Winter Semester 17/18

Software Engineering Design & Construction

Dr. Michael Eichberg
Fachgebiet Softwaretechnik
Technische Universität Darmstadt

Open-Closed Principle

O*pen-****C****losed* ***P****rinciple*

Software entities (classes, modules, functions, components, etc.) should be ***open for extension****, but* ***closed for modifications****.*

– Object-Oriented Software Construction; 2nd Edition; Bertand Meyer, 1997

–Agile Software Development; Robert C. Martin; Prentice Hall, 2003

Reasons for closing modules against changes/ for making modules open for extension

- The module was delivered to customers and a change will not be accepted. If you need to change something later, hopefully you opened your module for extension!

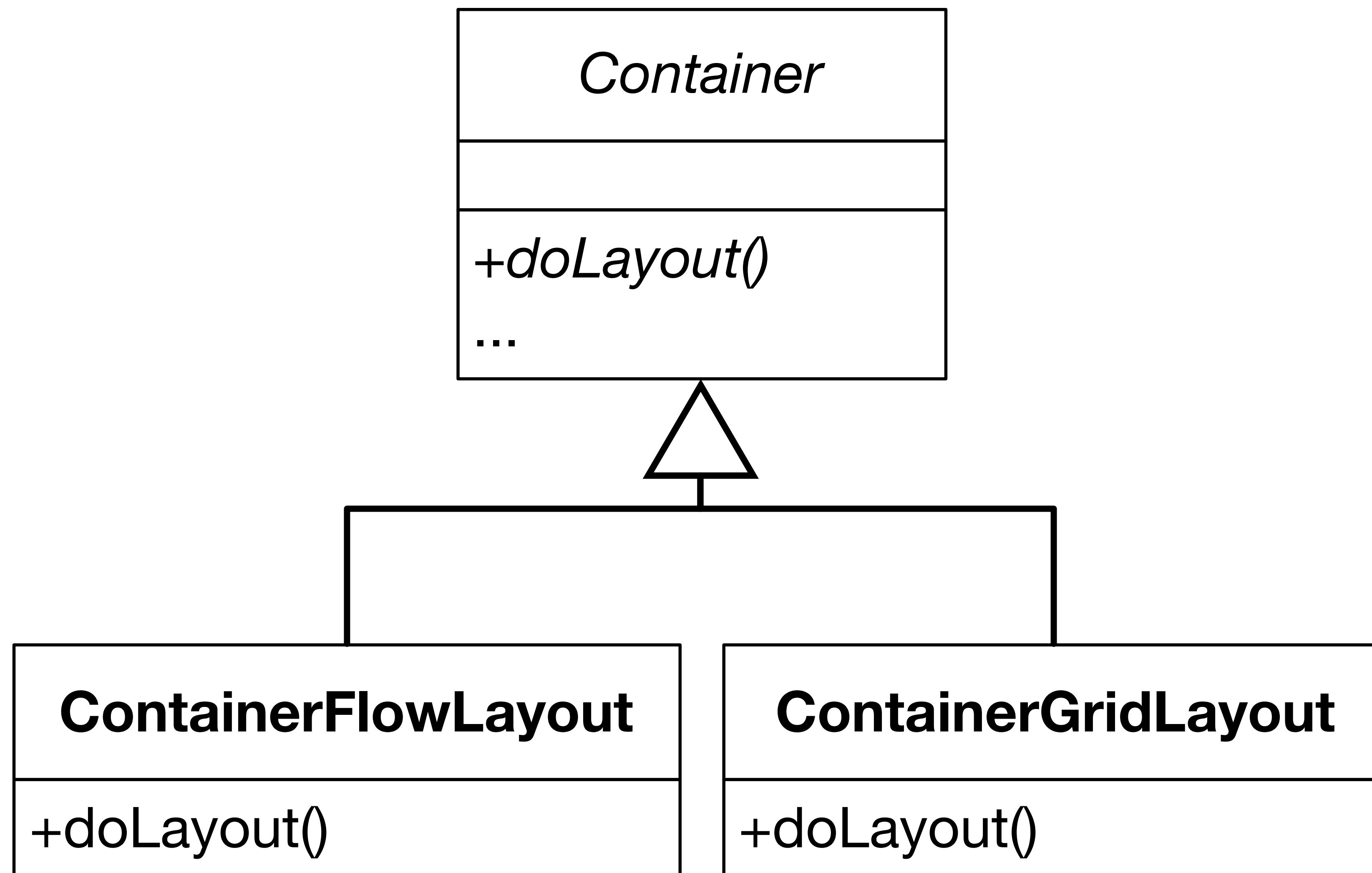
- The module is a third-party Library/Framework and only available as binary code. If you need to change something, hopefully the third-party opened the module for extension!

- Not having to change existing code means **modular compilation**, **testing** and **debugging**.

