Core Design Principles

Software Design (40007) – 2023/2024

Justus Bogner
Ivano Malavolta

VRIJE
UNIVERSITEIT
AMSTERDAM

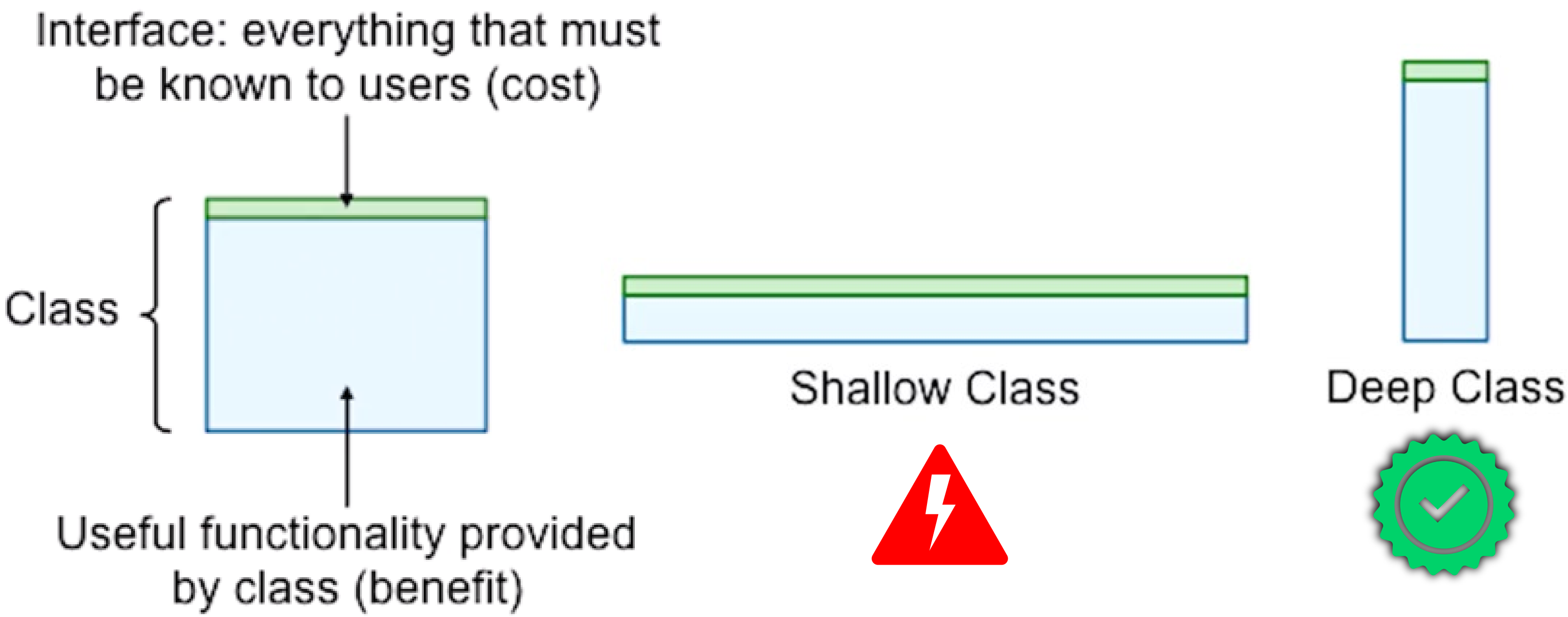
Software and Sustainability research group (S2)
Department of Computer Science, Faculty of Sciences

Roadmap

- The single responsibility principle (SRP)
- Encapsulation & immutability
- Avoid complexity (or *design for simplicity*)

How to structure your code (and class diagram)

- Strive towards having **deep classes**



Information hiding and operation usability

- Minimize the **information needs** of each class
  - Prioritize information hiding
  - `private` is your default
  - `getter()` and `setter()` methods only when needed
- Focus on the **usability** of the operations of each class
  - Exposed APIs should be easily and intuitively understandable
  - When adding an operation/parameter, think if it is really needed by the rest of the system
  - Push complexity as low as possible in the class diagram hierarchy

→ A **deep class** hides less relevant information / complexity and provides valuable and easily usable operations.

