NOKIA

Projektowanie złożonych systemów telekomunikacyjnych

Lecture 2: UML Basics & Design Patterns

Łukasz Marchewka

05-03-2020

Agenda:

- What is the Modelling and UML
  - Understanding the basics of UML
  - UML diagrams
    - Structural diagrams
    - Behavioral diagrams
      - Interaction diagrams
  - UML Modeling tools
    - OpenSource Tool – PlantUML

- Design Patterns
  - What are Design Patterns
    - Factory
    - Strategy
    - State

Modelling

Describing a system at a high level of abstraction

- A model of the system
- Used for requirements and specifications

Why do we model software systems?

- A model is a simplification, that helps in better understanding of the system

UML – Unified Modelling Language

- UML is a pictorial language for specifying, visualizing, constructing, and documenting the artifacts of software systems
- It is a simple modeling mechanism to model all possible practical systems
- UML is not a programming language but there are tools that can be used to generate code in various languages basing on UML diagrams
- UML is not dependent on any language or technology

Diagrams

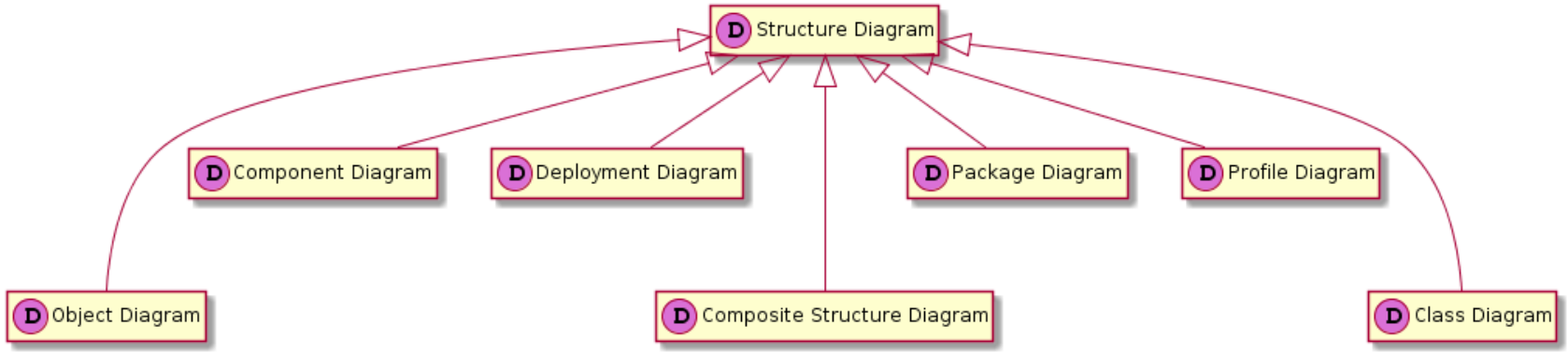
Structural Diagrams – describe static aspects of the software system using :

- Objects
- Attributes
- Operations and relationships

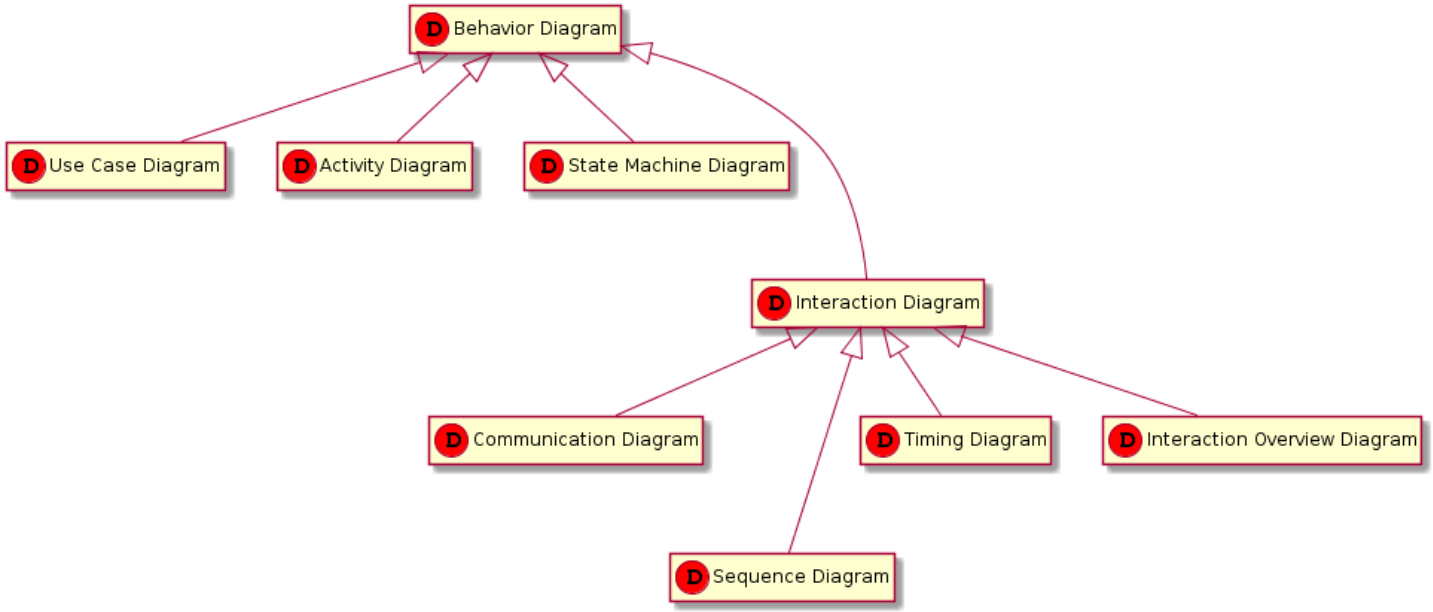
Behavioral (or Dynamic) Diagrams – describe dynamic aspects of the software system by showing collaboration among objects and changes to the internal states of objects:

