Ingegneria del Software

Corso di Laurea in Informatica per in Management

Design Patterns
part 1

Davide Rossi

Dipartimento di Informatica – Scienze e Ingegneria

Università di Bologna

Pattern

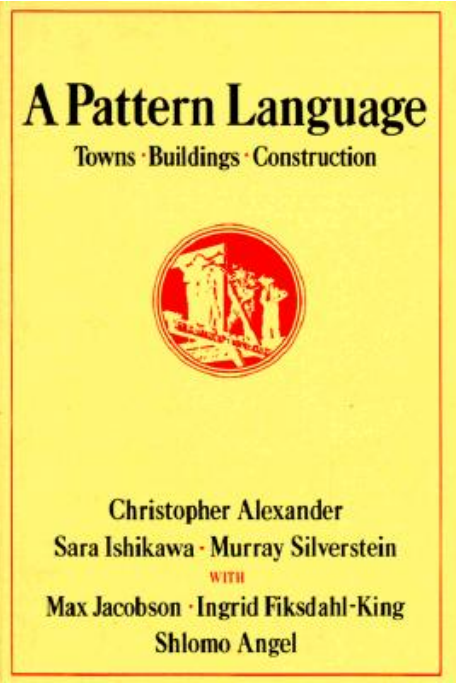
Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice [Alexander 77]

Design Pattern

- Patterns provide “workable solutions to all of the problems known to arise in the course of design”. [Beck, Cunningham - 87]
- A named description of a problem, solution, when to apply the solution, and how to apply the solution in new contexts. [Larman]
- A general reusable solution to a commonly occurring problem within a given context in software design [wikipedia]

Languages and catalogs

Patterns are usually grouped into a coherent structure with its own vocabulary, syntax (context) and grammar (use). Patterns are linked so the adoption of one suggests other to consider.



Patterns catalogs

The are several existing catalogs for patterns in software engineering. We will focus on design patterns. There are several existing catalogs for design patterns too.

We will present design patterns from *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (the *Gang of Four*) - 1994.

Documentation

- **Pattern Name and Classification**: a descriptive and unique name that helps in identifying and referring to the pattern.
- **Intent**: a description of the goal behind the pattern and the reason for using it.
- **Also Known As**: other names for the pattern.
- **Motivation (Forces)**: a scenario consisting of a problem and a context in which this pattern can be used.

Documentation

- **Applicability**: situations in which this pattern is usable; the context for the pattern.
- **Structure**: a graphical representation of the pattern. Class diagrams and Interaction diagrams may be used for this purpose.
- **Participants**: a listing of the classes and objects used in the pattern and their roles in the design.
- **Collaboration**: a description of how classes and objects used in the pattern interact with each other.

Documentation

- **Consequences**: a description of the results, side effects, and trade offs caused by using the pattern.
- **Implementation**: a description of an implementation of the pattern; the solution part of the pattern.
- **Sample Code**: an illustration of how the pattern can be used in a programming language.
- **Known Uses**: examples of real usages of the pattern.
- **Related Patterns**: other patterns that have some relationship with the pattern; discussion of the differences between the pattern and similar patterns.