The Single Responsibility Principle (SRP)

What is the single responsibility principle (SRP)?

Design classes in such a way that there is **only a single reason to change a class.**

- Leads to smaller and more cohesive classes
- Leads to less complex classes

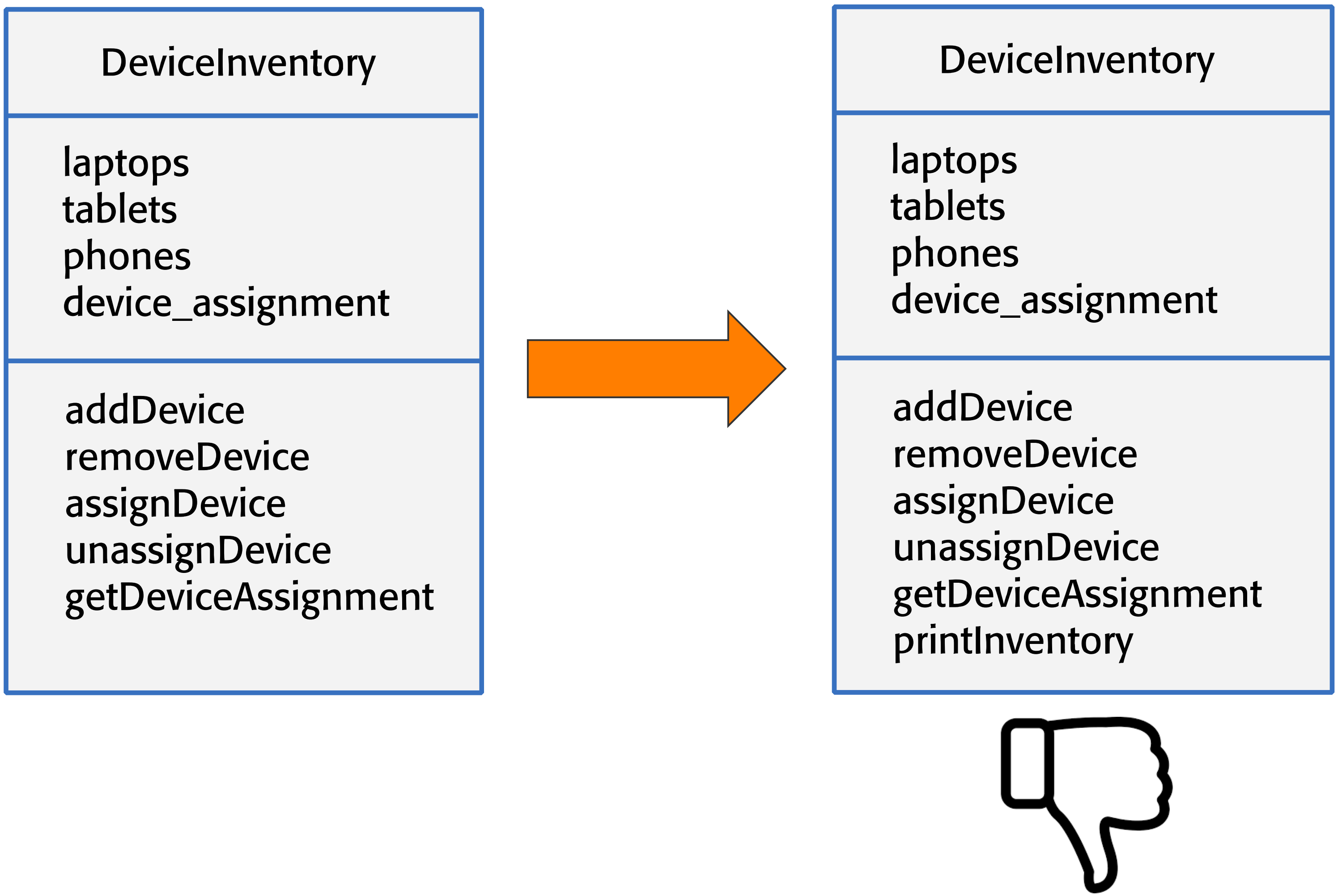
→ Classes that are easier to understand and change

