Projektowanie złożonych systemów telekomunikacyjnych

Lecture 3: Design Patterns

Łukasz Marchewka

14-03-2022

What are Design Patterns?

        <Document ID: change ID in footer or remove> <Change information classification in footer>

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

- Classification
    - Creational
    - Structural
    - Behavioral
    - Concurrency

       <Document ID: change ID in footer or remove> <Change information classification in footer>

Design Patterns in Embedded Systems

Criteria for embedded software:

- Performance,
- Deterministic,
- Low amount 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.

 <Document ID: change ID in footer or remove> <Change information classification in footer>

Creational Design Patterns

- Provide flexible mechanisms for object creation and encourage the reuse of existing code e

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

    <Document ID: change ID in footer or remove> <Change information classification in footer>

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

　　　　<Document ID: change ID in footer or remove> <Change information classification in footer>

Behavioral Design Patterns

- Describe how objects and classes communicate/interact with each other and divide the responsibility.
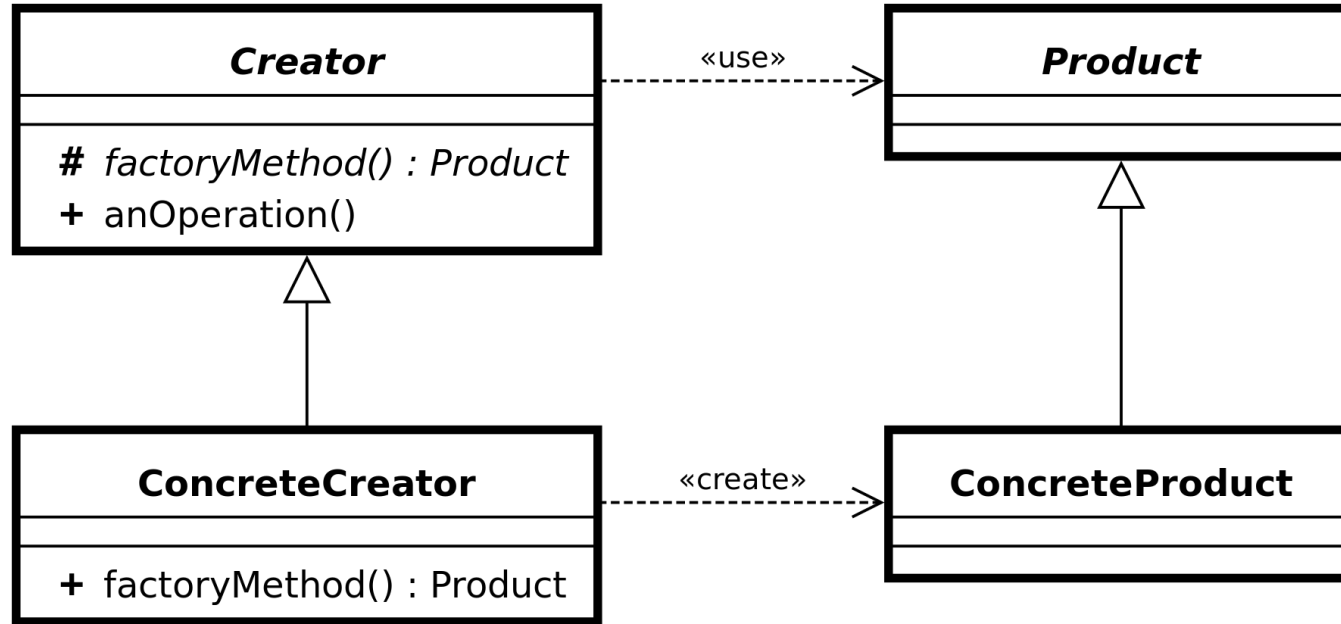
  Examples:
  - Chain of Responsibility
  - Command
  - Interpreter (class type)
  - Iterator
  - Mediator
  - Memento
  - Observer
  - **State**
  - **Strategy**
  - Template Method (class type)
  - Visitor

<Document ID: change ID in footer or remove> <Change information classification in footer>

