Projektowanie złożonych systemów telekomunikacyjnych

Lecture 3: Design Patterns

Łukasz Marchewka

14-03-2022

Agenda

- OOP Principles
    - SOLID

- Design Patterns
    - Definition
    - Classification
    - Examples

OOP Principles

- Good-quality OOP code:
  - should be reusable
  - should be easy to extend (by adding a new functionality)
  - should be easy to service
  - should be easy to test

- S.O.L.I.D.
  - **S**ingle Responsibility Principle
  - **O**pened-Closed Principle
  - **L**iskov Substitution Principle
  - **I**nterface Segregation Principle
  - **D**ependency Inversion Priniciple

OOP Principles

Single Responsibility Principle:
Each class should have only one reason to change!

A responsibility of a module or class should be a single part of the functionality provided by the software. That responsibility should be entirely encapsulated by the class.

OOP Principles

Opened-Closed Principle
Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification

OOP Principles

Liskov Substitution Principle

If **S** is a subtype of **T**, then objects of type **T** in a program may be replaced with objects of type S without altering any of the desirable properties of that program (e.g. correctness)

- Preconditions cannot be strengthened in a subtype
- Postconditions cannot be weakened in a subtype
- Invariants of the supertype must be preserved in a subtype

OOP Principles

Interface Segregation Principle

- No client should be forced to depend on methods it does not use

OOP Principles

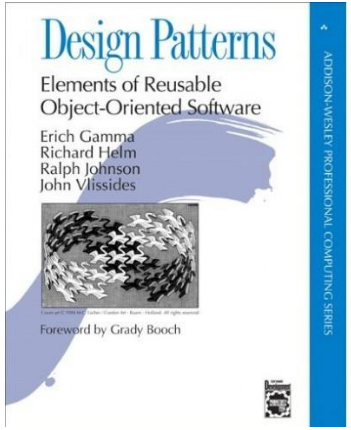
Dependency Inversion Principle

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

Design Patterns

- Common solutions for common problems within a given context in software design
- Their source is in architecture (Christopher Alexander)
- First introduced in programming by *Kent Beck* and *Ward Cunningham* (1987)
- The most important source – Gang of Four (*E. Gamma*, *R. Helm*, *R. Johnson*, *J. Vlissides*), *Design Patterns* (1995)
- In alignment with good programming practices

"Descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context."
„Gang of Four" („Design Patterns")


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Design Patterns - Definition

- The pattern provides an abstract description of a design problem and how a general arrangement of elements (classes and objects) solves it.

Elements of a pattern:

- **Name** can be used to describe a design problem, its solutions, and consequences. It lets us design at a higher level of abstraction.
- **Context/Problem** indicates when to apply the pattern.
- **Solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations. The solution doesn't describe a particular concrete design or implementation.
- **Consequences** are pros and cons of applying the pattern.

Design Patterns in Embedded Systems

Criteria for embedded software:

- Performance,
- Deterministic,
- Low number of reusable objects in the system. A lot of very specific classes,
- Low usage of external libraries.

- This excludes some of the design patterns like Observer, Facade, Chain of responsibility, Mediator
- Of course, these still can be used but their use is limited.

Design Patterns

- Classification
    - Creational
    - Structural
    - Behavioral

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

Creational Design Patterns

- Provide flexible mechanisms for object creation and encourage the reuse of existing code
- Abstract the instantiation process
- Help to make a system independent of how the objects are created

Examples:

- **Factory Method**
- **Abstract Factory**
- Builder
- Prototype
- Singleton

Structural Design Patterns

- Provide efficient solutions for assembling different classes and objects into larger structures based on composition.

Examples:

- **Adapter**
- Bridge
- Composite
- **Decorator**
- Facade
- Proxy
- Flyweight

Behavioral Design Patterns

- Describe how objects and classes communicate/interact with each other and divide the responsibility.

