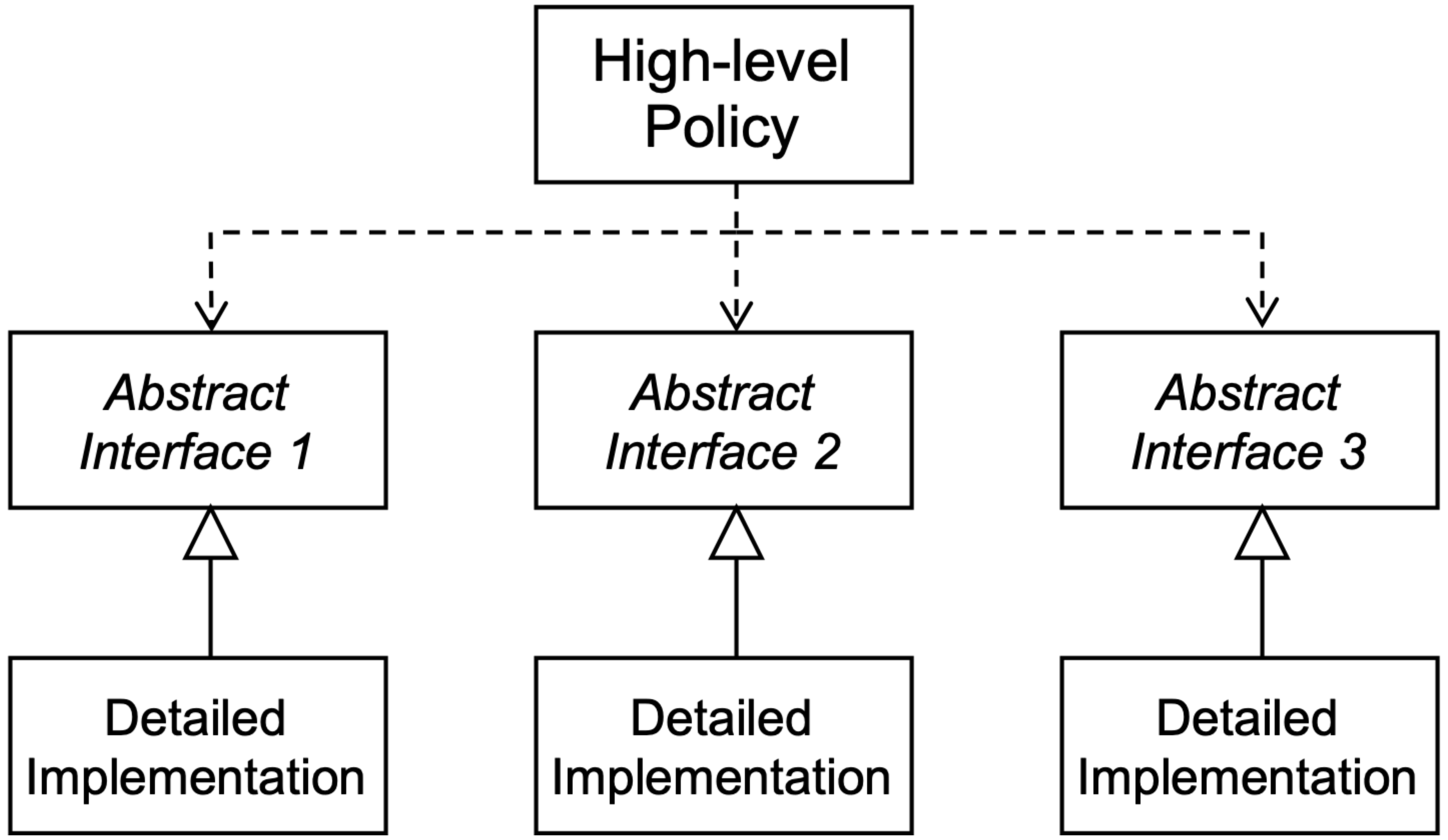
- Open-closed principle is the most important principle (according to Robert Martin)
- Dependency Inversion Principle provides the mechanism for getting OCP to work
- DIP states that our strategy should be to depend on interfaces — abstract functions and abstract classes — rather than concrete functions or concrete classes
- It is called the DIP by contrast with typical procedural design:
  - High-level modules (functions) know about and thus depends on lower-level ones (that they call), and these in turn depend on still lower-level ones