Factory Method / Abstract Factory

     <Document ID: change ID in footer or remove> <Change information classification in footer>
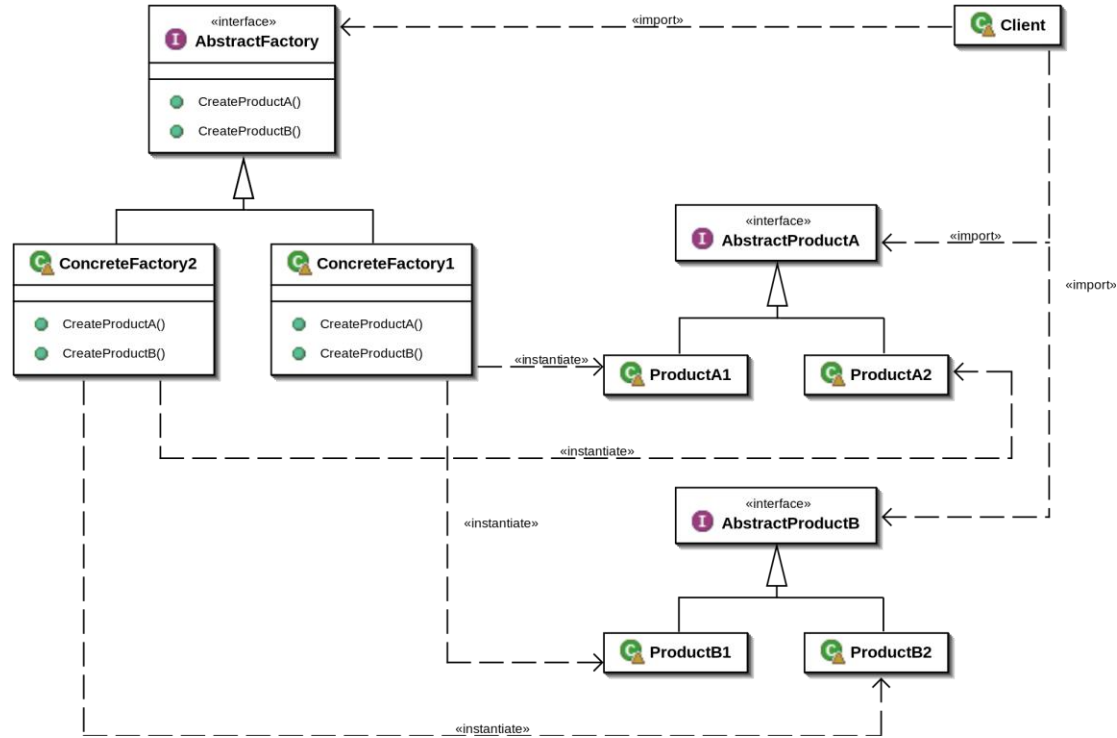
Factory Method/Abstract Factory

- 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

 <Document ID: change ID in footer or remove> <Change information classification in footer>

Factory Method


```
┌─────────────────────────────┐          «use»        ┌─────────────────────────────┐
│          Creator            │- - - - - - - - - - ->│          Product            │
├─────────────────────────────┤                       ├─────────────────────────────┤
│                             │                       │                             │
├─────────────────────────────┤                       └─────────────────────────────┘
│ # factoryMethod() : Product │                                     △
│ + anOperation()             │                                     │
└─────────────────────────────┘                                     │
              △                                                      │
              │                                                      │
┌─────────────────────────────┐        «create»       ┌─────────────────────────────┐
│       ConcreteCreator       │- - - - - - - - - - ->│       ConcreteProduct       │
├─────────────────────────────┤                       ├─────────────────────────────┤
│                             │                       │                             │
├─────────────────────────────┤                       └─────────────────────────────┘
│ + factoryMethod() : Product │
└─────────────────────────────┘
```

https://sourcemaking.com/

<Document ID: change ID in footer or remove> <Change information classification in footer>

Abstract Factory = set of Factory Methods

© 2019 Nokia   <Document ID: change ID in footer or remove> <Change information classification in footer>