- Use-case
- Interaction
- State Chart
- Activity

Structure Diagrams Hierarchy (Static)



Behavioral (Dynamic) Diagrams Hierarchy



NOKIA

Structure Diagrams

Class Diagram

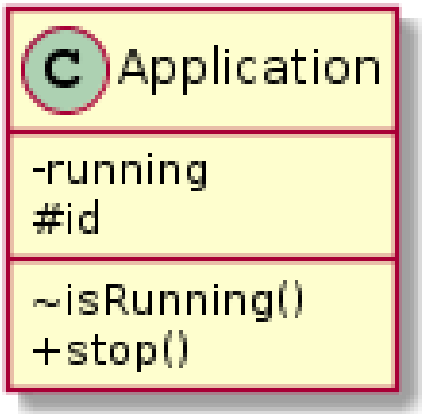
Class Diagram describes static structure of a system using:

- Classes/Interfaces
- Attributes
- Operations (or methods)
- Relationships between classes

Class

Describes a set of objects having similar:

- Attributes (status)
- Operations (behavior)
- Relationships with other classes



Attributes and operations may have their visibility marked:

"+" for public

"#" for protected

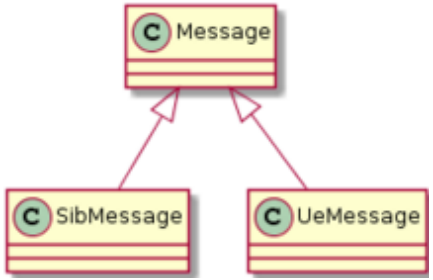
"–" for private

"~" for package

Class diagram – replationships between classes

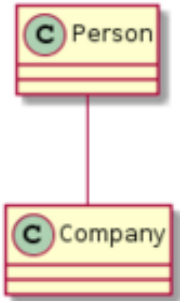
Generalization:

- Indicates that objects of the specialized class (subclass) are substitutable for objects of the generalized class (super-class)
- Generalization expresses a parent/child relationship among related classes
- Used for abstracting details in several layers



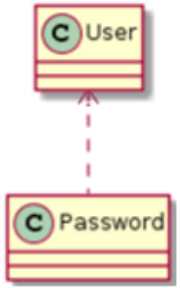
Association:

- Indicates that there is a relationship between objects, however the objects are independent



Dependency:

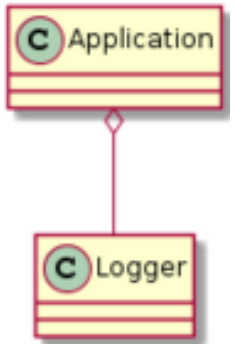
- An element, or set of elements, requires other model elements for their specification or implementation



Class diagram – replationships between classes

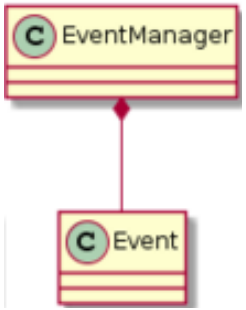
Aggregation:

- It can be treated as variant of the "has a" association relationship.
- It represents a part-whole or part-of relationship.