Dependency Inversion Principle (DIP)



- Procedural Design
  - High-level modules (functions) know about and thus depends on lower-level ones (that they call), and these in turn depend on still lower-level ones

Dependency Inversion Principle (DIP)



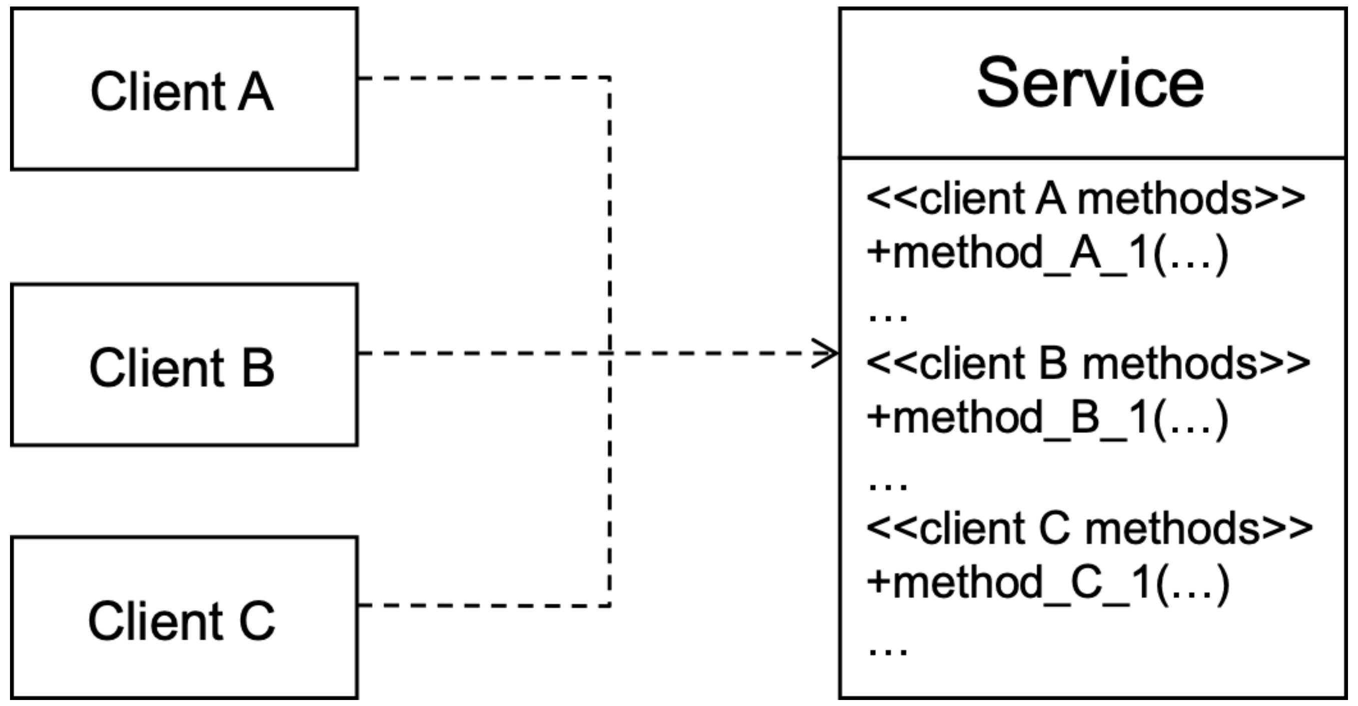
- Dependency structure of an OO System designed according to DIP
  - We should follow DIP as much as possible:
    - Abstract things change infrequently, whereas concrete implementations can change often over system’s lifetime
    - Abstraction provide hinge-points in a design, where the design can bend or be extended, via the OCP (in concert with the LSP)

Interface Segregation Principle (ISP)