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

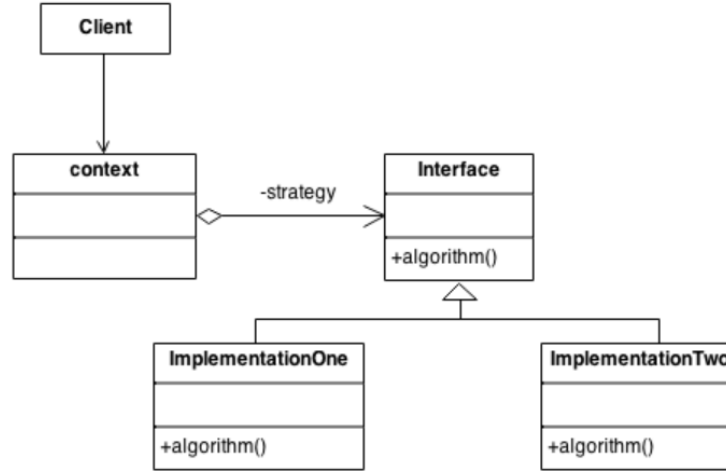© 2019 Nokia   <Document ID: change ID in footer or remove> <Change information classification in footer>

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.

     <Document ID: change ID in footer or remove> <Change information classification in footer>

Strategy

   <Document ID: change ID in footer or remove> <Change information classification in footer>

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

 <Document ID: change ID in footer or remove> <Change information classification in footer>

Strategy pattern

https://sourcemaking.com/

          <Document ID: change ID in footer or remove> <Change information classification in footer>

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.

 <Document ID: change ID in footer or remove> <Change information classification in footer>
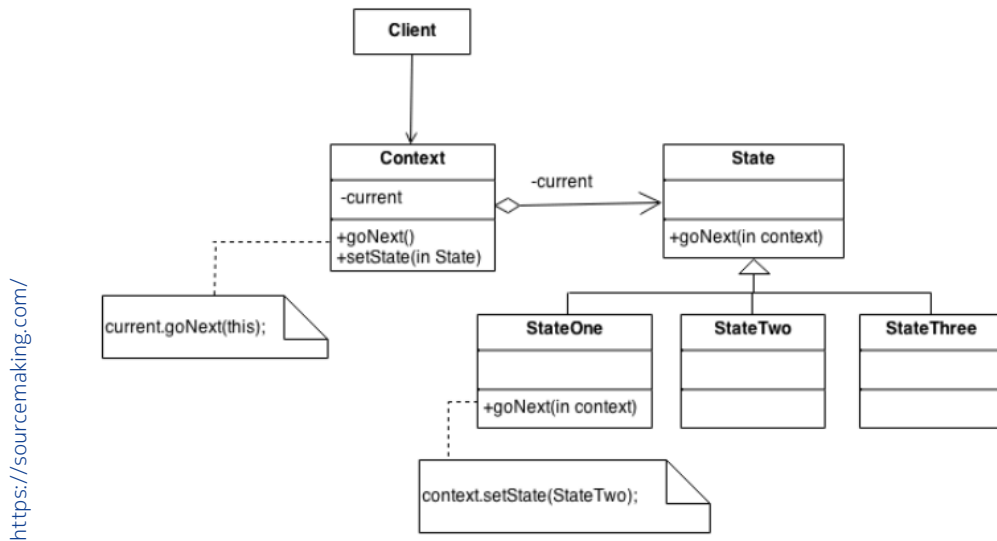
Strategy pattern – Applicability

- 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.

<Document ID: change ID in footer or remove> <Change information classification in footer>

State

     <Document ID: change ID in footer or remove> <Change information classification in footer>
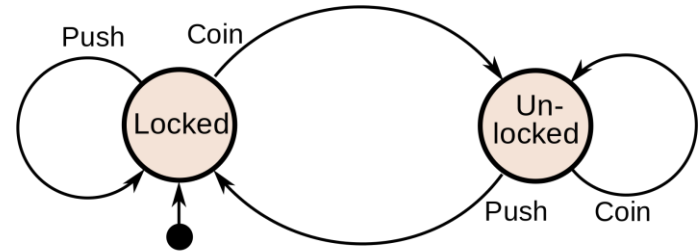
State pattern

- 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.