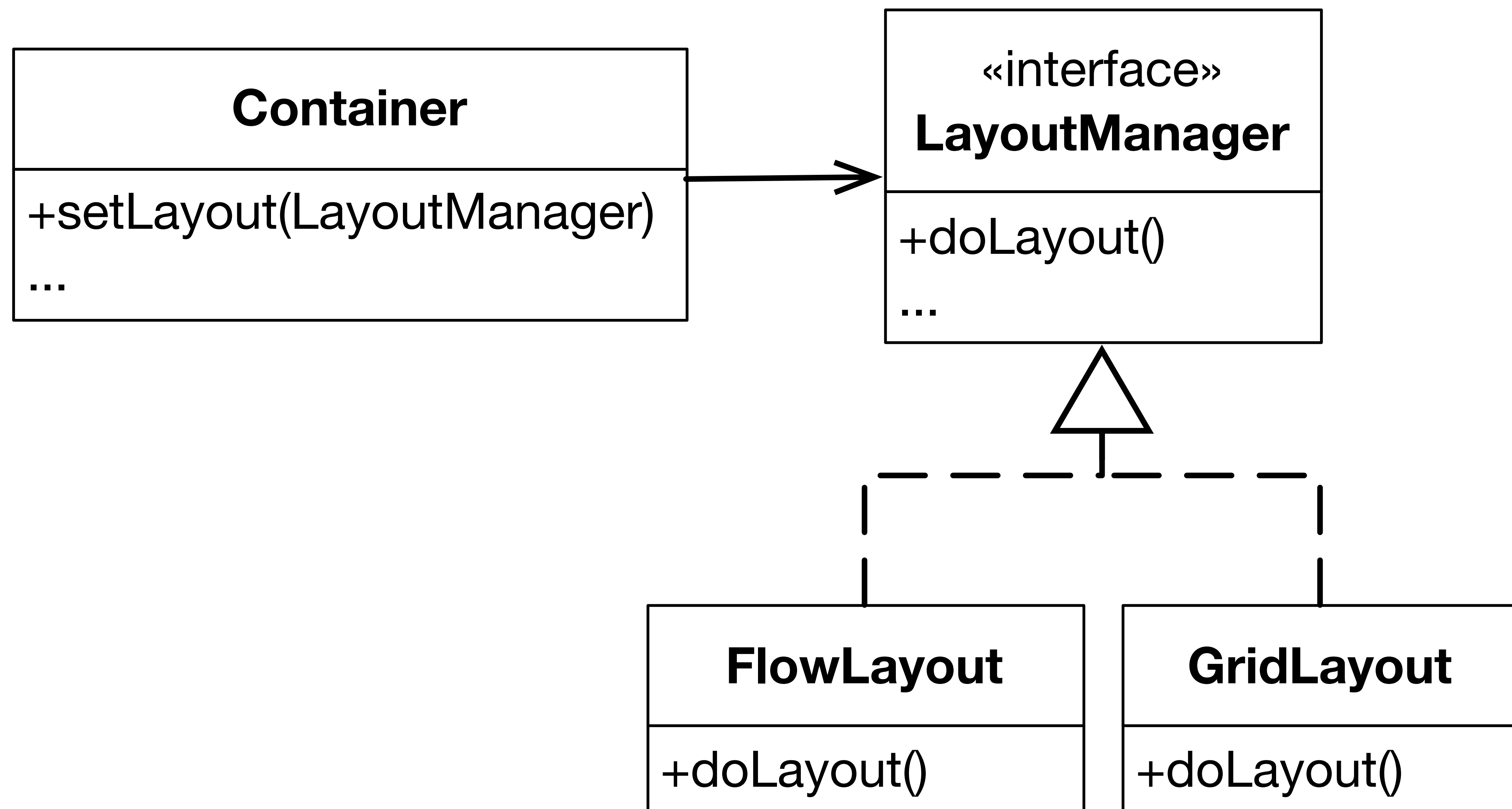
Abstraction is the Key

To enable extending an entity without modifying it, **abstract over subparts** of its behavior.