- Therefore:
  - Group entities* that change for the same reasons
  - Separate entities that change for different reasons

→ **Functional cohesion**

** Entity = class, method, attribute*

Example: violating SRP



Example: preserving SRP

DeviceInventory
laptops tablets phones device_assignment
addDevice removeDevice assignDevice unassignDevice getDeviceAssignment

InventoryReport
report_data report_format
updateData updateFormat print

How are we going to ”link” those two classes?

Encapsulation & Immutability

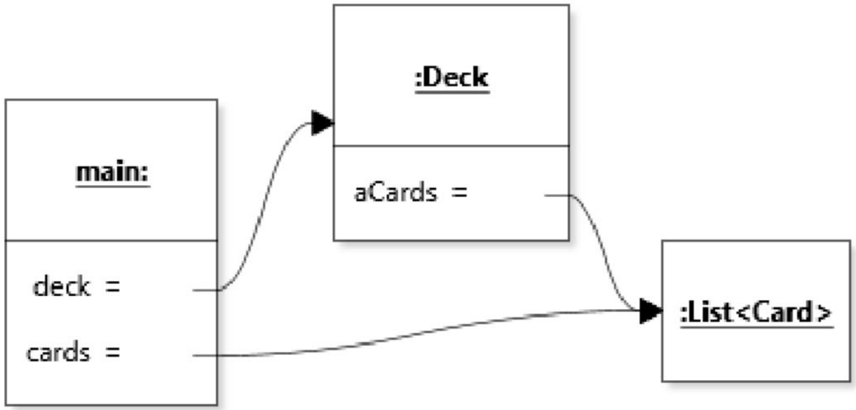
What is encapsulation?

- The act of keeping both the data and the computation together to limit the number of contact points between different parts of your system

- Closely related to information hiding
- Advantages:
  - Understanding a piece of code in isolation is easier
  - Using a piece of code becomes less error-prone
  - Changing a piece of code less likely breaks something else

Encapsulation is often violated through references that escape.

Escaping references




```java
public class Card {
    private Rank aRank;
    private Suit aSuit;

    public Card(Rank pRank, Suit pSuit) {
        aRank = pRank;
        aSuit = pSuit;
    }

    public Rank getRank() {
        return aRank;
    }

    public Suit getSuit() {
        return aSuit;
    }
}
```

```java
public class Deck  {
    private List<Card> aCards = new ArrayList<>();

    public Deck() {
        // add and shuffle cards
    }

    public Card draw() {
        return this.aCard.remove(0);
    }

    public List<Card> getCards() {
        return this.aCards;
    }
}
```

```java
Deck deck = new Deck();
List<Card> cards = deck.getCards();
cards.add(new Card(Rank.ACE, Suit.HEARTS));
```

How references escape

There are 3 ways in which references can escape:

1. Returning a reference to an external object
   - See previous slide
2. Storing an external reference internally
3. Leaking a reference through a shared structure
   - Similar to the previous ones but indirect, e.g., lists of lists

Example of 2:

```java
public class Deck  {
    private List<Card> aCards = new ArrayList<>();

    public Deck() {
        // add and shuffle cards
    }

    public void setCards(List<Card> cards) {
        this.aCards = cards;
    }
}

// ...

List<Card> cards = new ArrayList<>();
Deck deck = new Deck();
deck.setCards(cards);
cards.add(new Card(Rank.ACE, Suit.HEARTS));

```

Immutability

Objects are **immutable** if their class provides no way to change the internal state of the object after instantiation.
Immutable class = a class that yields immutable objects

Advantages:

- You can share information without breaking encapsulation
- You avoid temporal method dependencies (invocation order)
- It leads to thread safety
- It allows the caching of objects