<Document ID: change ID in footer or remove> <Change information classification in footer>

State pattern

```
         ┌──────────┐
         │  Client  │
         └──────────┘
              │
              ▼
   ┌──────────────────┐        -current      ┌──────────────────┐
   │     Context      │                       │      State       │
   ├──────────────────┤◇─────────────────────▶├──────────────────┤
   │ -current         │                       │                  │
   ├──────────────────┤                       ├──────────────────┤
   │ +goNext()        │                       │ +goNext(in context)│
   │ +setState(in State)│                     └──────────────────┘
   └──────────────────┘                              △
                                                     │
                          ┌──────────────┬───────────┴──────────┐
   ┌────────────────┐ ┌──────────────┐ ┌──────────────┐ ┌──────────────┐
   │ current.goNext(this); │ │  StateOne    │ │   StateTwo   │ │  StateThree  │
   └────────────────┘ ├──────────────┤ ├──────────────┤ ├──────────────┤
                      │              │ │              │ │              │
                      ├──────────────┤ ├──────────────┤ ├──────────────┤
                      │+goNext(in context)│ │          │ │              │
                      └──────────────┘ └──────────────┘ └──────────────┘

              ┌────────────────────────┐
              │ context.setState(StateTwo); │
              └────────────────────────┘
```

© 2019 Nokia   <Document ID: change ID in footer or remove> <Change information classification in footer>

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

Push   Coin

Locked   Un-locked

Push   Coin

    <Document ID: change ID in footer or remove> <Change information classification in footer>

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.

<Document ID: change ID in footer or remove> <Change information classification in footer>

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

<Document ID: change ID in footer or remove> <Change information classification in footer>

Adapter

     <Document ID: change ID in footer or remove> <Change information classification in footer>

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

<Document ID: change ID in footer or remove> <Change information classification in footer>

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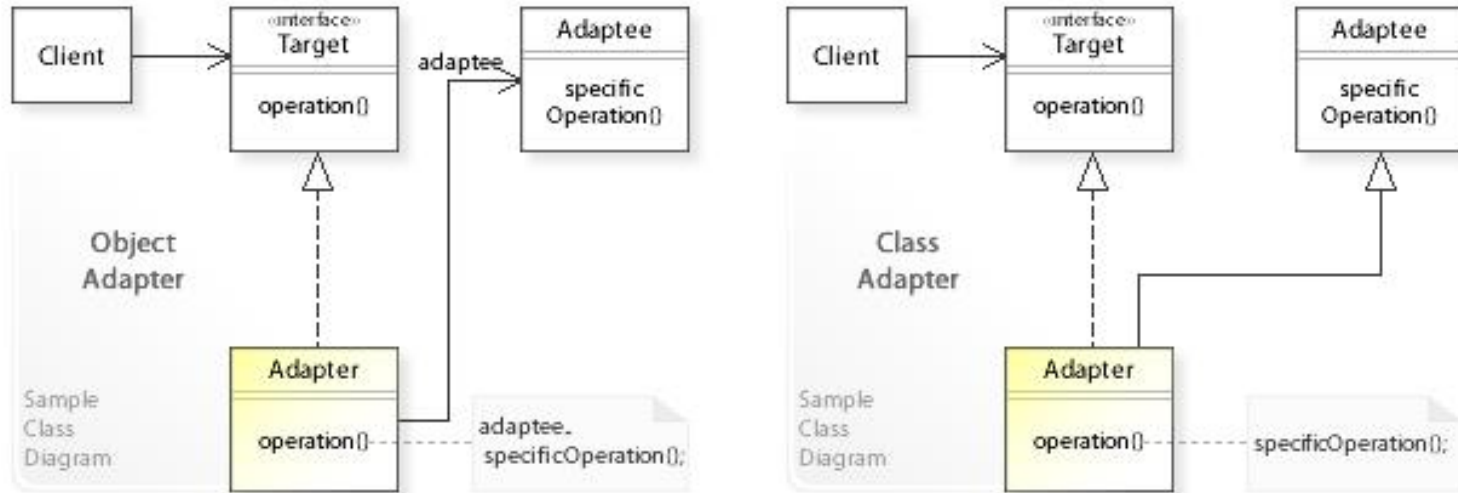
https://en.wikipedia.org/wiki/Adapter_pattern

<Document ID: change ID in footer or remove> <Change information classification in footer>

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.