Abstracting Over Variations I

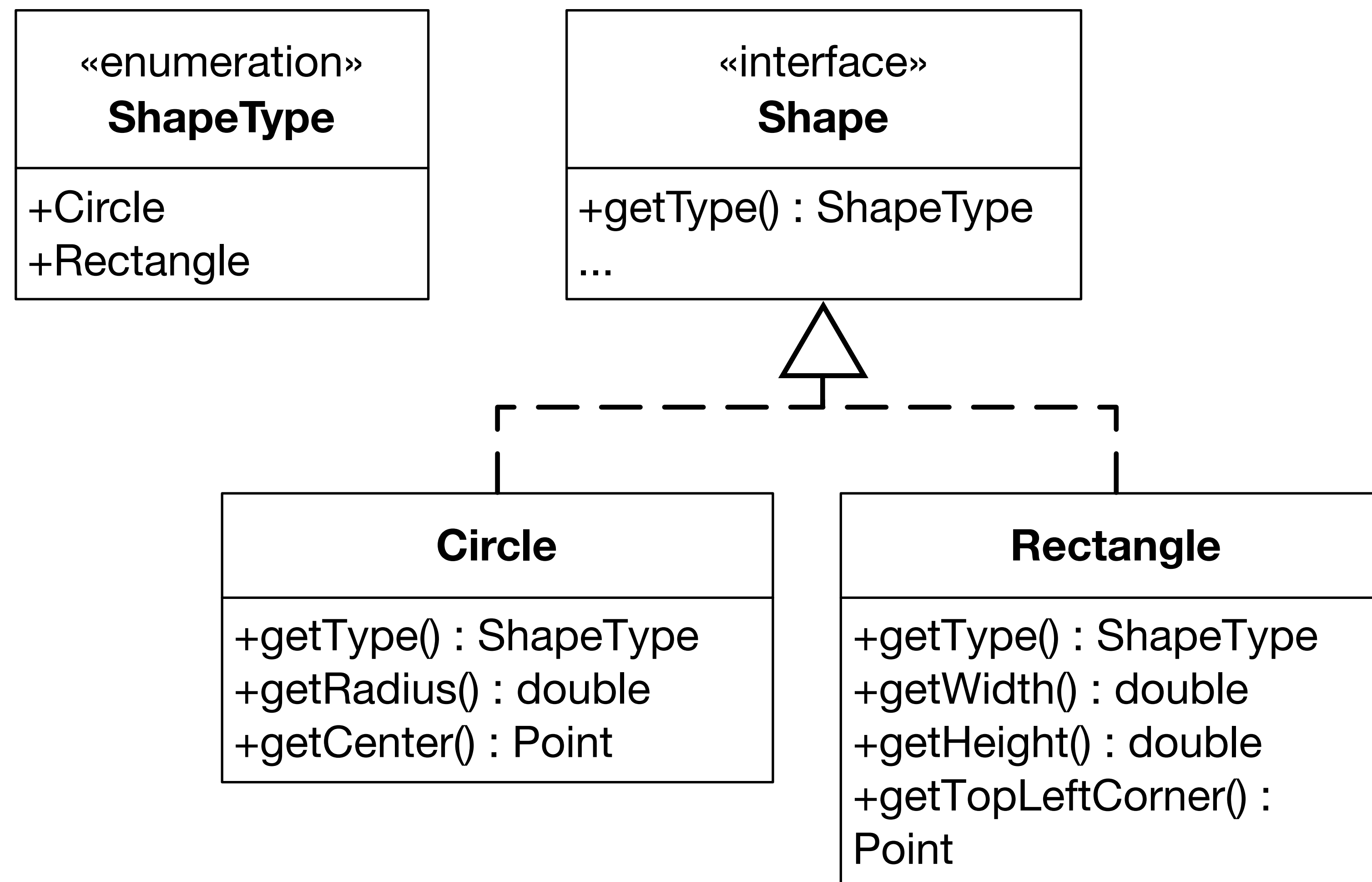


Abstracting Over Variations II

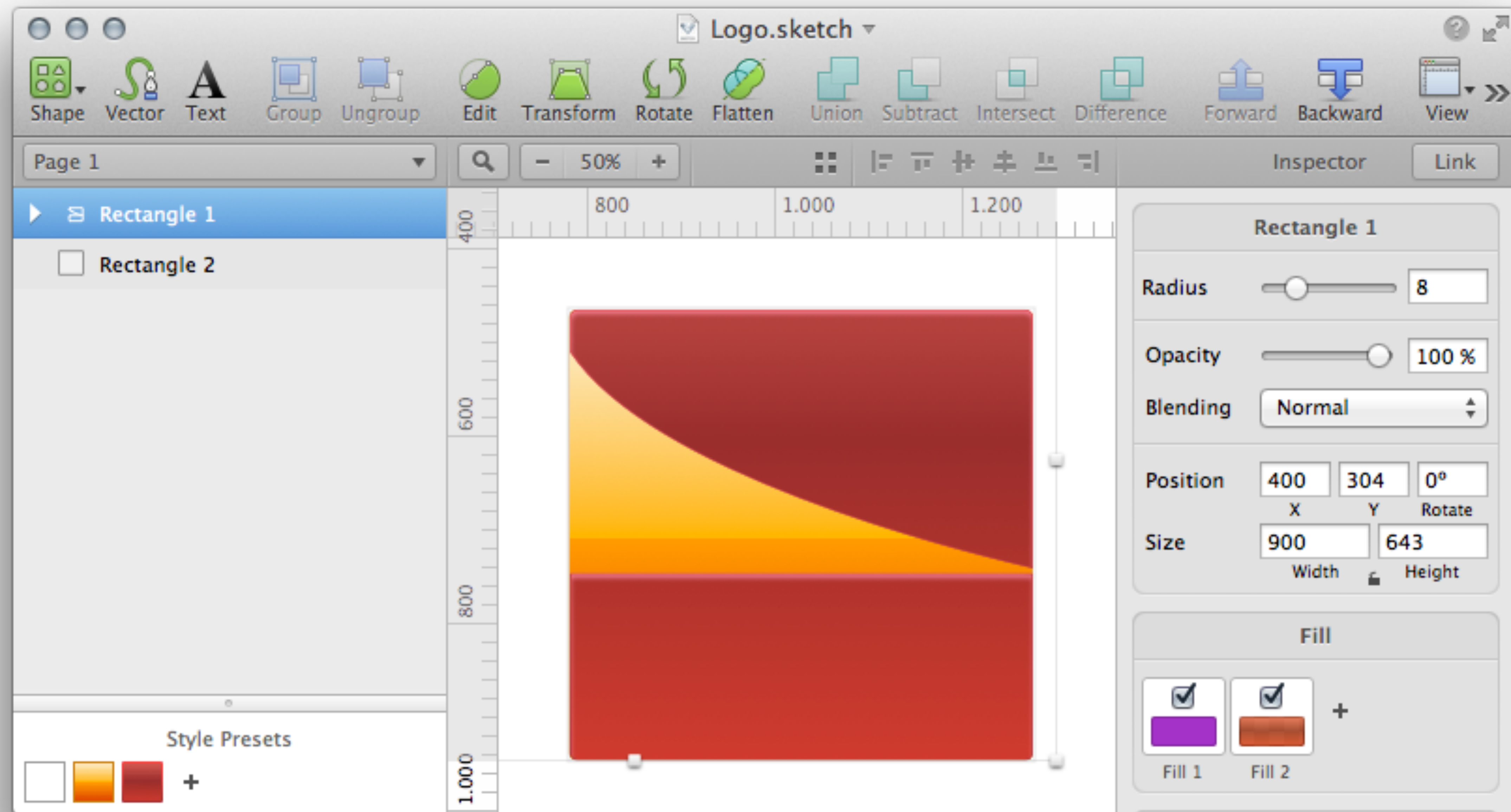


A Possible Design for Drawable Shapes

Drawing is implemented in separate methods (e.g., of class `Application`).



Consider an application that draws shapes - circles and rectangles – on a standard GUI.



Implementation of the Drawing Functionality

```
class ShapeDrawer {
  public void drawAllShapes(List<Shape> shapes) {
    for(Shape shape : shapes) {
      switch(shape.getType()) {
      case Circle:
        drawCircle((Circle)shape);
        break;
      case Rectangle:
        drawRectangle((Rectangle)shape);
        break;
  } } }

  private void drawCircle(Circle circle) { ... }

  private void drawRectangle(Rectangle rectangle) { ... }
}
```

Implementation of the Drawing Functionality

```
class ShapeDrawer {