Examples:

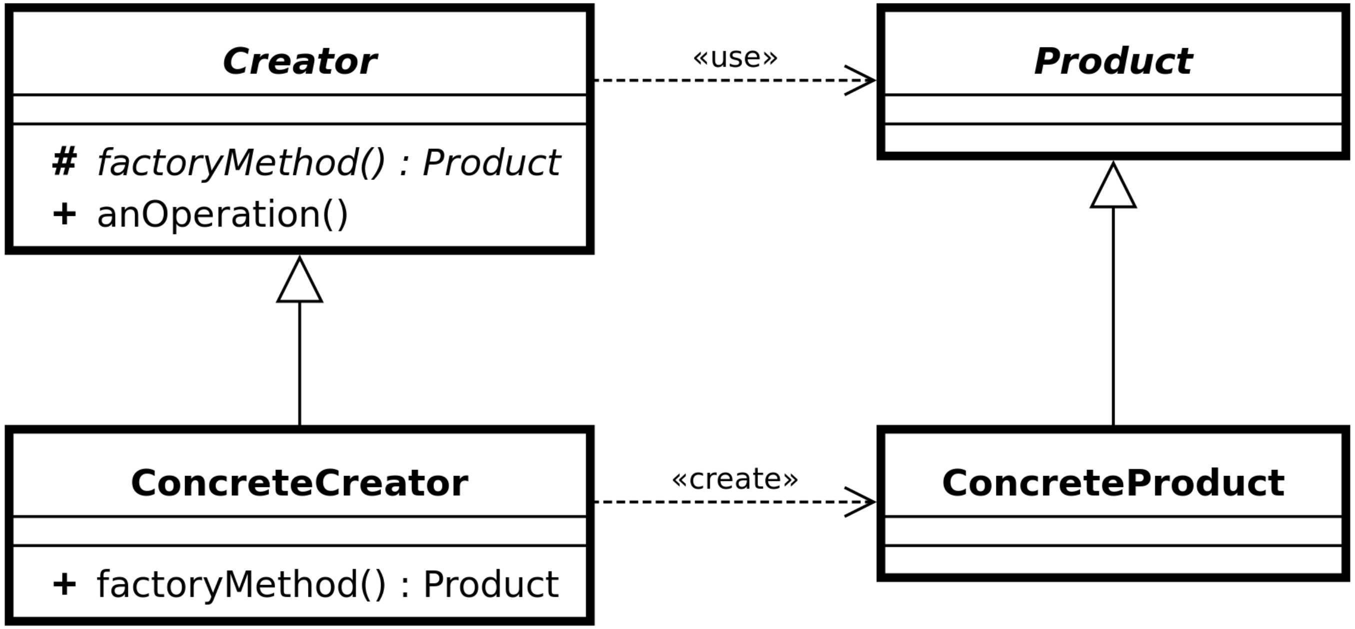
- Chain of Responsibility
- Command
- Interpreter (class type)
- Iterator
- Mediator
- Memento
- Observer
- **State**
- **Strategy**
- Template Method (class type)
- Visitor

Factory Method / Abstract Factory

Factory Method/Abstract Factory

- Creational patterns
- Encapsulate business logic related to object creation and hide this process (support dependency injection)
- Decouples code for object hierarchy construction from runtime logic code, thus helps enforce Single Responsibility Rule,
- The Factory Method separates product construction code from the code that uses the product,
- Allows avoiding explicit usage of *new* and sometimes *if* or *#ifdef*
- Factory Method creates one object, Abstract Factory – a family of objects