     <Document ID: change ID in footer or remove> <Change information classification in footer>

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

          <Document ID: change ID in footer or remove> <Change information classification in footer>

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")

       <Document ID: change ID in footer or remove> <Change information classification in footer>

Decorator

https://sourcemaking.com/

<Document ID: change ID in footer or remove> <Change information classification in footer>
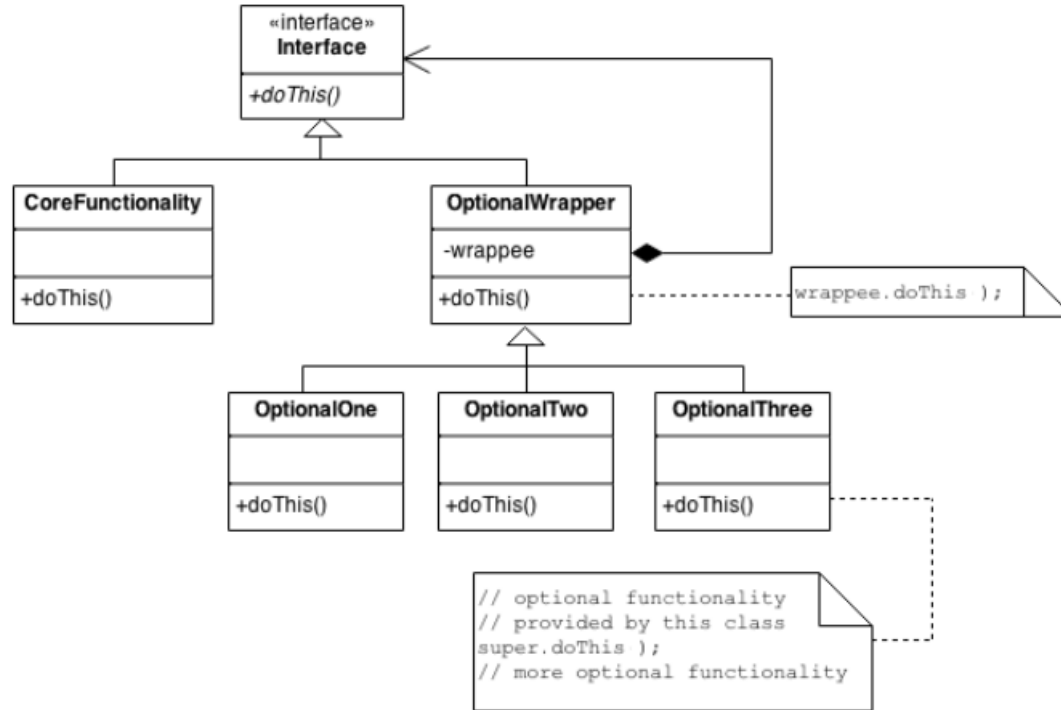
Decorator – Applicability

- To assign new functionalities at runtime without breaking the client code,

- To avoid unnecessary inheritance or when inheritance is not allowed (final keyword)
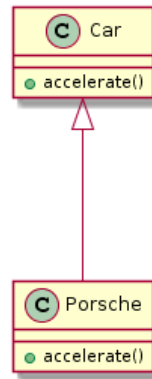
          <Document ID: change ID in footer or remove> <Change information classification in footer>

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
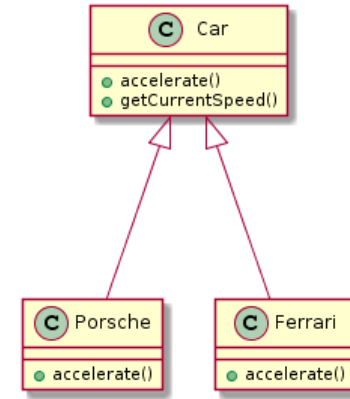
Exercise: Car Factory & Strategy

       <Document ID: change ID in footer or remove> <Change information classification in footer>

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.

```
┌─────────────────┐
│  C  Car         │
├─────────────────┤
├─────────────────┤
│  ● accelerate() │
└─────────────────┘
         △
         │
         │
┌─────────────────┐
│  C  Porsche     │
├─────────────────┤
├─────────────────┤
│  ● accelerate() │
└─────────────────┘
```

<Document ID: change ID in footer or remove> <Change information classification in footer>

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.


<Document ID: change ID in footer or remove> <Change information classification in footer>

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.