  public void drawAllShapes(List<Shape> shapes) {
    for(Shape shape : shapes) {
      switch(shape.getType()) {
      case Circle:
        drawCircle((Circle)shape);
        break;
      case Rectangle:
        drawRectangle((Rectangle)shape);
        break;
  } } }

  private void drawCircle(Circle circle) { ... }

  private void drawRectangle(Rectangle rectangle) { ... }
}
```

In which cases is such a design Ok?

Assessing Designs

- **Rigid designs** are hard to change – every change causes many changes to other parts of the system.
- **Fragile designs** tend to break in many places when a single change is made.
- **Immobile designs** contain parts that could be useful in other systems, but the effort and risk involved with separating those parts from the original system are too big.

Some Terminology

Evaluating the Design

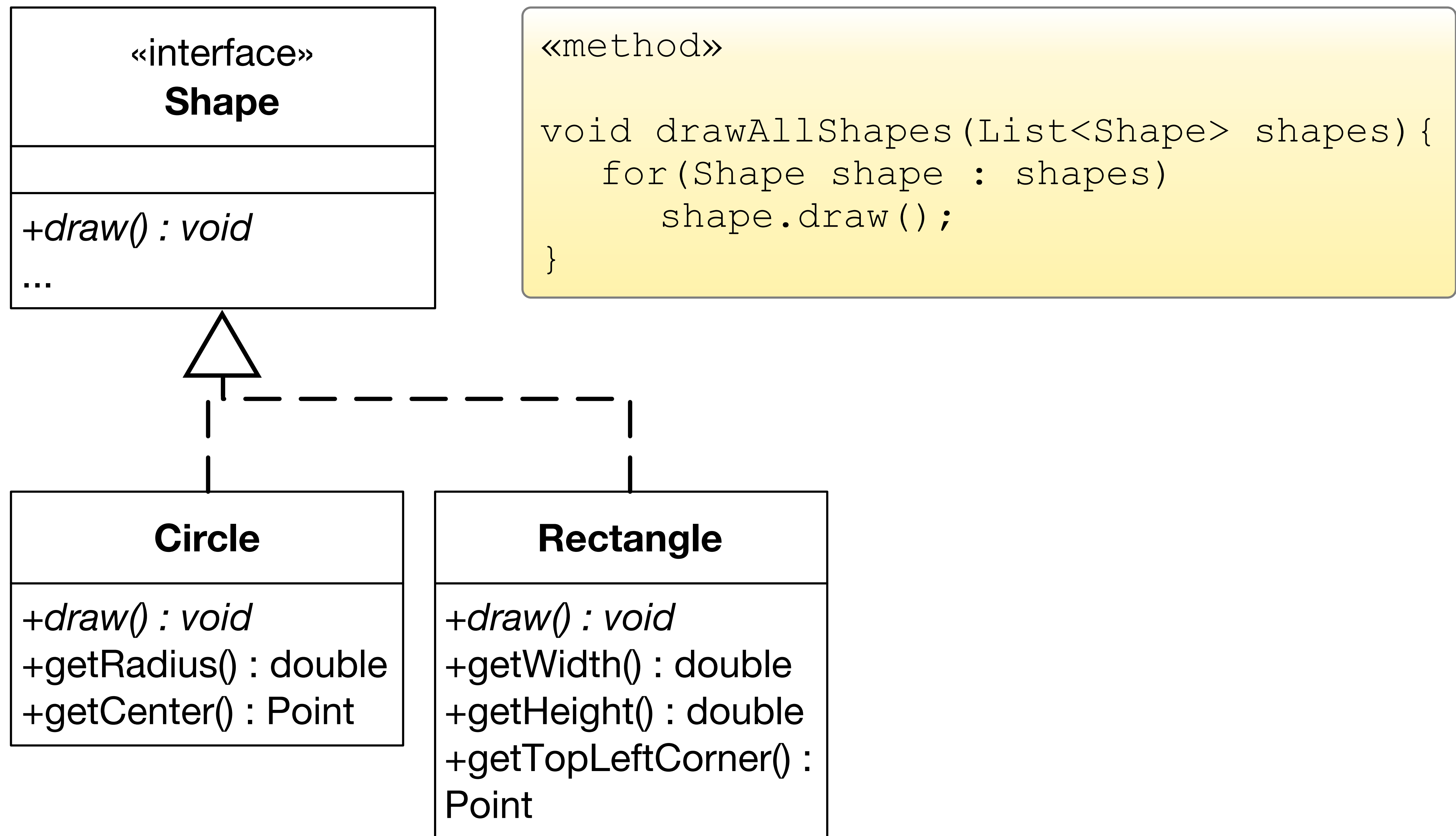
Our design is rigid, fragile and immobile.

- The proposed design **violates the open-closed design principle** with respect to extensions with new kinds of shapes.

- We need to close our module against this kind of change by building appropriate abstractions.

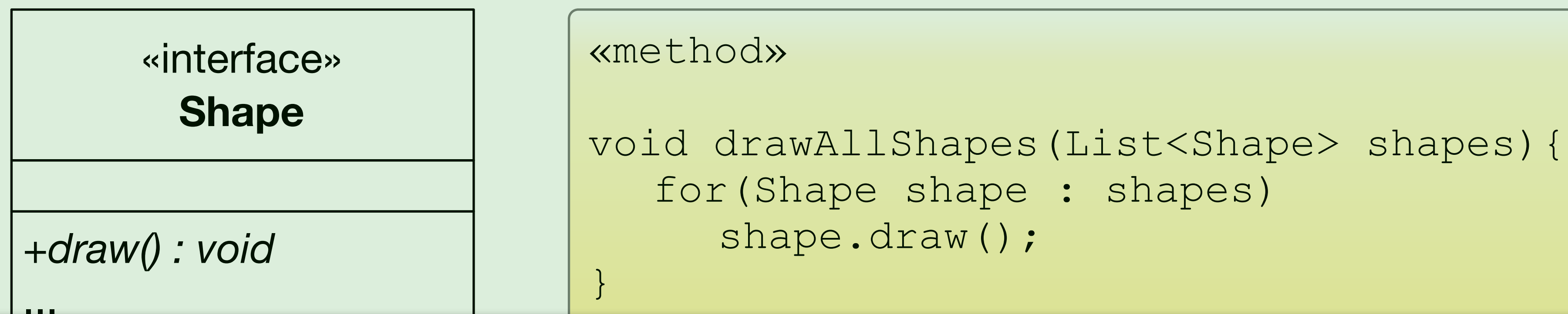
```
class Application {
  public void drawAllShapes(List<Shape> shapes) {
    for(Shape shape : shapes) {
      switch(shape.getType()) {
      case Circle:
        drawCircle((Circle)shape);
        break;
      case Rectangle:
        drawRectangle((Rectangle)shape);
        break;
  } } }

  private void drawCircle(Circle circle) { ... }

  private void drawRectangle(Rectangle rectangle) { ... }
}
```