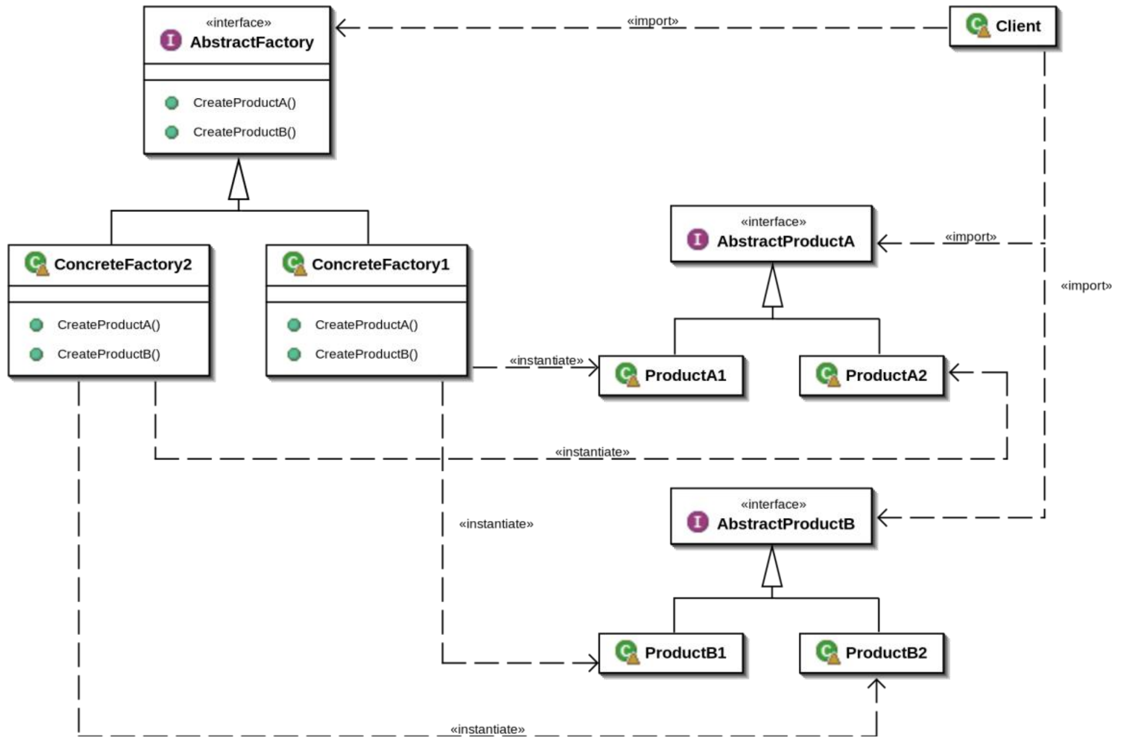
Factory Method



https://sourcemaking.com/

Abstract Factory = set of Factory Methods

By Giacomo Ritucci, https://commons.wikimedia.org/w/index.php?curid=741978



Factory Method/Abstract Factory – Pros and Cons

+ Tight coupling between creator and the concrete products is avoided,

+ Follow Single Responsibility Principle -> creation of a product is extracted from other logic,

+ Follow Open/Closed Principle -> new products can be introduced without breaking the existing client code.

- The code may become complicated

Factory Method/Abstract Factory – Applicability

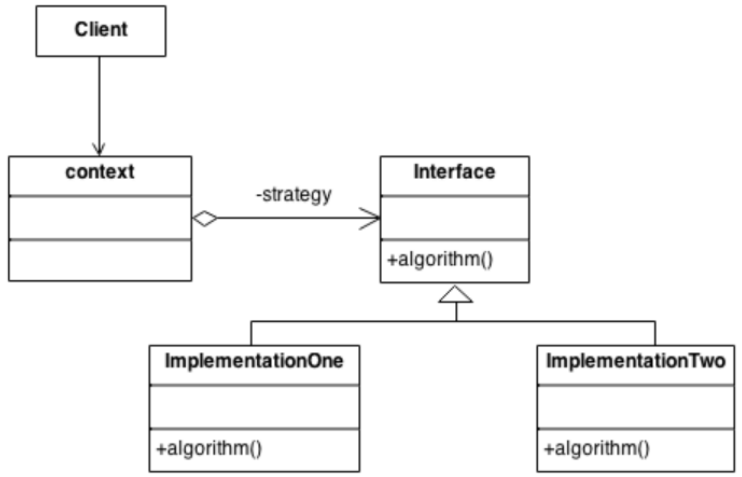
- To provide users of your library or framework with a way to extend its internal components,
- When exact types and dependencies of the objects are unknown beforehand,
- To save resources by code reuse,
- When introducing TestSuites to replace real dependencies with mocks.