Disadvantage:

- You tend to create more objects
- Decreased performance efficiency (more garbage collection)

In Java: primitive types and enumerations are immutable by default

  - Some other cases in Java libraries (see documentation)

How to make objects immutable

- Ensure that all the fields of your class are:
  - Either `private` and not changed by any instance method
  - Or immutable by default
    - Primitive types or enumerations
    - `final`
- Expose internal data consciously
  - Extended interface
  - Return copies
  - Via dedicated design patterns (covered later in the course)

Extended interfaces

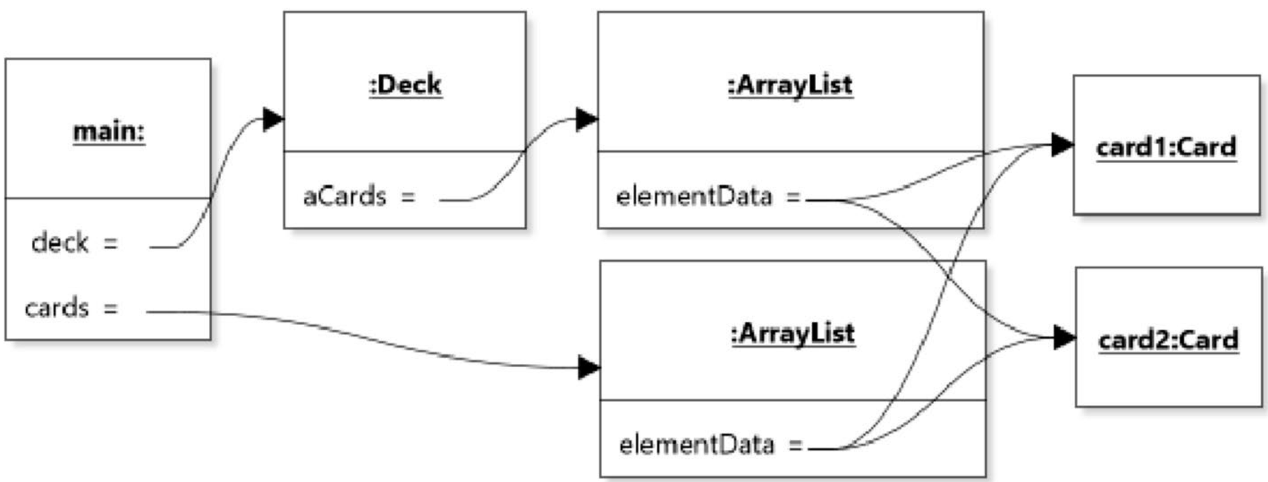
You extend the interface of the class, i.e., its set of `public` methods, with methods returning only references to immutable objects.

```java
public class Deck  {
    private List<Card> aCards = new ArrayList<>();

    public int size() {
        return this.aCards.size();
    }

    public Card getCard(int index) {
        //assuming Card is immutable
        return this.aCards.get(index);
    }

}
```

Returning object copies

You internally **clone** the stored object and return the newly created copy instead of the original.

```java
public class Deck  {
    private List<Card> aCards = new ArrayList<>();

    public Card getCards() {
        //assuming Card is immutable
        return new ArrayList<>(this.aCards);
    }
}
```



Knowing how to copy

In the previous example, we are trusting the implementation of the constructor of `ArrayList`.

- Always check the documentation of the methods you are calling!
- This could have led to the same result:

```java
public List<Card> getCards() {
    return Collections.unmodifiableList(this.aCards);
}
```

PROBLEM: how deep should we copy objects?

ANSWER: until we reach immutable referenced objects

Copy constructors

A popular technique for copying objects is to use a **copy constructor.**

```java
public class Card {
    private Rank rank;
    private Suit suit;

    public Card(Card pCard) {
        this.rank = pCard.rank;
        this.suit = pCard.suit;
    }
}

public class Deck {

  // ...

  public List<Card> getCards() {
      ArrayList<Card> result = new ArrayList<>();
      for(Card card : this.aCards) {
          result.add(new Card (card )
      }
      return result;
  }
}
```

Avoid Complexity / Design for Simplicity

What is complexity?