Refined Design for Drawable Shapes

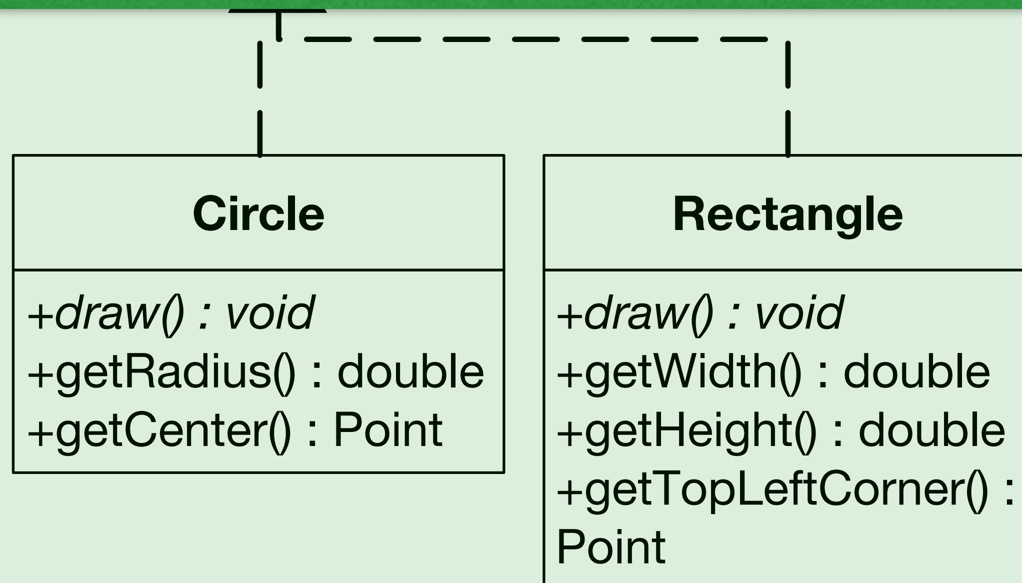


Evaluating the Extensibility

Refined Design for Drawable Shapes

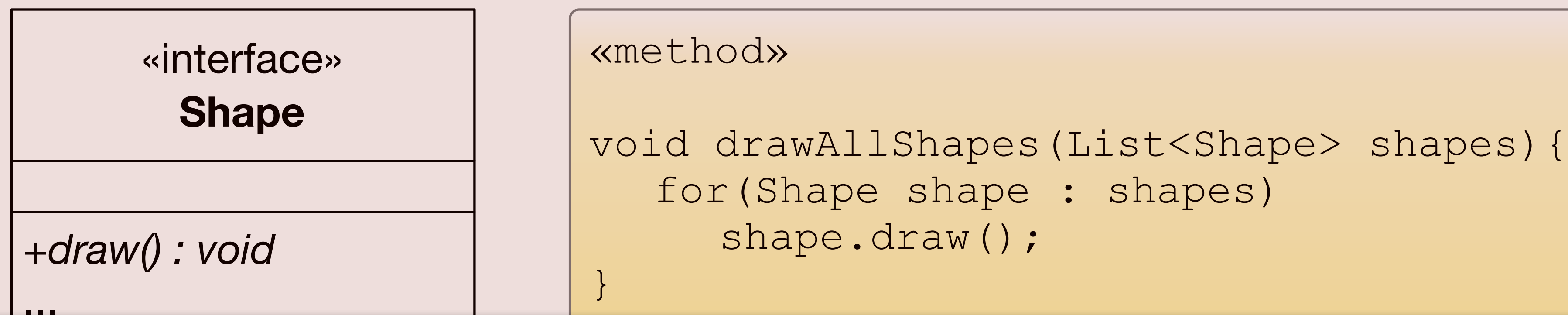


This solution complies to the open-closed design principle.



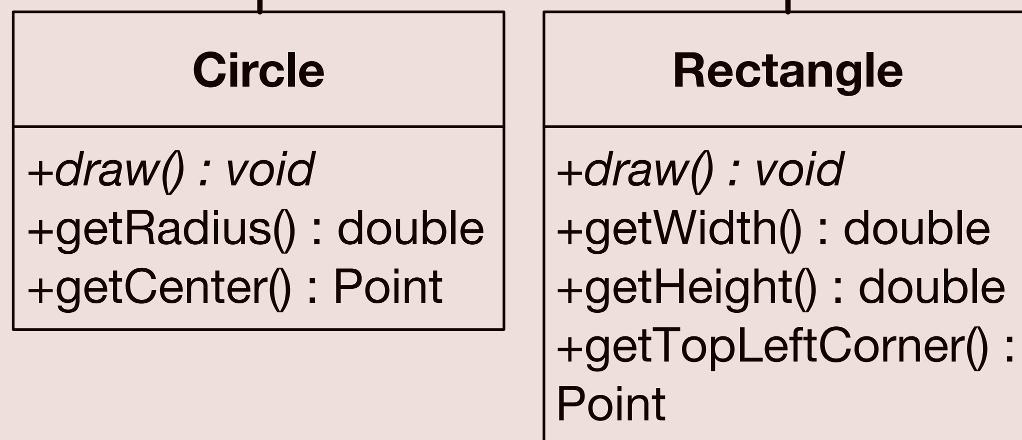
Evaluating the Extensibility

Refined Design for Drawable Shapes



This solution complies to the open-closed design principle.

These abstractions are more of an hindrance to several other kinds of changes.



Abstractions May Support or Hinder Change!

- Change is easy if change units correspond to abstraction units.

- Change is tedious if change units do not correspond to abstraction units.

Abstractions Reflect a Viewpoint

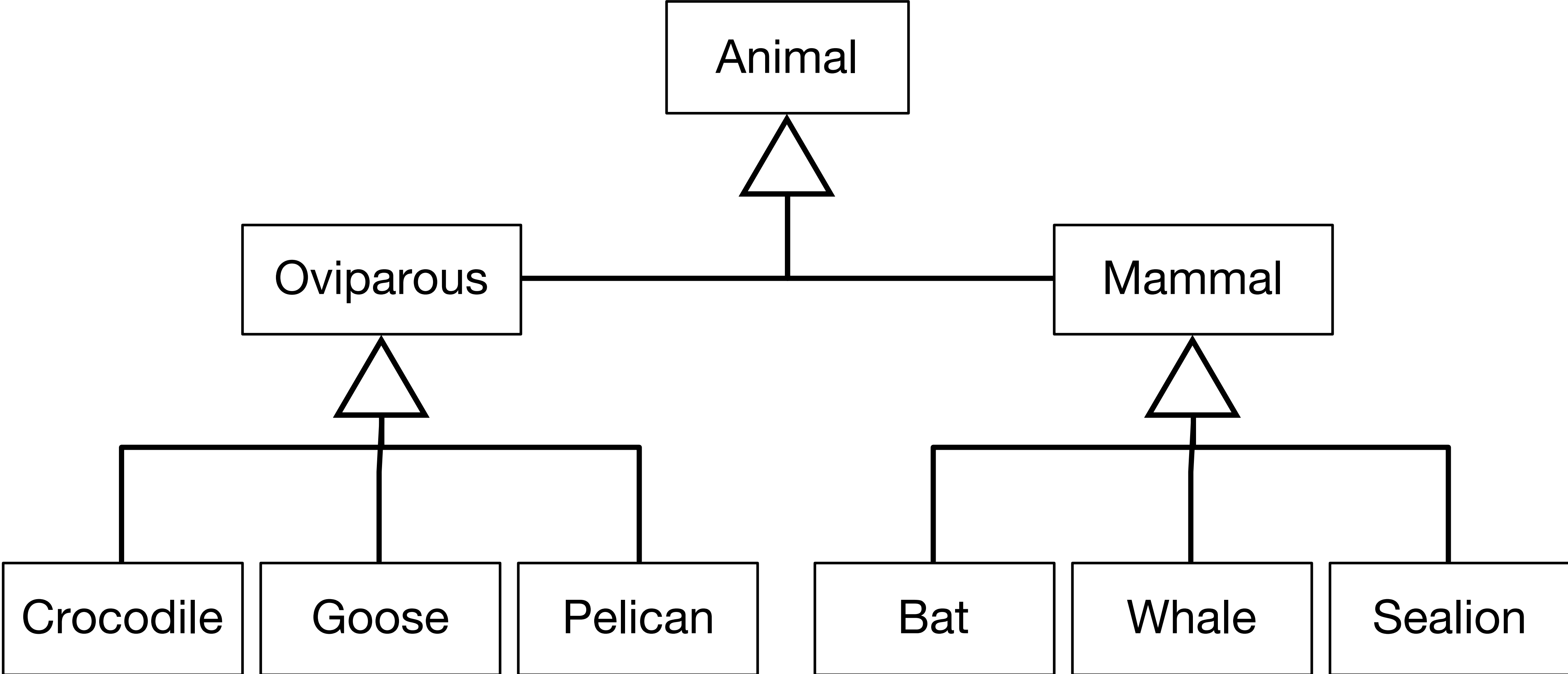
No matter how "closed" a module is, there will always be some kind of change against which it is not closed.

Imagine: Development of a "Zoo Software"

On the "Natural" Model Structure

- Three stakeholders:
  - Veterinary surgeon:What matters is how animals **reproduce**!
  - Trainer: What matters is the **intelligence**!
  - Keeper: What matters is what they **eat**!

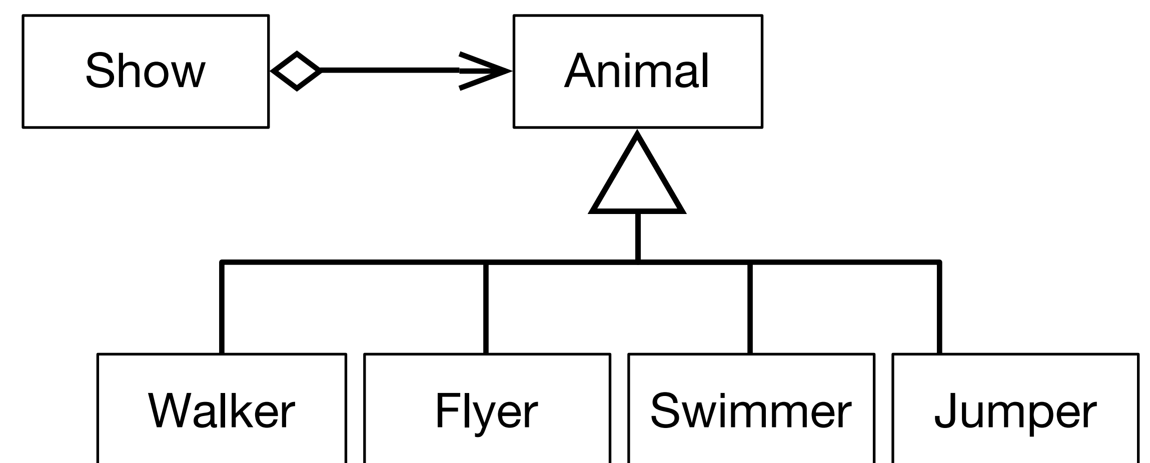
One Possible Class Hierarchy When Modeling Animals