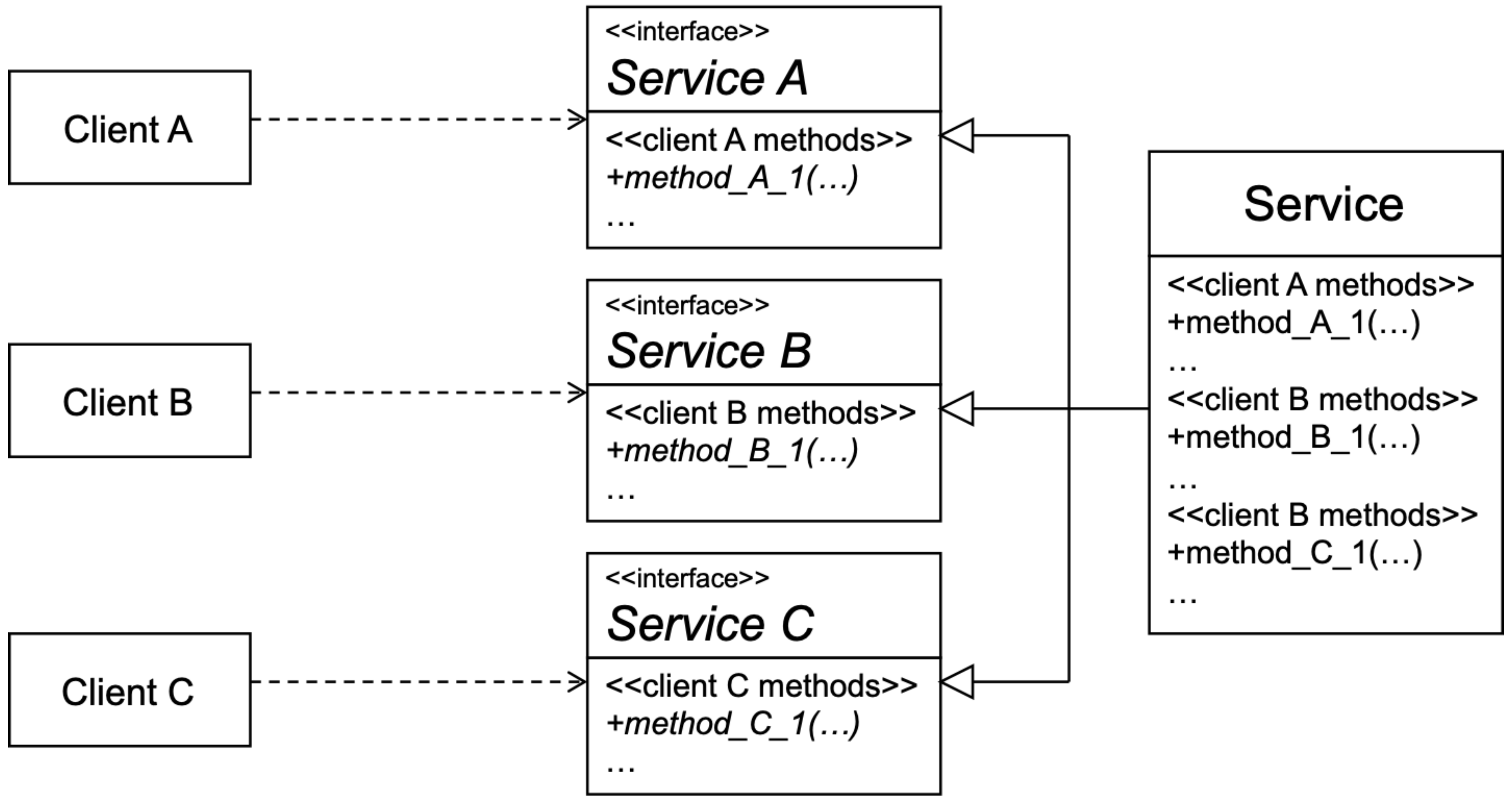
- **The ISP requires that clients should not be forced to depend on interfaces that they do not use**
  - If you have a module that has several clients that use it differently, create a separate, abstract, interface for each client

  - This removes implicit coupling between the clients, that is present due to shared interface

Example of a fat-service, that has a single large interface to serve all of its clients