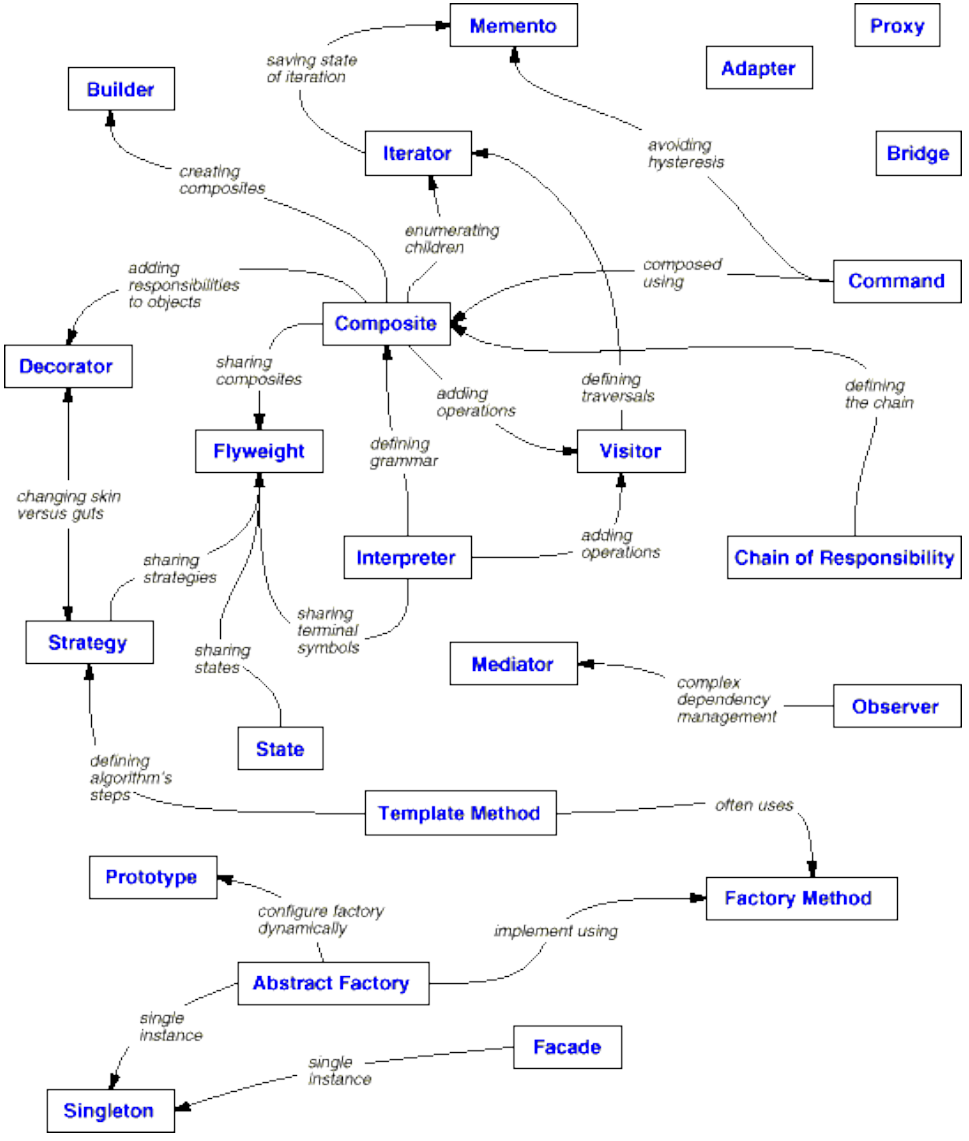
GoF patterns purposes

- Creational
  - abstract the instantiation process
- Structural
  - are concerned with how classes and objects are composed to form larger structures
- Behavioral
  - are concerned with algorithms and the assignment of responsibilities between objects

The 23

- **Creational**: Abstract Factory, Builder, Factory Method, Prototype, Singleton
- **Structural**: Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy
- **Behavioral**: Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template method, Visitor

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyw eight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor


Memento
Proxy
saving state of iteration
Builder
Adapter
Iterator
avoiding hysteresis
Bridge
creating composites
enumerating children
adding responsibilities to objects
composed using
Command
Composite
Decorator
sharing composites
adding operations
defining traversals
defining the chain
Flyweight
defining grammar
Visitor
changing skin versus guts
sharing strategies
Interpreter
adding operations
Chain of Responsibility
Strategy
sharing terminal symbols
sharing states
Mediator
complex dependency management
Observer
State
defining algorithm's steps
Template Method
often uses
Prototype
Factory Method
configure factory dynamically
implement using
Abstract Factory
single instance
Facade
single instance
Singleton

Composition over inheritance

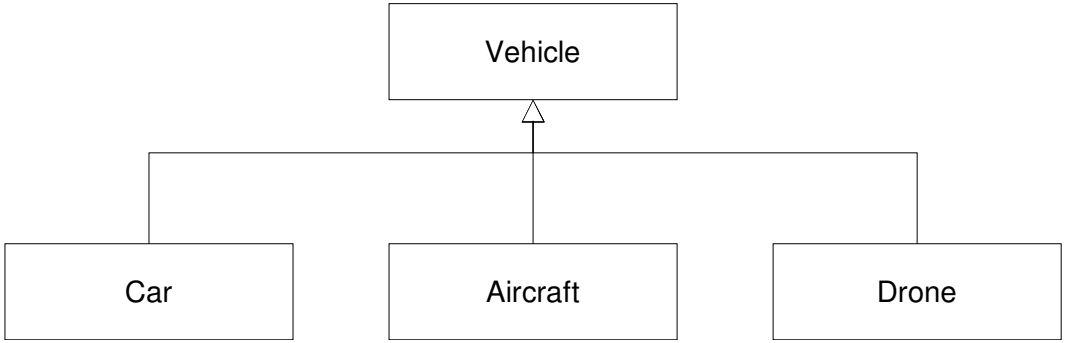
- The two most common techniques for reusing functionality in object-oriented systems are class inheritance and object composition
- White box vs black box
- Static vs dynamic
- **Favoring object composition over class inheritance** helps you keep each class encapsulated and focused on one task
- Delegation is a way of making composition as powerful for reuse as inheritance (*having* instead of *being*)

What’s wrong with inheritance?

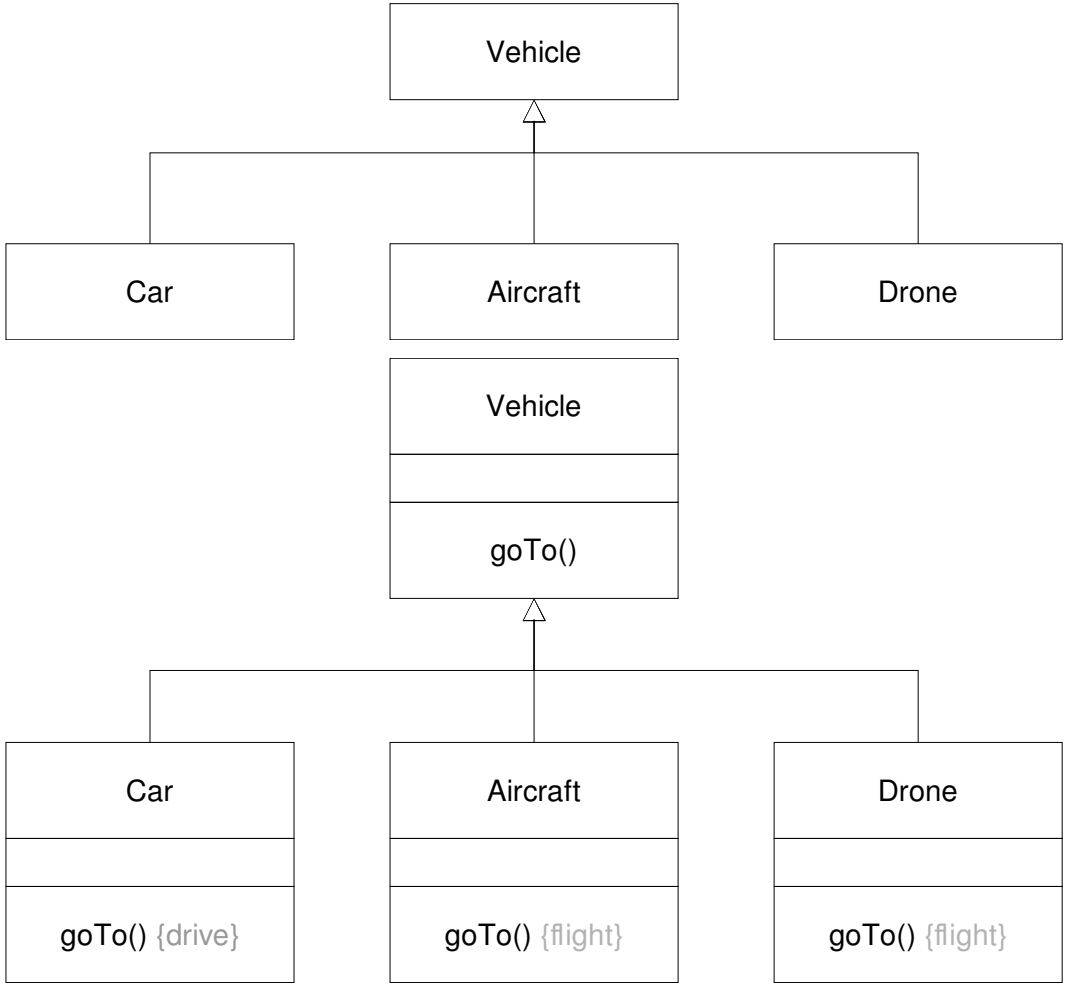
Inheritance addresses two unrelated problems at once:

- Polymorphism, via subtyping (as an approximation of substitutability, see LSP)
- Behavior sharing (well, state as well, but that’s not particularly relevant here)

What’s wrong with inheritance?



What’s wrong with inheritance?

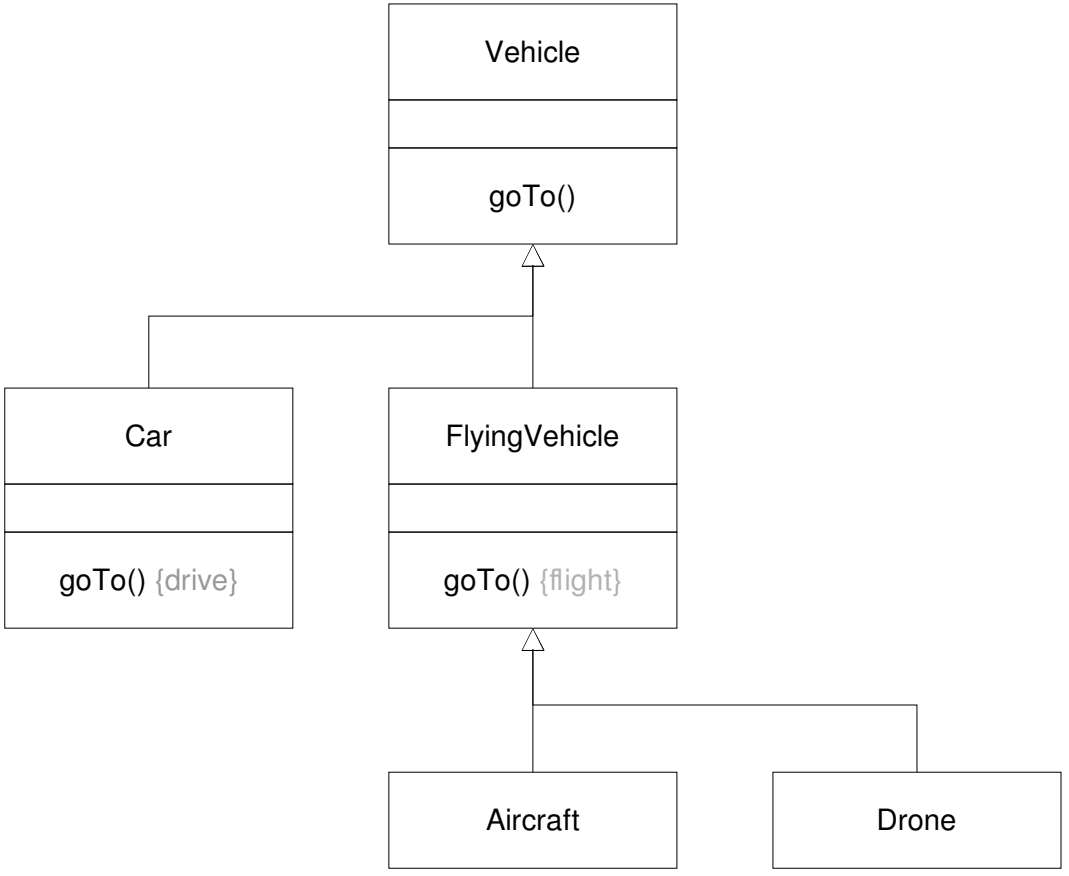


What’s wrong with inheritance?

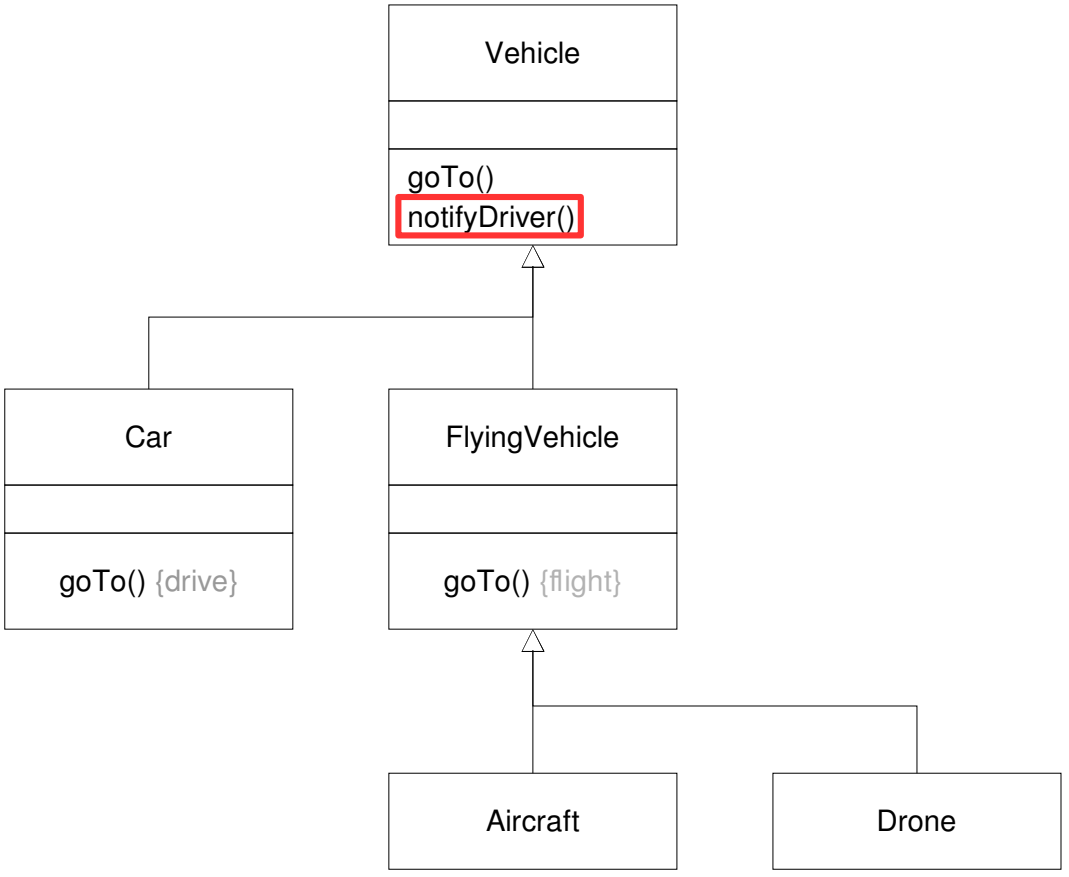
- `goTo` in `Aircraft` and in `Drone` behave in the same manner
- Needless Repetition (from design smells):

  Copy and paste may be useful text-editing operations, but they can be disastrous code-editing operations.
  When the same code appears over and over again, in slightly different forms, the developers are missing an abstraction.

What’s wrong with inheritance?



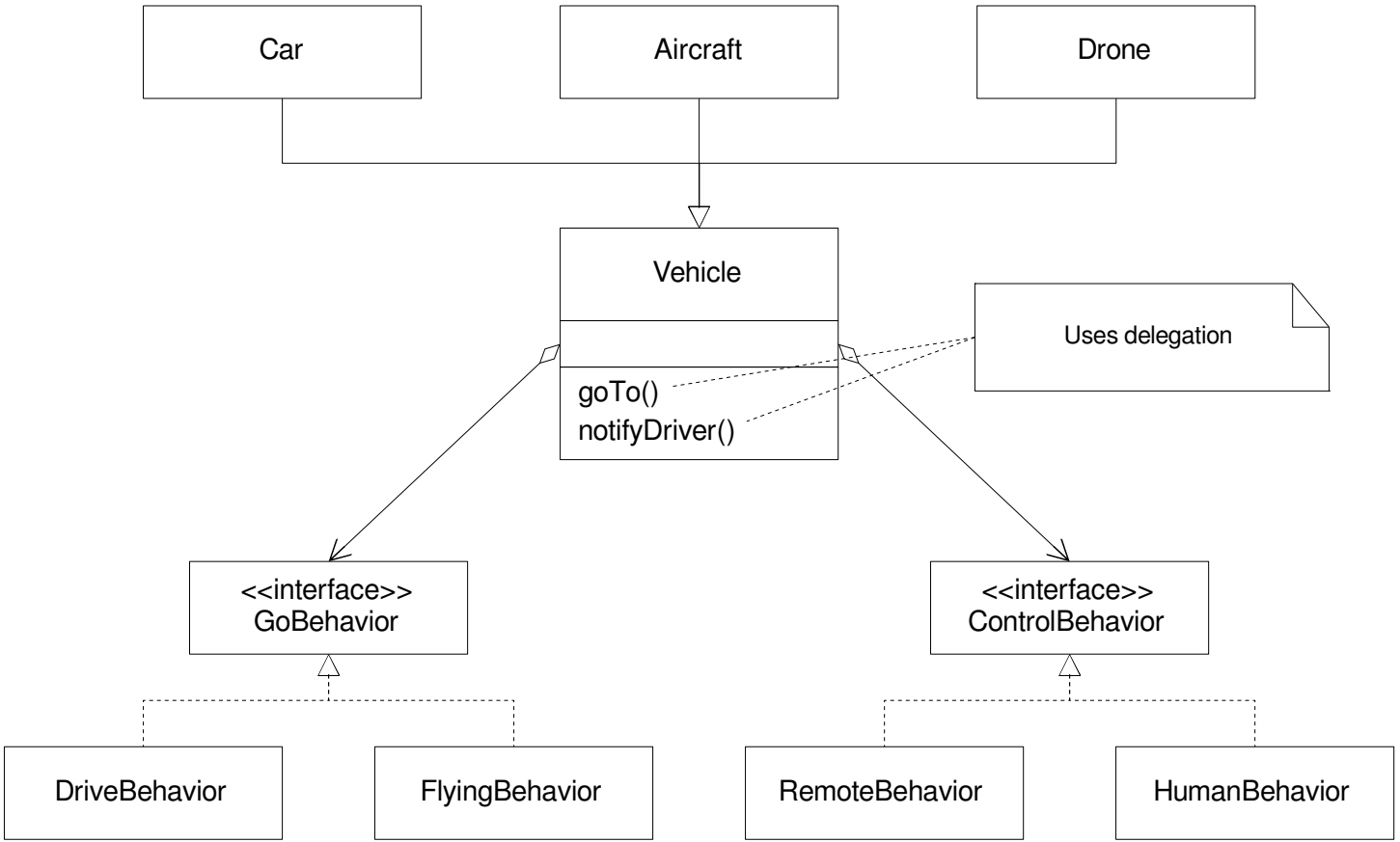
What’s wrong with inheritance?



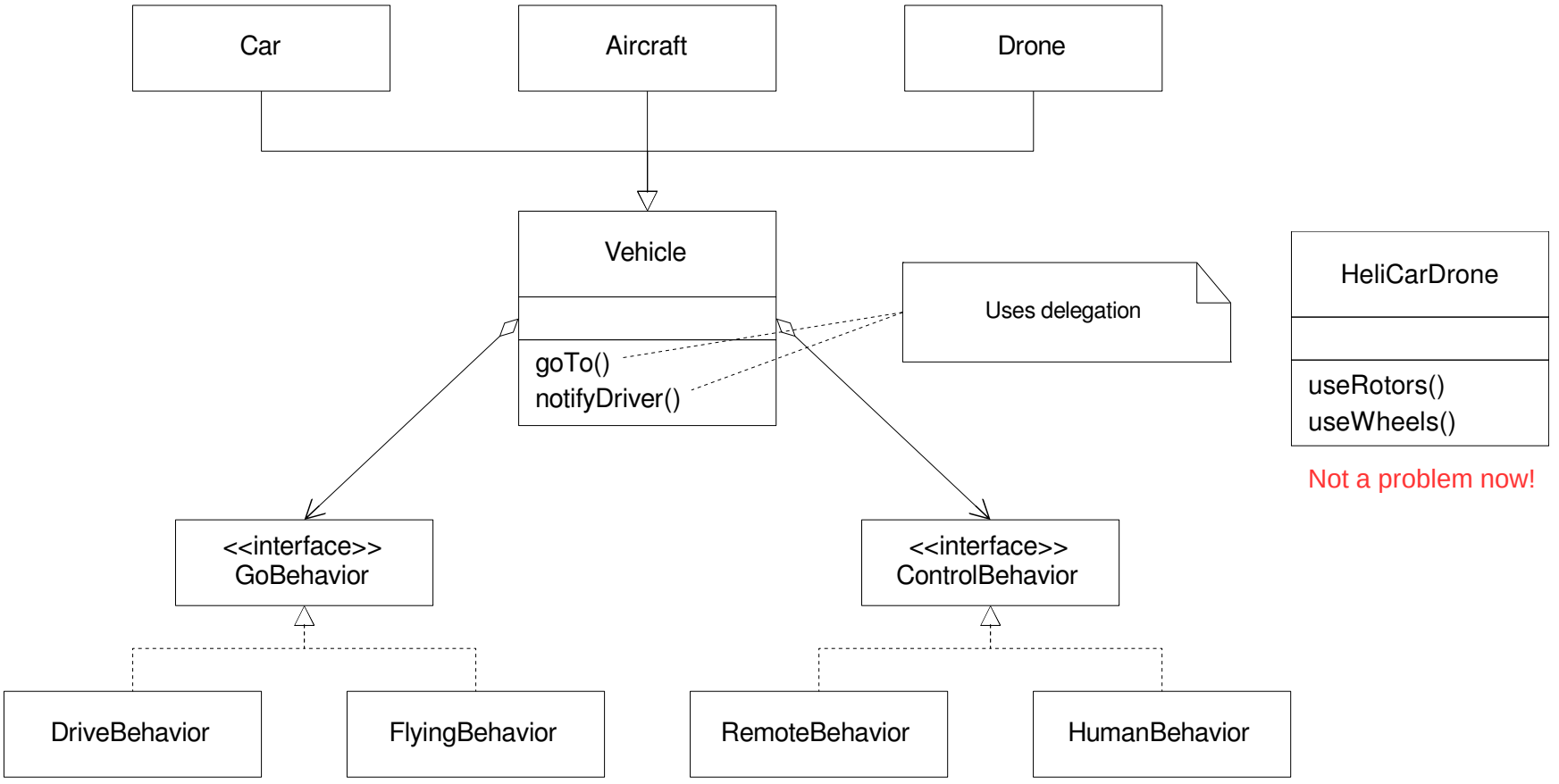
What’s wrong with inheritance?

- What’s wrong with our use of inheritance?
  - We are not addressing OCP correctly
  - We are not addressing PV correctly
- Solution
  - Superclasses are not the only possible abstractions
  - Use composition: more flexible and can change at run time (favor composition over inheritance)

What’s wrong with inheritance?



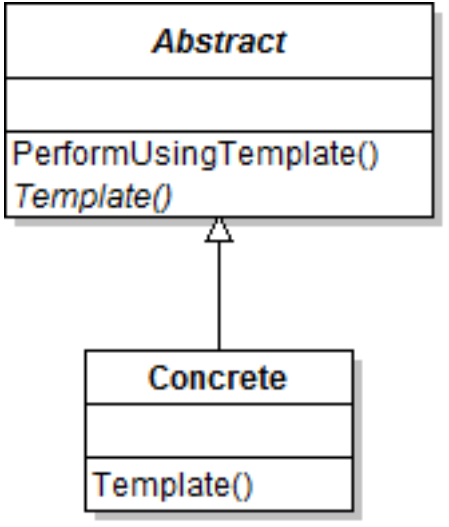
What’s wrong with inheritance?