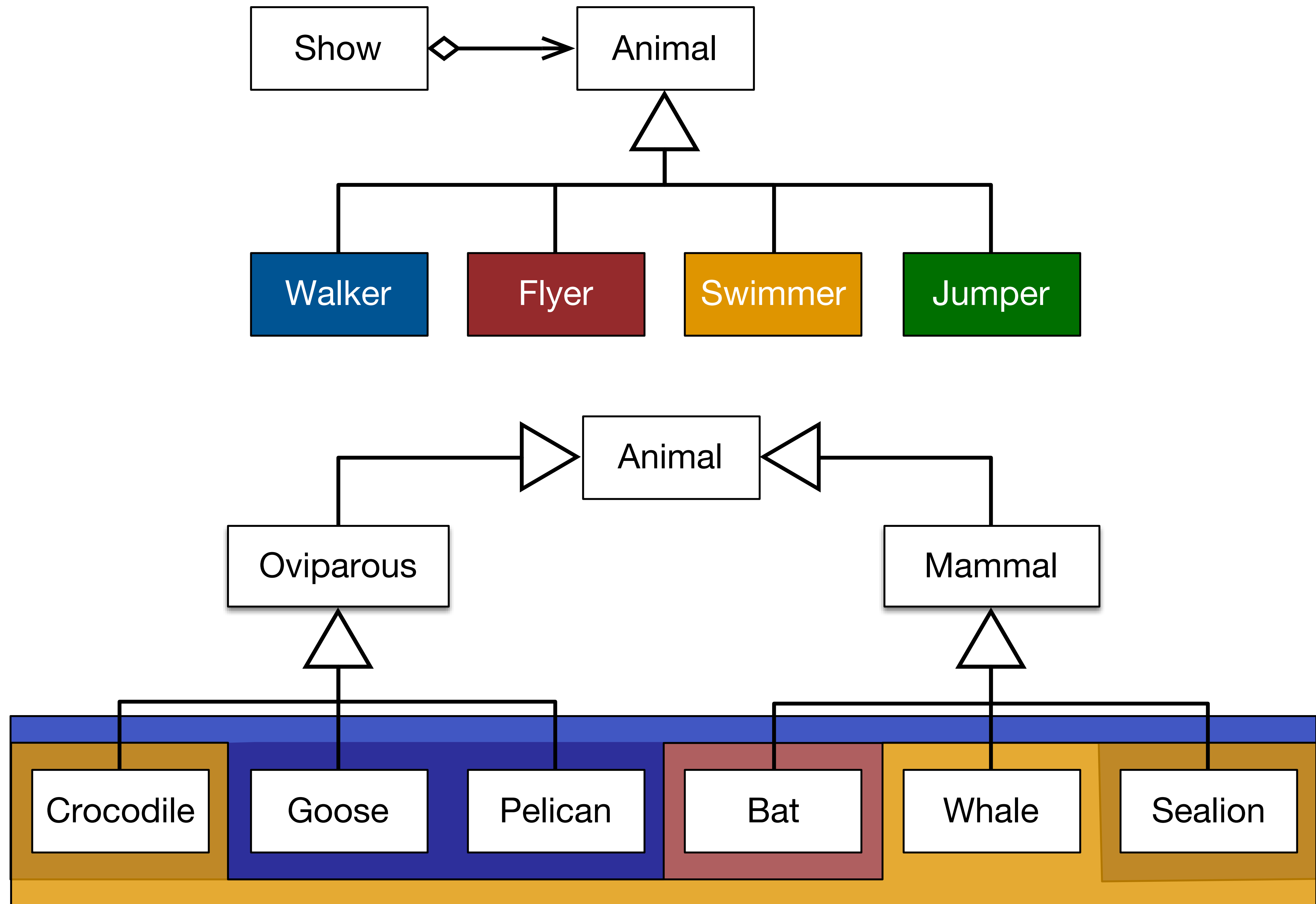
The Animal World From a Trainer's Viewpoint

The Show

The show shall start with the pink pelicans and the African geese **flying** across the stage. They are to land at one end of the arena and then **walk** towards a small door on the side. At the same time, a killer whale should **swim** in circles and jump just as the pelicans fly by. After the jump, the sea lion should swim past the whale, **jump** out of the pool, and walk towards the center stage where the announcer is waiting for him.



Models Reflecting Different Viewpoints Overlap



Most programming languages (e.g., Java) and tools do not well support the modeling of the world based on co-existing viewpoints.

No matter how “closed” a module is, there will always be some kind of change against which it is not closed.

Strategic Closure

- Choose the kinds of changes against which to close your module.
    - Guess at the most likely kinds of changes.
    - Construct abstractions to protect against those changes.
- Prescience derived from experience:
    - Experienced designers hope to know the user and an industry well enough to judge the probability of different kinds of changes.
    - Invoke open-closed principle against the most probable changes.

Be Agile

Recall that guesses about the likely kinds of changes to an application over time will often be wrong.

- Conforming to the open-closed principle is expensive:
  - Development time and effort to create the appropriate abstractions.
  - Created abstractions might increase the complexity of the design.
  - Needless, accidental complexity.
  - Incorrect abstractions supported/maintained even if not used.
- Be agile: Wait for changes to happen and close against them.

O*pen-****C****losed* ***P****rinciple*

- Abstraction is the key to supporting the open-closed design principle.
- No matter how closed a module is, there will always be some kind of change against which it is not closed.

– Object-Oriented Software Construction; 2nd Edition; Bertand Meyer,1997
– Agile Software Development; Robert C. Martin; Prentice Hall, 2003