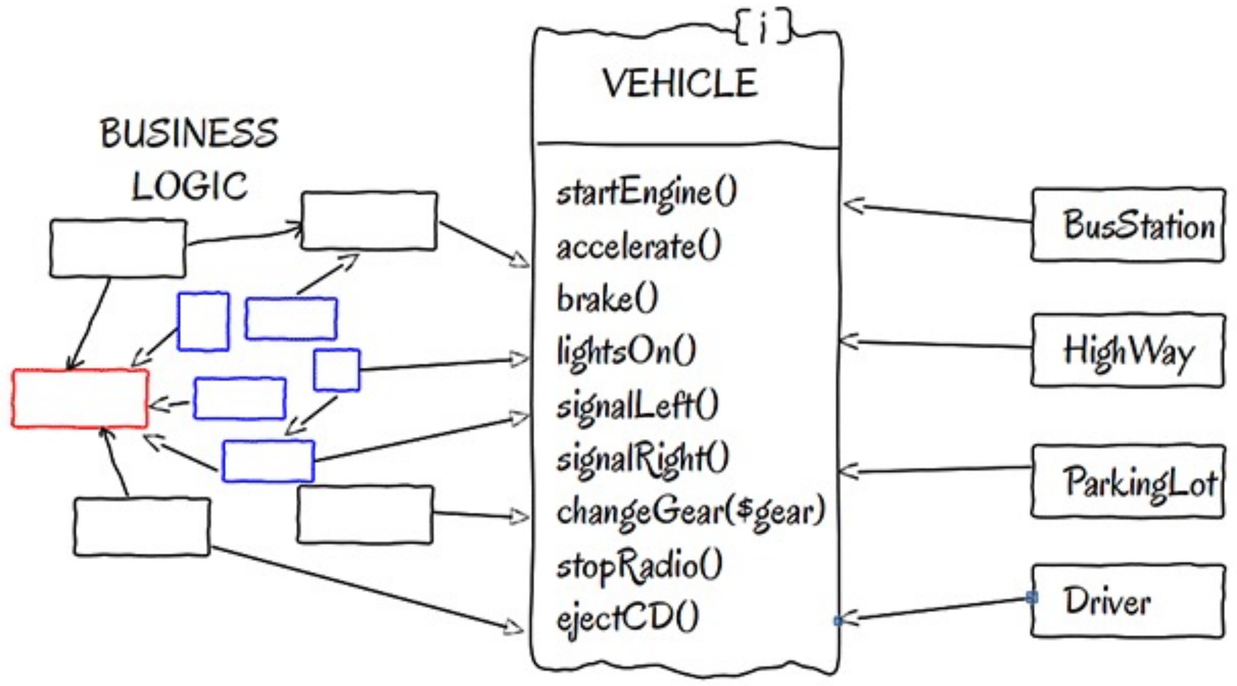


Improved Design using ISP

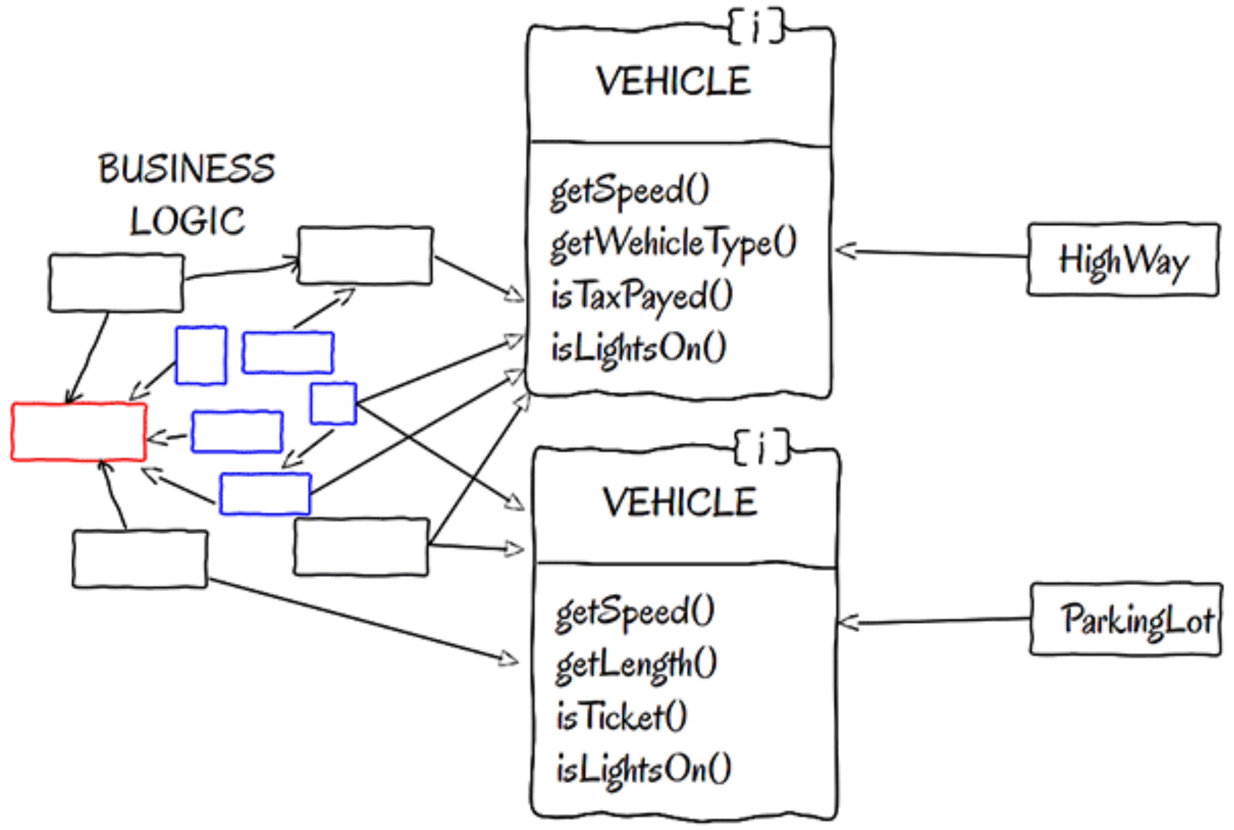


Interface Segregation Principle (ISP)

- Suppose a Vehicle interface shown in the figure is designed for clients to use



- This violates ISP because clients are forced to depend on methods they do not use: HighWay does not use stopRadio() or ejectCD(), and ParkingLot does not need accelerate() or ejectCD().
- A better design is to design smaller interfaces for different types of clients as shown in the following figure



Interface Segregation Principle (ISP)

- **Why is design using ISP better?**
  - In the original design, all the clients depended directly on the service class
  - Any change in the interface serving a single client would require a change in Service
  - This would then trigger the recompilation of all the clients’ s they all depend on that Service

- The ISP design removes implicit dependency between all clients due to the separate interface
  - A change in the interface fo a single client no longer means that all clients have to he recompiled and re-deployed

Single Responsibility Principle (SRP)

- **A class should have only a single responsibility**
  - If a class SalesOrder keeps information about a sales order, and in addition has a method saveOrder() that saves the SaleOrder in a database and a method exportXML() that exports the SalesOrder in XML format, this design will violate the SRP because there will be different types of users of this class and different reasons for making changes to this class.
  - A change made for one type of user, say change the type of database, may require the re-test, recompilation, and re-linking of the class for the other type of users.
- A better design will be to have the SalesOrder class only keeps the information about a sales order, and have different classes to save order and to export order, respectively. Such a design will confirm to SRP

Robert Martin’s Collected Principles

- Open-Closed Principle ✅
- Liskov Substitution Principle ✅
- Dependency Inversion Principle ✅
- Interface Segregation Principle ✅
- Single Responsibility Principle ✅
- Common Closure Principle
- Common Reuse Principle
- Acyclic Dependencies Principle
- Stable Dependencies Principle
- Stable Abstractions Principle
- Reuse/Release Equivalency Principle

- Cover several of the ideas in this lecture
  - Available with many other excellent articles on OO Design at http://www.objectmentor.com/
  - http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf