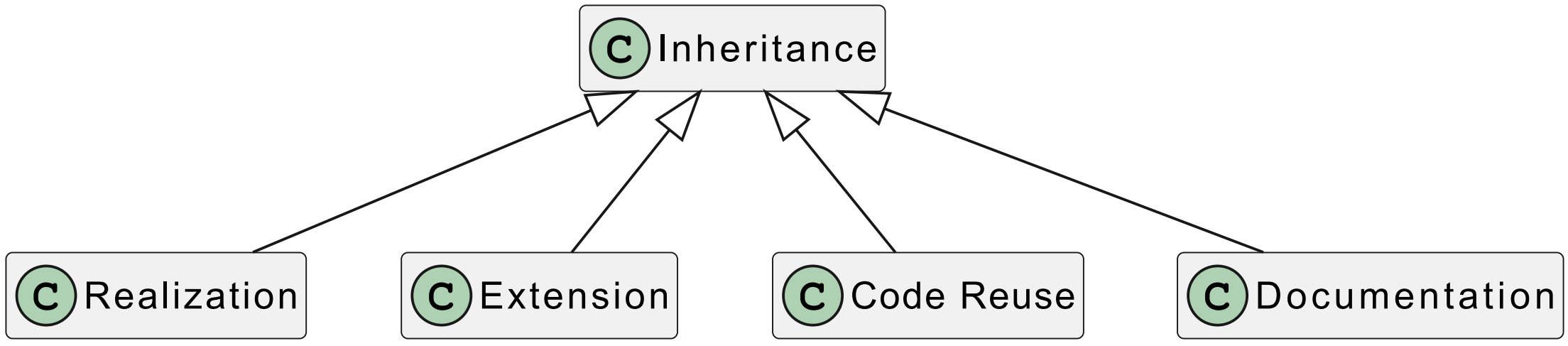
Avoid Inheritance if you can

A discussion of the problems with using inheritance in OO design

(And why you'd better avoid using it)

What is inheritance



What is inheritance

- Realization
  - Declare conformance of an implementation to an interface
  - Abstraction/Polymorphism
  - Class to interface
- Extension
  - Publicly declare covariant compatibility of two types
  - Interface to interface (but also class, since classes are interfaces too)
- Code reuse
  - Inherit methods from the parent class
  - Class to class
- Documentation
  - Establish an organization of types
  - Create a strong conceptual relation between types, like Father and Son

What inheritance is not

- Only one of these things.

Inheritance does not allow you to get one without the others

Why is this problematic?

- Sometimes one of these is not necessary
    - You pay the cost of all of these things even if you just need one
        - It makes your code harder to modify or read
- Sometimes one of these does not apply (in particular compatibility and hierarchy)
    - We are constraint to preserve the compatibility of types even if it is not perfect
        - It makes your code harder to modify or read
    - We are confused by why two unrelated concepts inherit from each other
        - It makes your code harder to modify or read

Example 1: Inheritance for polymorphism

```
interface Drawable {
    void Draw(Surface s);
}
class Line implements Drawable {
    int x1; int x2; int y1; int y2;
    void Draw(Surface s) { ... }
}
class Image implements Drawable {
    void Draw(Surface s) { ... }
}
class Scene {
    List<Drawable> objects;
    void RenderScene(Surface s) {
        s.Clear();
        for (Drawable d : objects) {
            d.Draw(s);
        }
    }
}
```

Example 2: Inheritance as code reuse

```
class Rectangle extends Line {
    // we recycle the x1,x2,y1,y2 fields and their getter/setter to act as
    // top-left/bottom-right. I'm so smart

    // Override this to change behavior (draw a rectangle instead of a line)
    void Draw() { ... }
}
```

Example 3: Inheritance as extension

```
interface VectorDrawable extends Drawable {
    void DrawVector(VectorSurface s);
}

class Shape implements VectorDrawable {
    void Draw(Surface s) { ... }
    void DrawVector(VectorSurface s) { ... }
}
// change Line to also implement VectorDrawable since now
// we support vector graphics as well
class Line implements VectorDrawable {
    ...
    void DrawVector(VectorSurface s) { ... }
}
```

Then chaos ensues

```
## Describe the bug
Rectangles are not being drawn correctly when using a vector surface

## Expected behavior
When drawing to a vector surface rectangles are drawn as a diagonal line
```

Why went wrong?