GoF: Template Method

- Problem: how can I share partially defined behavior in a inheritance hierarchy?
- Template Method: define the skeleton of an algorithm in an operation, deferring some steps to subclasses.
- Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

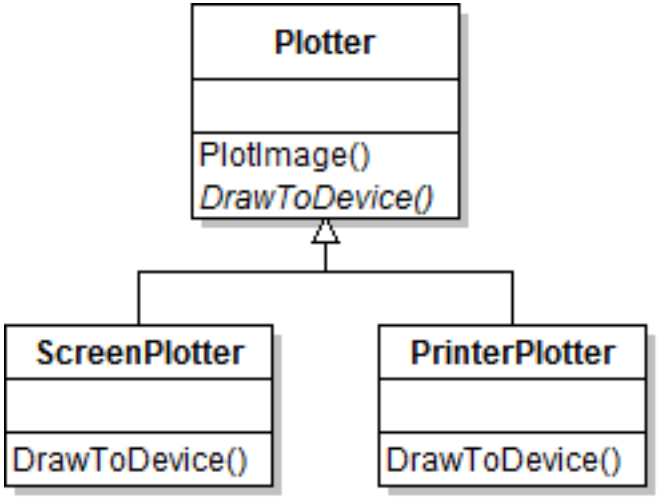
GoF: Template Method



GoF: Template Method

- Example: `Canvas/ScreenCanvas/PrinterCanvas`. `DrawSpiral` method: same algorithm but one uses `showPoint`, the other `printPoint`.
- In class hierarchies I want to “push” common behavior as high as possible. Yet sometimes, I have behaviors that are not exactly the same because some details change in sub-classes.

Template Method



Template Method in Java standard library

```
java.util.AbstractList<E> {
    int indexOf(Object o)
    abstract E get(int index)
    …
}
```

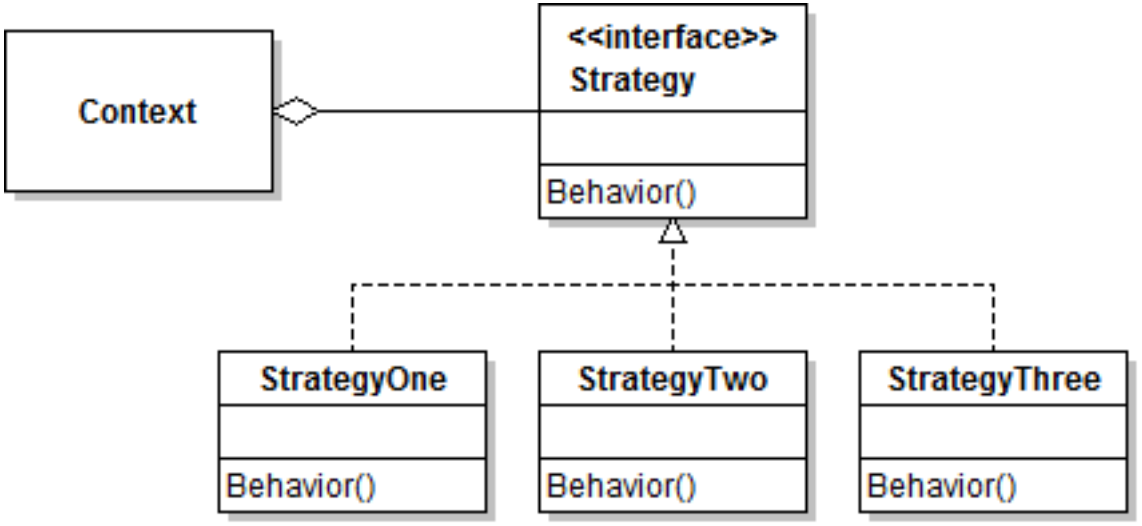
Template Method

- Moves common behaviors up in the inheritance tree even when partially specified
- New implementations of the abstract superclass do not break clients: dependencies are directed to more stable elements
- Helps adherence to OCP

GoF: Strategy

- Problem: how can I separate an object from (part of) its behavior and change it at run time?
- Strategy: define a family of algorithms, encapsulate each one, and make them interchangeable
- Strategy lets the algorithm vary independently from clients that use it

GoF: Strategy



Strategy

- Helps implementing OCP
- Obeys Protected Variations
- Favors composition (that is dynamic) over inheritance (that is static): “has a” over “is a”

Resources

Books

- Eric Freeman & Elisabeth Robson, Head First Design Patterns: Building Extensible and Maintainable Object-Oriented Software (2nd Edition), O'Reilly

Online:

- http://www.vincehuston.org/dp/
- http://www.oodesign.com/
- https://refactoring.guru/design-patterns/
- http://www.informit.com/articles/article.aspx?p=1404056