Practical definition from (Ousterhout, 2018):
«anything related to the structure of a software system that makes it **hard to understand and modify** the system.»

- **Inherent complexity**

Unavoidable domain complexity

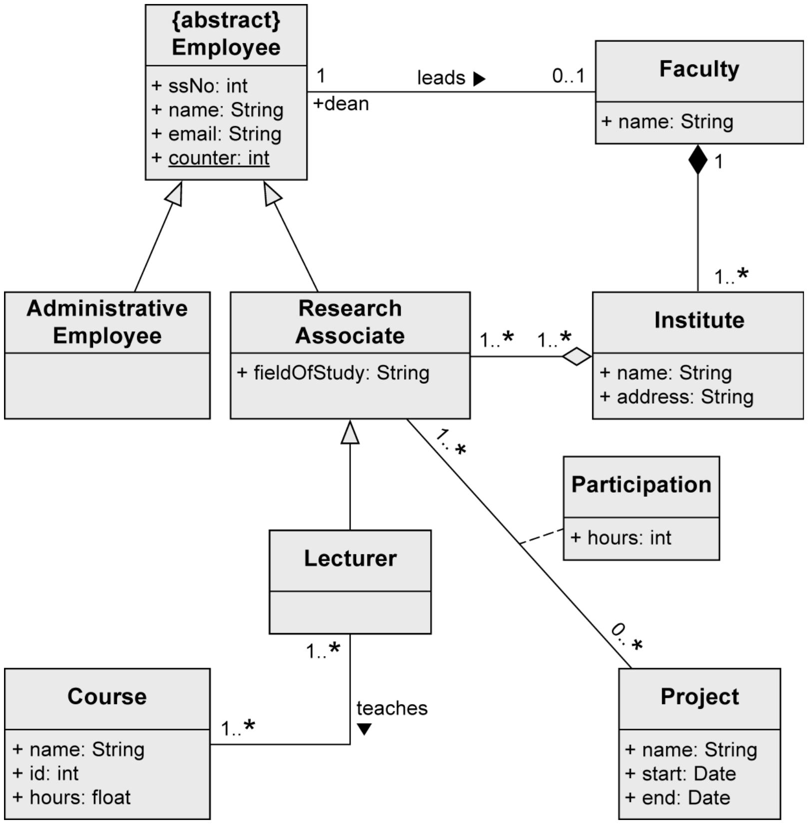
vs.

- **Accidental complexity**

Avoidable technical complexity introduced through suboptimal design

Two general strategies:

- Encapsulate inherent complexity
- Reduce accidental complexity through good, simple design



Types of complexity

- **Structural complexity [= coupling]**
  The number and strength of relationship between the structures in your program (packages, classes, methods)
- **Reading complexity**
  How hard it is to read and understand the program
- **Data complexity**
  The data representations and relationships between the data elements in your program
- **Decision complexity**
  The complexity of the decision flows in your program

Which of these can you influence **the least**?

Guidelines for reducing complexity (1)

- **Structural complexity**
  - Methods should do one thing and one thing only
  - Every class should have a single responsibility