Strategy

Strategy pattern

- Behavioral design pattern,
- Allows selecting an algorithm or specific behavior **at runtime** (based on composition, not inheritance),
- Provides good decoupling and independent testing of algorithms,
- Allows change of the class behavior without the need of changing the class,
- Allows code reuse and avoid code duplication,
- Class needs to hold strategies as a reference, pointers or smart pointers,
- Holding strategy by reference is permanent for the lifetime of the object, strategies hold by pointers can be changed during the object lifetime,
- Object can have multiple strategy hierarchies

Strategy pattern



https://sourcemaking.com/

Strategy pattern – Pros and Cons

+ Algorithms can be swapped at runtime,

+ The implementation details isolated an algorithm from the code that uses it,

+ Composition over inheritance,

+ Open/Closed Principle. New strategies can be introduced without changing the context.

- For only a couple of algorithms that are changed rarely the code may become overcomplicated,

- Clients must be aware of the differences between strategies to be able to select a proper one,

- Modern programming languages have the support of anonymous functions that allow implementation of different versions of an algorithm inside a set.

Strategy pattern – Applicability

- To use different variants of an algorithm within an object and be able to switch from one algorithm to another during runtime,
- To avoid extensive conditional operators to choose different variants of the same algorithm,
- To sort out a lot of similar classes that only differ in the way they execute some behavior.

State

State pattern

- Behavioral design pattern,

- Class diagram looks the same as for Strategy – the difference is in the intention,

Strategy makes objects completely independent and unaware of each other.
State doesn't restrict dependencies between concrete states, letting them alter the state of the context at will.

- Provides good decoupling and independent testing of algorithms,

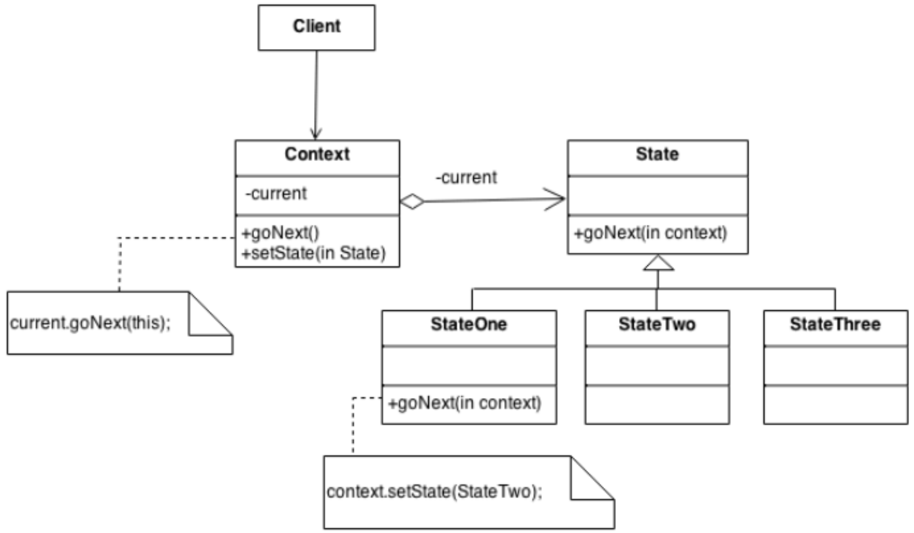
- Combines state of a class with code to be run in a given state,

- Uses polymorphism instead of conditional statements to pick appropriate actions in the given state,

- Improves readability of transitions between states,

- Class can have multiple state hierarchies.

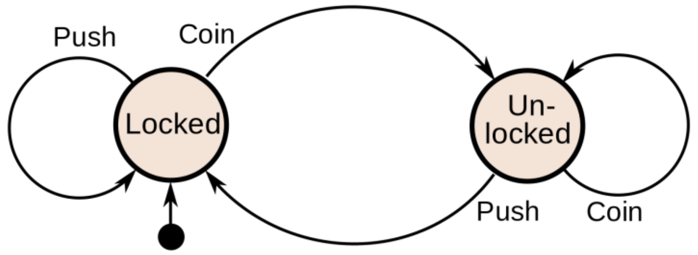
State pattern



https://sourcemaking.com/

Finite State Machine

- Mathematical model of computation, widely used in mathematics and computer science,
- Usually depicted in a form of State Diagram,
- Represents by a set of States, and a set of Transitions, and an entry point,
- There is always one Active State,
- Each Transition has an associated event or condition which triggers the transition,
- May be extended by Entry Actions or Exit Actions



https://en.wikipedia.org/wiki/Finite-state_machine

State pattern - Applicability

- Use the State pattern when you have an object that behaves differently depending on its current state, the number of states is enormous, and the state-specific code changes frequently

- To remove massive conditionals that alter the class behavior according to the current values of the class’s fields.