- `Rectangle` is not a specialization of `Line`, as it cannot be used as a line (Liskov's substitution principle), but reusing code from `Line` to `Rectangle` makes perfect sense.
- Allowing `Line` to be extended makes further specifying its public interface (adding methods or interfaces) an illegal modification violating OCP.
  Simply adding a method can create conflict with child classes by either:
  - being overridden by a method that does not respect the semantics of the parent class method
  - exposing the state of the parent class that was overridden by the subclass
- Although `VectorDrawable` looks like it should intuitively extend `Drawable` there is no advantage in doing that. On the contrary, this means that clients that depend on `VectorDrawable` will also be having dependence on `Drawable` implicitly even if they don't use any of the `Drawable` methods. This is a violation of the ISP

How can we do better?

- Avoid extending classes.
  - Use some other method for code reuse (see later).
- Avoid using classes as types.
  - classes should only depend on interfaces
  - only use the class name when you are calling a constructor or static method
  - the missing methods of a class should be considered part of the contract
- Avoid extending interfaces
  - while this sounds fine in theory, in practice you are creating a bigger interface with no reason
- Use languages that have an explicit interface implementation and require explicit interface casting to use interface method (e.g. F#)

```
type IPrintable =
    abstract member Print: unit -> unit

type SomeClass1(x: int, y: float) =
    interface IPrintable with
        member this.Print() = ...
    member this.Preview () = ...
```

If the interface changes we get a compiler error instead of silently using the declared method

```
type IPrintable =
    abstract member Print: unit -> unit
    abstract member Preview: unit -> unit
```

Example 4: Inheritance as documentation

```
interface Animal { ... }

interface FlyingAnimal extends Animal {
    void fly();
}

class Bird implements Flyer {
    ...
    void fly() {
        ...
    }
}

class Swallow extends Bird {
    ...
}
```

Welcome penguins:

```
class Penguin extends Bird {
    void fly() {
        throw new SorryNotSorryException()
    }
}
```

What went wrong here

We tried to reproduce a real-world classification in our type hierarchy.
This is dangerous because:

- Real-world "is a" is a lot less strict than Liskov's substitution principle
    - it has a lot of exception
    - it can be not transitive
    - sometimes it goes in the opposite direction
- Even if the substitution principle may hold at a certain point (because we are using a simplified model of the business) this may not hold anymore in the future once we refine our model.
    - The hierarchy above would be fine if the "Flyer" interface had only methods for getting wing information

How can we do better

- Avoid using inheritance between classes/interfaces that represent real-world objects.
- Use modules/namespaces/files and folders to organize code instead of class hierarchy

In conclusion

- Implementing interfaces is OK and is necessary for dependency inversion and polymorphism.
    - Use generic programming if available and polymorphism is parametric (e.g. containers)
- Extending interfaces may be ok sometimes to imply compatibility (for example, if you want to express that V2 of an interface is compatible with V1), but beware of growing interfaces too much (in particular, you won't be able to understand if clients of V100 still needs methods from V1). Inheritance can't be broken as it is part of the public interface.
- Do not introduce subtype relations just because something "should-be" something else. Wait for a practical reason. Do not implement interfaces that you do not need yet.
- Avoid inheritance as a code reuse mechanism
    - Prefer composition (see below)
    - Split behavior into multiple small interfaces to allow implementing them into multiple unrelated small classes

Composition over inheritance

Inheriting from a parent class is not required to use its code.
More often than not you can get away with just holding a reference to an instance of that class ("has-a" relation rather than "is-a").