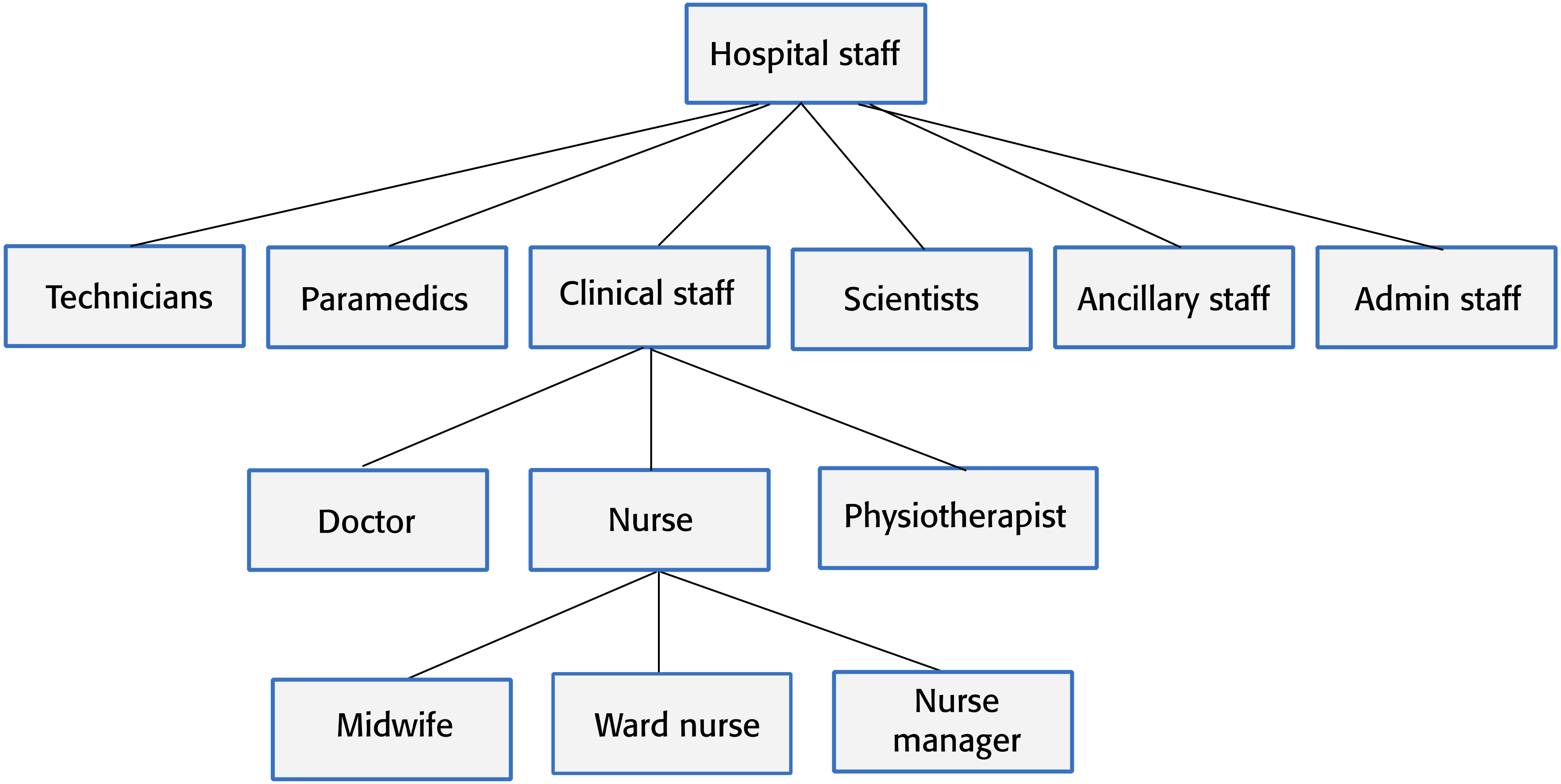
  SRP
  - Methods should not have side-effects
  - Minimize the depth of inheritance hierarchies
  - Avoid multiple inheritance
  - Avoid threads (parallelism) unless absolutely necessary

Guidelines for reducing complexity (2)

- **Data complexity**
  - Define understandable interfaces for important abstractions
  - Define abstract data types if it substantially reduces duplication
  - Avoid using floating-point numbers if possible [1]
- **Decision complexity**
  - Avoid deeply nested conditional statements
  - Avoid complex conditional expressions, e.g., extract parts to functions with clearly understandable names

[1] https://floating-point-gui.de

Minimize the depth of inheritance hierarchies



Why is deep hierarchy a problem?

What happens if we "flatten" this hierarchy?