State pattern – Pros and cons

+ Single Responsibility Principle. Organize the code related to particular states into separate classes,

+ Open/Closed Principle. Introduce new states without changing existing state classes or the context,

+ Code simplification of the context by huge state machine conditionals.

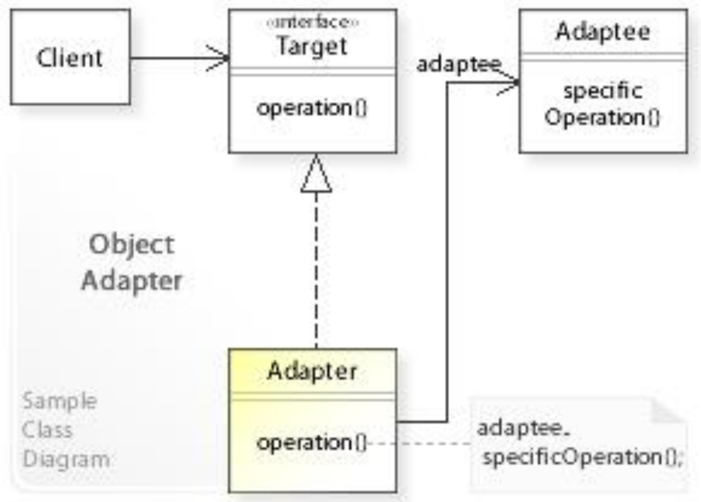
- For only a few states it can complicate the code

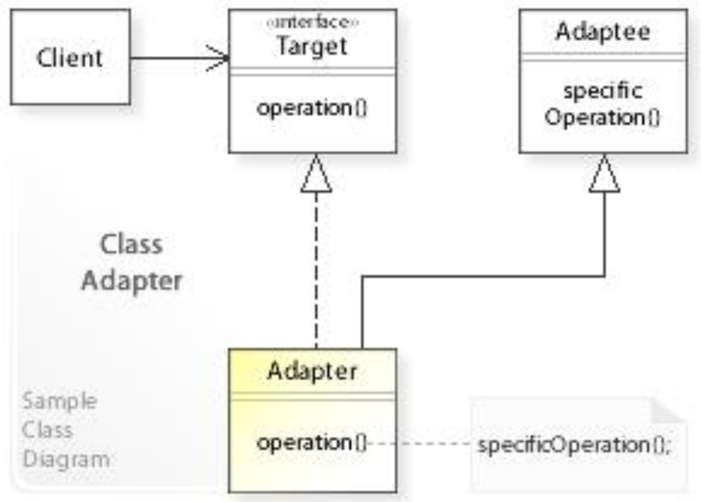
Adapter

Adapter

- Structural design pattern,
- Allows two classes with incompatible interfaces to work together,
- Should not contain logic,
- In alignment with the “open-close” principle,
- Two possible ways of implementation – inheritance, and composition

Adapter





https://en.wikipedia.org/wiki/Adapter_pattern

Adapter - Applicability

- To use existing class with incompatibile interface,
- To reuse existing subclasses that lack some common functionality that can’t be added to the superclass.

Adapter – Pros and cons

+ Single Responsibility Principle. Separate the interface or data conversion from the primary business logic,

+ Open/Closed Principle. Introduce new adapters without changing existing client code,

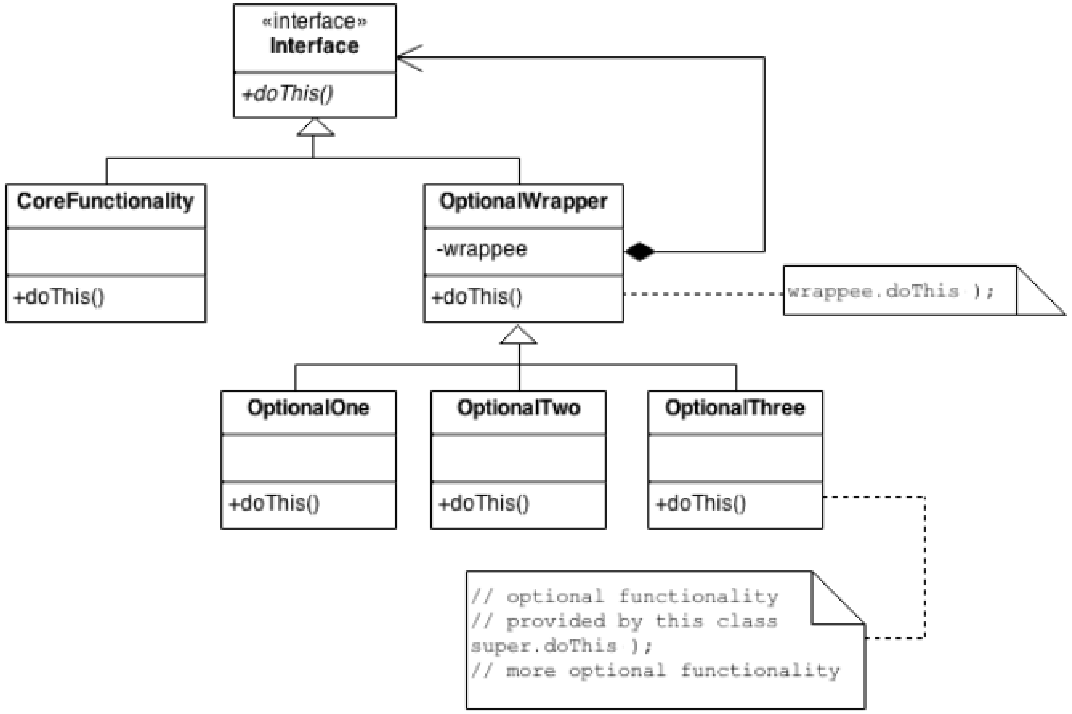
- Overcomplication, sometimes it’s better to change the service class

Decorator

Decorator

- Structural pattern,
- Allows to dynamically add behaviors to the class,
- More flexible alternative to inheritance,
- Possibility of adding many behaviors or combinations of them,
- Each Decorator object may be treated in the same way as the object being decorated (they have the same interface),
- Decorating an object does not change the object itself (*open-close principle*),
- Does not depend on creating subclasses (avoided “class explosion”)

Decorator



```
// optional functionality
// provided by this class
super.doThis );
// more optional functionality
```

https://sourcemaking.com/

Decorator - Applicability

- To assign new functionalities at runtime without breaking the client code,
- To avoid unnecessary inheritance or when inheritance is not allowed (final keyword)

Decorator – Pros and cons

+ Single Responsibility Principle. Divide a class that delivers many functionalities into several smaller classes,

+ Open/Closed Principle. Add/remove responsibilities at runtime,

+ Combine several behaviors by wrapping an object into multiple decorators.

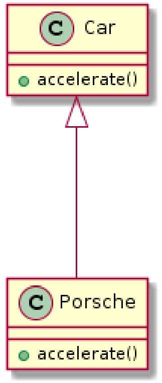
- Many similar objects therefore high complexity,

- Difficult debugging

Exercise: Car Factory & Strategy

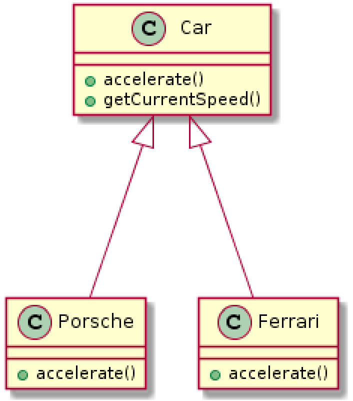
Exercise: Car Factory & Strategy

- Porsche:
  - Has 200 HP
  - Has Cx of 0.66
  - Accelerates each time by HP/2
  - Has top speed limited by HP/Cx
- Test top speed on the test track that accepts Car.