Advantages of composition over inheritance:

- Dependence between classes can be abstracted with an interface, inheritance requires a specific instance. This helps in testing for example as with other reasons DIP exists.
- Favors code splitting and reusing because a class can depend on multiple other classes (but can only inherit one)
- Makes the dependency an implementation detail rather than a part of the public interface (which allows changing/removing it later)
- Allow changing dependencies later in the instance lifetime, while the instance of the base class is fixed at construction time

Composition patterns: delegation

A class `A` implements an interface `B` by delegating all method calls for that interface to a member instance that implements `B`, called the delegate.
Delegation happens by writing stubs of the methods that forward the call to the delegate.
This can be very verbose, but some languages such as Kotlin have a specific syntax for it

Delegation example (Classic Java 1/2)

```
interface ClosedShape {
    int area();
}

class Rectangle implements ClosedShape {
    ...
    public int area() {
      ...
    }
}
```

Delegation example (Classic Java 2/2)

```
class Square implements ClosedShape {
    private Rectangle rectangle;

    public Square(int side) {
        this.rectangle = new Rectangle(side, side);
    }

    public int area() {
        return rectangle.area();
    }
}
```

Delegation example (Kotlin)

```kotlin
interface ClosedShape {
  fun area(): Int
}

class Rectangle(...) : ClosedShape {
  override fun area() = ...
}

class Square(private val side: Int) : ClosedShape by rectangle {
    private val rectangle = Rectangle(side, side)
}
```

Delegation example (Java with `default` 1/2)

```
interface ClosedShape {
    default int area() {
      return delegate.area();
    };
    ClosedShape delegate();
}

class Rectangle implements ClosedShape {
    ...
    public int area() {
      ...
    }
    public ClosedShape delegate() {
      return null;
    }
}
```

Delegation example (Java with `default` 2/2)

```
class Square implements ClosedShape {
    private Rectangle rectangle;

    public Square(int side) {
        this.rectangle = new Rectangle(side, side);
    }

    public ClosedShape delegate() {
        return this.rectangle;
    }
}
```

Composition patterns: Strategy

Similar to the Delegation pattern, but the delegate is polymorphic and is provided externally (and can be switched at runtime).

This is mostly used to configure behavior dynamically, or to switch behavior after some state transition of the object (*State* pattern)

Strategy example 1/2

```
class Person { ... }

class CompareByAge implements Comparator<Person> {
    public int compare(Person p1, Person p2) {
        return Integer.compare(p1.getAge(), p2.getAge());
    }
}

class CompareByName implements Comparator<Person> {
    public int compare(Person p1, Person p2) {
        return p1.getName().compareTo(p2.getName());
    }
}
```

Strategy example 2/2

```
class PersonSorter {
    private Comparator<Person> strategy;

    public PersonSorter(Comparator<Person> strategy) {
        this.strategy = strategy;
    }

    public void setStrategy(Comparator<Person> strategy) {
        this.strategy = strategy;
    }

    public void sort(Person[] people) {
        Arrays.sort(people, strategy);
    }
}
```

Questions?

SOLID (the easy part)

Single Responsibility Principle (SRP)

- A class should have only one reason to change.
  - Divide functionality that can change for different reasons
- A modification should affect only one module.
  - Group together functionality that changes together

SOLID (the easy part)

Interface Segregation Principle (ISP)

- Interfaces should be minimal and defined on the requirements of the dependent
  - Interfaces are tightly coupled with the class that depends on them and change with it

Dependency Inversion Principle (DIP)

- Classes only know each other through the interfaces they implement
  - Interface is a "contract" between implementer and user that guarantees freedom of action to both

SOLID (the hard part)

Open-Closed Principle (OCP)

- Open: I can add functionality to a module/interface
- Closed: provided that I guarantee the constraints of the original module for users

Liskov Substitution Principle (LSP)

- Subtype relationship is a subset relationship
- The subtype must guarantee all the properties of the type from which it derives: if A extends B then $\forall P : P(B) \Rightarrow P(A)$