Winter Semester 16/17

Software Engineering Design & Construction

Dr. Michael Eichberg
Fachgebiet Softwaretechnik
Technische Universität Darmstadt

Open-Closed Principle

Open-Closed Principle

*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

Extension: Extending the behavior of a module.

Modification: Changing the code of a module.

Open for extension means that when requirements of the application change, we can extend the module with new behaviors that reflect those changes. We change what the module does.

Closed for modification means that changes in behavior do not result in changes in the module's source or binary code.

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.

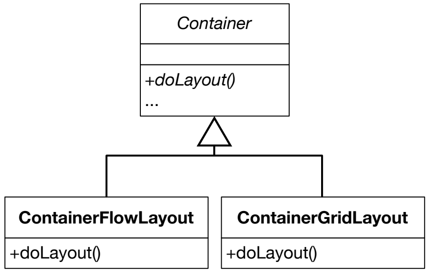
Many programming languages allow to create abstractions that are fixed and yet represent an unbounded group of possible behaviors!

Different kinds of abstraction mechanisms exist:

- **Object-oriented languages**
  - abstractions are encoded in abstract base classes resp. interfaces.
  - unbounded group of possible behaviors is represented by all the possible derivative classes resp. implementations.
- **Functional languages**
  - abstractions are encoded in function types.
  - unbounded group of possible behaviors is represented by all the possible first-class functions of the declared type.

In the following, we shortly discuss the two main ways of abstracting over variability in object-oriented programs.

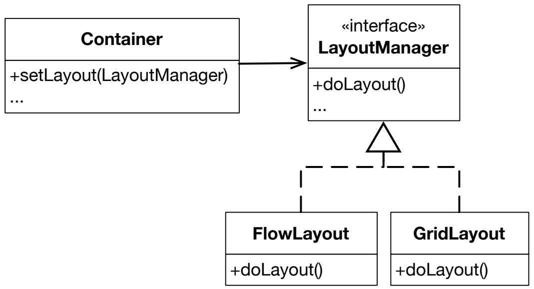
Abstracting Over Variations



- `Container` declares the layout functionality but does not implement it. The rest of `Container` is implemented against the abstraction.
- Concrete subclasses fill in the details over which `Container`'s implementation abstract.

Abstraction by means of inheritance.

Abstracting Over Variations

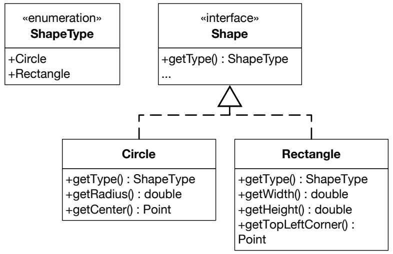


- `Container` delegates the layout functionality to an abstraction. The rest of its functionality is implemented against this abstraction.
- To change the behavior of an instance of `Container` we configure it with the `LayoutManager` of our choice.
- We can add completely new behavior by implementing our own `LayoutManager`.

Abstraction by means of subtype polymorphism.

A Possible Design for Drawable Shapes

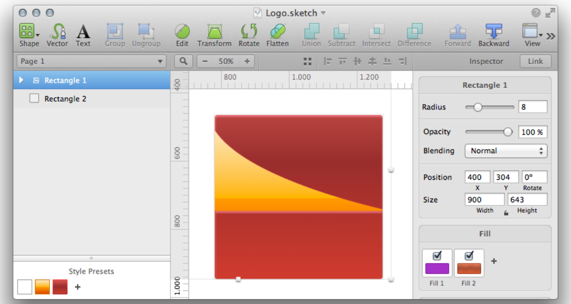
Drawing is implemented in separate methods (e.g., of class `Application`).



- Each `Shape` identifies itself via the enumeration `ShapeType`.
- Realizations of `Shape` declare specialized methods for the shape type they represent.

(Such a design may be desirable, because you don't want to pollute the interface of **Shape**/you want to have a SRP compliant solution.)

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) { ... }
}
```


Does this design conform to the open-closed design principle?

Evaluating the proposed design

- Adding new shapes is hard, we need to:
  - Implement a new realization of `Shape`.
  - Add a new member to `ShapeType`.
    This possibly leads to a recompile of all other realizations of `Shape`.
  - `drawAllShapes` (and every method that uses shapes of different types) must be changed.
    We have to hunt for every place that contains conditional logic that distinguishes between types of shapes and we have to add code to it.
- `drawAllShapes` is hard to reuse! When we reuse it, we have to bring along `Rectangle` and `Circle`.

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

I.e., when we want to evaluate a design, we can ask ourselve: Does it show signs of rigidity, fragility or immobility?

Evaluating the Design

Our design is rigid, fragile and immobile.

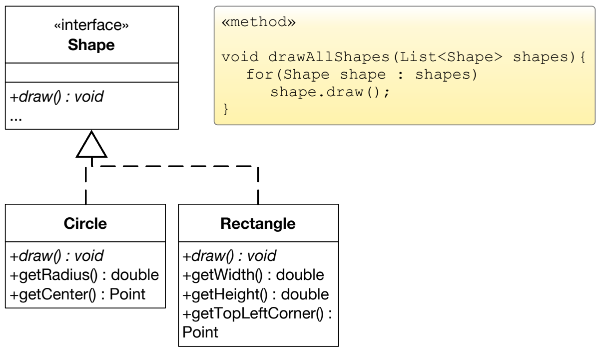
- The proposed design **violates the open-closed design principle** with respect to extensions with new kinds of shapes.
- We need to close our module against this kind of change by building appropriate abstractions.