<Document ID: change ID in footer or remove> <Change information classification in footer>
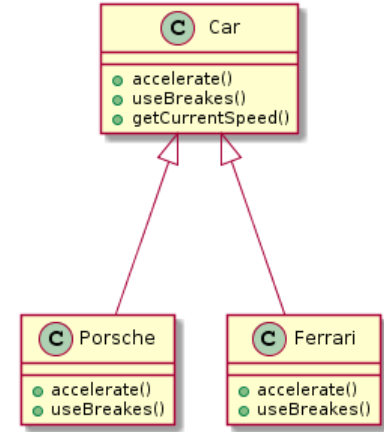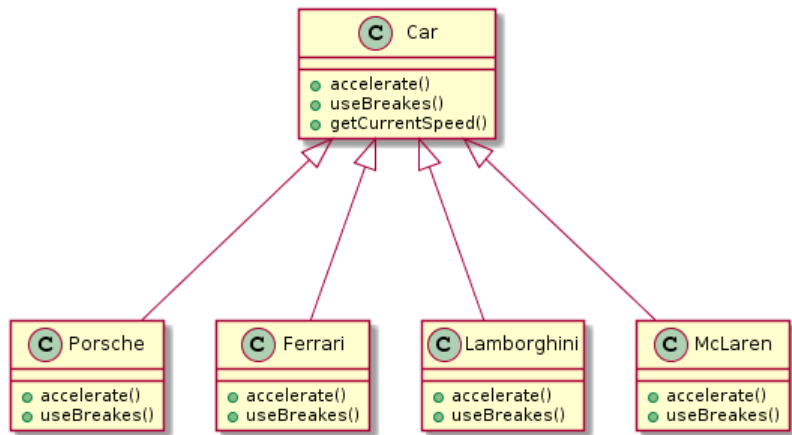
Exercise: Car Factory & Strategy

- 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.

         <Document ID: change ID in footer or remove> <Change information classification in footer>
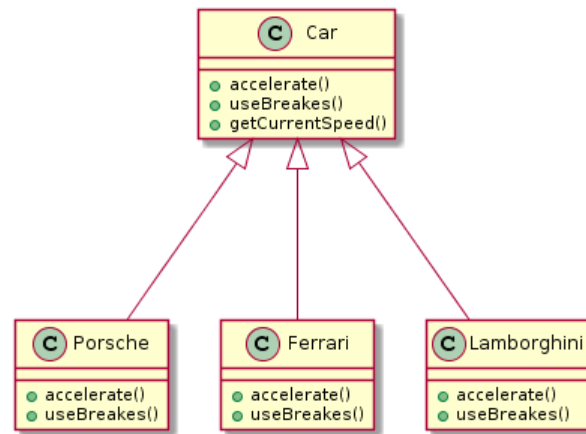
Exercise: Car Factory & Strategy

- 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.

Copyright and confidentiality

The contents of this document are proprietary and confidential property of Nokia. This document is provided subject to confidentiality obligations of the applicable agreement(s).

This document is intended for use of Nokia's customers and collaborators only for the purpose for which this document is submitted by Nokia. No part of this document may be reproduced or made available to the public or to any third party in any form or means without the prior written permission of Nokia. This document is to be used by properly trained professional personnel. Any use of the contents in this document is limited strictly to the use(s) specifically created in the applicable agreement(s) under which the document is submitted. The user of this document may voluntarily provide suggestions, comments or other feedback to Nokia in respect of the contents of this document ("Feedback").

Such Feedback may be used in Nokia products and related specifications or other documentation. Accordingly, if the user of this document gives Nokia Feedback on the contents of this document, Nokia may freely use, disclose, reproduce, license, distribute and otherwise commercialize the feedback in any Nokia product, technology, service, specification or other documentation.

Nokia operates a policy of ongoing development. Nokia reserves the right to make changes and improvements to any of the products and/or services described in this document or withdraw this document at any time without prior notice.

The contents of this document are provided "as is". Except as required by applicable law, no warranties of any kind, either express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose, are made in relation to the accuracy, reliability or contents of this document. NOKIA SHALL NOT BE RESPONSIBLE IN ANY EVENT FOR ERRORS IN THIS DOCUMENT or for any loss of data or income or any special, incidental, consequential, indirect or direct damages howsoever caused, that might arise from the use of this document or any contents of this document.

This document and the product(s) it describes are protected by copyright according to the applicable laws.

Nokia is a registered trademark of Nokia Corporation. Other product and company names mentioned herein may be trademarks or trade names of their respective owners.

    <Document ID: change ID in footer or remove> <Change information classification in footer>

NOKIA

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

    <Document ID: change ID in footer or remove> <Change information classification in footer>

NOKIA