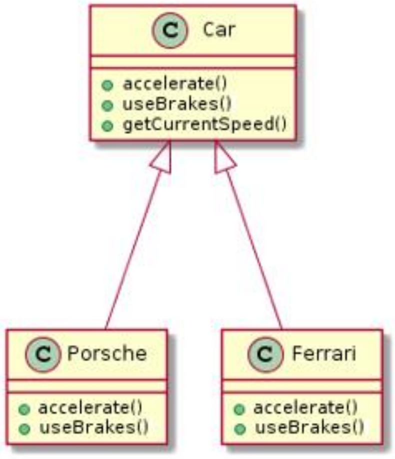
Exercise: Car Factory & Strategy

- Ferrari:
    - Has 300HP
    - Accelerates according to sequence HP/2, HP/4, HP/8 …
    - Has unlimited top speed
- Test it on the test track together with Porsche.



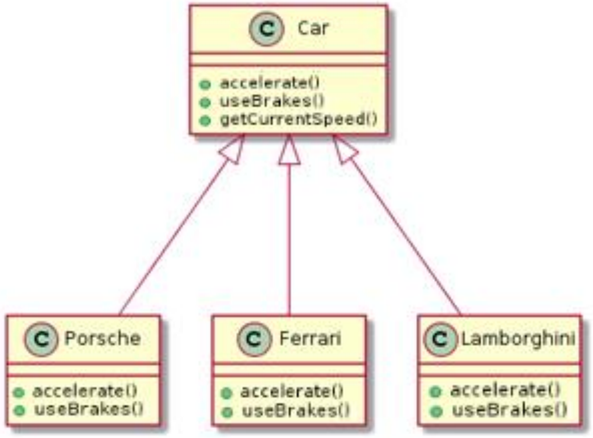
Exercise: Car Factory & Strategy

- Add breakes:
  - Ferrari stops on a dime (useBrakes() sets currentSpeed to 0)
  - Porsche useBrakes()
    - If currentSpeed < 50 sets currentSpeed to 0
    - Otherwise sets currentSpeed to currentSpeed/2
- Test brakes of both cars on the test track.



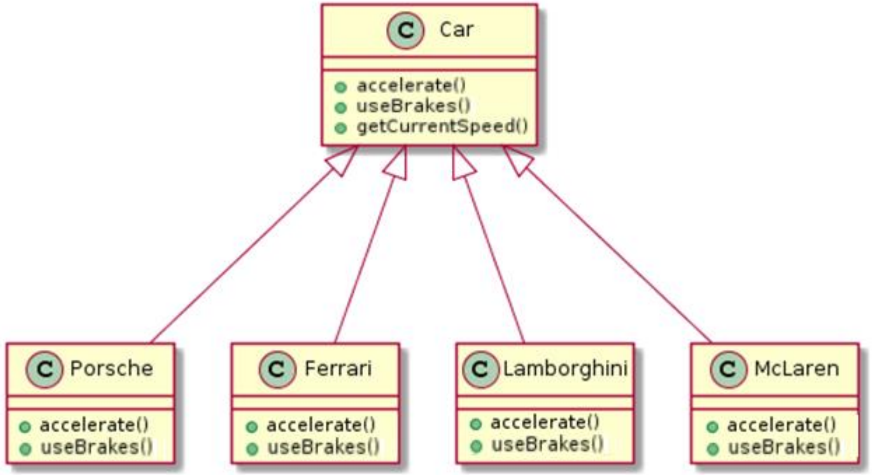
Exercise: Car Factory & Strategy

- Lamborghini:
  - Has 400HP
  - Accelerates like Porsche
  - Brakes like Ferrari
  - Has top speed limited by 315
- Test it on the test track together with other cars.



Exercise: Car Factory & Strategy

- McLaren:
  - Has 366HP
  - Cx 0.5
  - Accelerates like Ferrari
  - Brakes like Porsche
  - Has speed limit like Porsche
- Test it on the test track together with other cars.



NOKIA

Copyright and confidentiality

Revision history and metadata

Please delete this slide if document is uncontrolled

Document ID: DXXXXXXXXX
Document Location:
Organization:

Version	Description of changes	Date	Author	Owner	Status	Reviewed by	Reviewed date	Approver	Approval date
		DD-MM-YYYY					DD-MM-YYYY		DD-MM-YYYY