```
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) { ... }
}
```


Assessing our design w.r.t. its rigidity, fragility and immobility:

- Our example design **is rigid**: Adding a new shape causes many existing classes to be changed.
- Our example design **is fragile**: Many switch/case (if/else) statements that are both hard to find and hard to decipher.
- Our example design **is immobile**: `drawAllShapes` and `drawXXX` is hard to reuse.

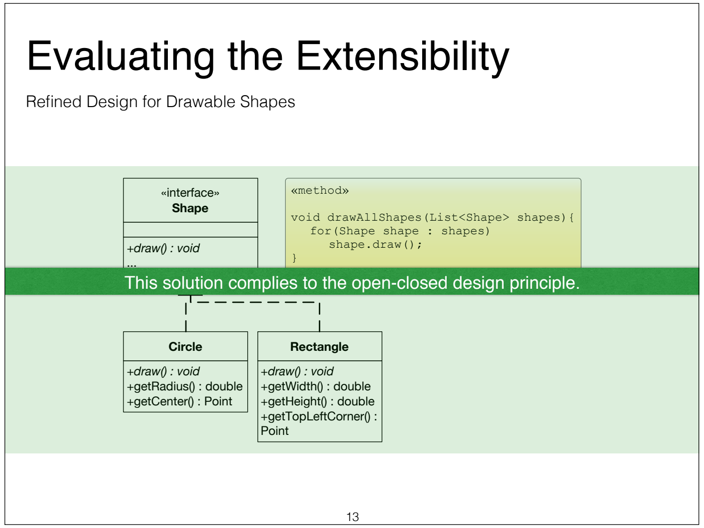
Refined Design for Drawable Shapes



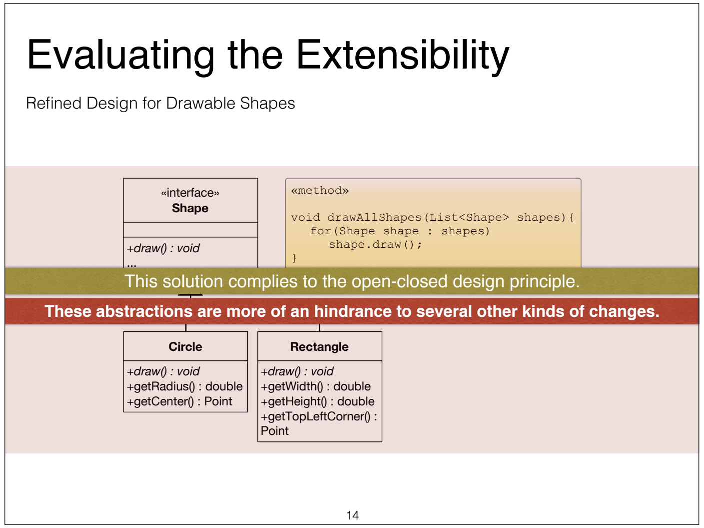
- Makes adding new shapes possible without modification.
  We just need to implement a new realization of `Shape`.

- `drawAllShapes` only depends on `Shape`.
  We could reuse it unchanged; however, it has become so trivial that there is no immediate need.

But, now `Shape` has two responsibilities: a “knowing” and a “doing” responsibility.



Do ask yourself whether this unconditional statement is true!



The statement: "*This solution complies to the open-closed design principle.*" is – of course – not unconditionally correct. It is not possible to be open for all kinds of extension and also be closed for modification.

Examples of other types of extensions:

- Consider extending the design with further shape functions:
  - shape transformations, shape dragging, …
  - calculating the intersection or union of shapes, etc.
- Consider adding support for different operating systems.
  The implementation of the drawing functionality varies for different operating systems.

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.

Here, “change” means adding or changing behavior.

In our example, adding a new type of `Shape` is easy as it is directly supported by inheritance and subtyping.

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.

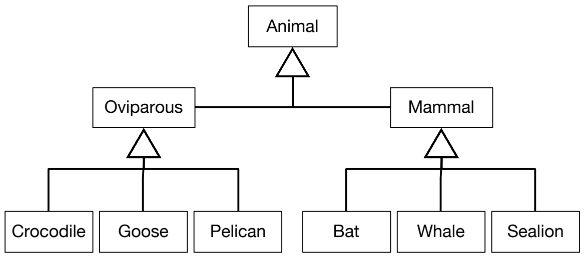
In general, there is no model that is natural to all contexts.

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



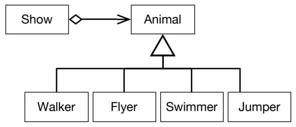
Do you see a problem? (In other words: whose viewpoint was chosen?)

When we consider the classes Oviparous and Mammal it is obvious that the class hierarchy reflects the veterinary surgeon's understanding.

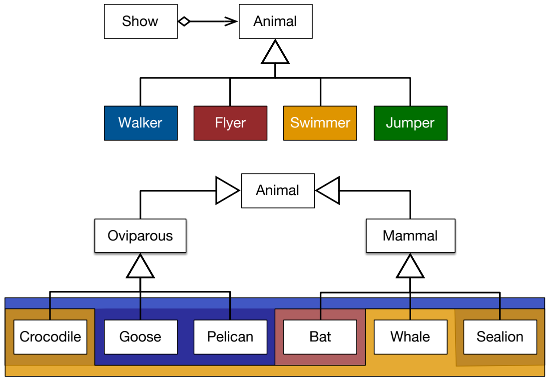
The Animal World From a Trainer's Viewpoint

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



- Elements of a category in one model correspond to several categories in the other model (and vice versa).
- Adopting the veterinary viewpoint hinders changes concerning trainer's viewpoint (and vice versa).

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.

Using a programming language which offers more advanced modeling mechanisms (such as Scala using traits), it may be possible to create a design that more closely models the presented world.

However,

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.

Open-Closed Principle

- 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