Composition:

- A strong form of aggregation
- Whole/part relationship -> The whole is the sole owner of its part
- If a composite (whole) is deleted, all of its composite parts are also deleted

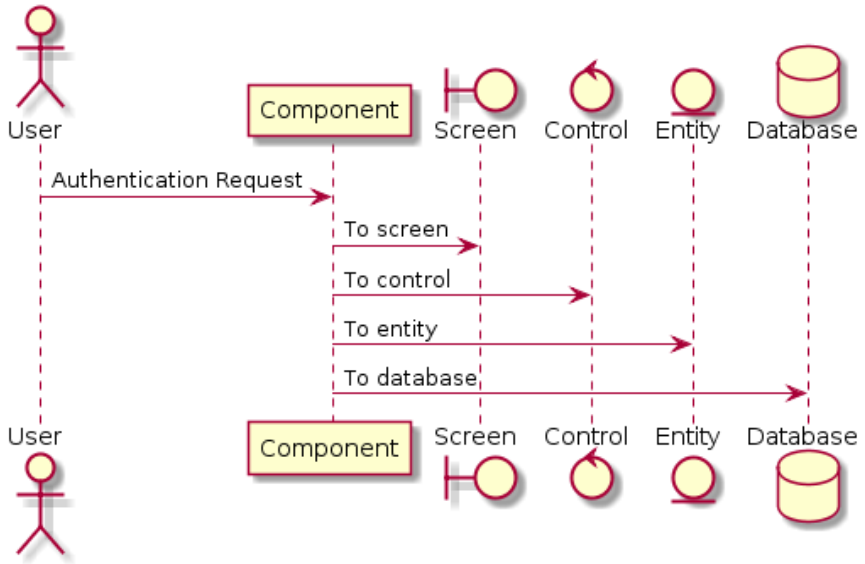


NOKIA

Behavioral Diagrams

Sequence Diagram

- Describes the interactions between objects in the sequential order
- The diagram conveys this information along the horizontal and vertical dimensions:
  - the vertical dimension shows, top down, the time sequence of messages/calls as they occur, and the horizontal dimension shows, left to right, the object instances that the messages are sent to.



Sequence Diagram

Lifeline:

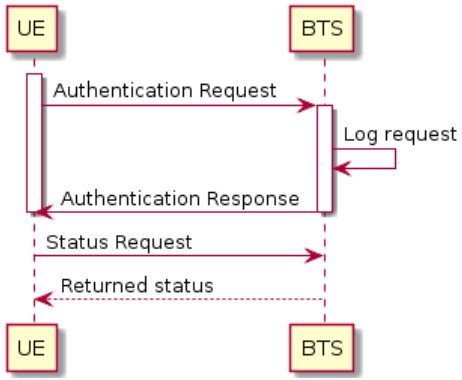
- A vertical line that represents the sequence of events that occur in a participant during an interaction, while time progresses down the line. This participant can be an instance of a class, component, or actor.

Messages:

- Messages are arrows that represent communication between objects. Half-arrowed lines are used to represent asynchronous messages. Asynchronous messages are sent from an object that will not wait for a response from the receiver before continuing its tasks.

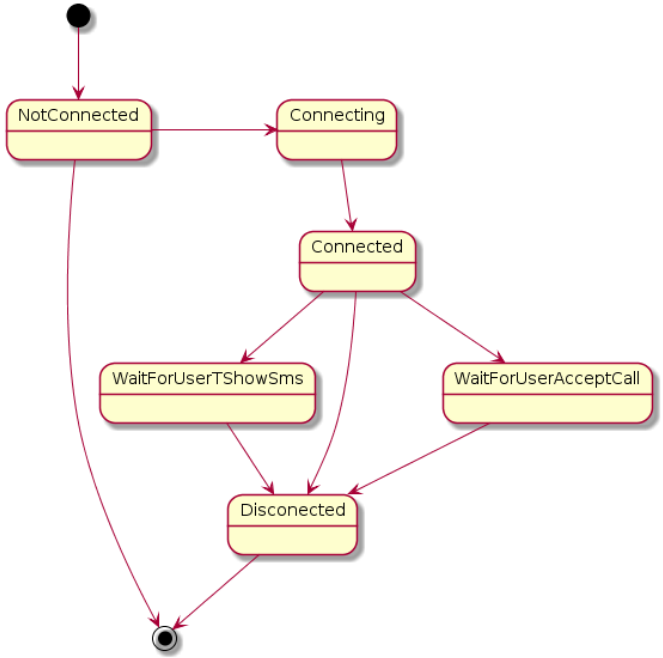
Activation boxes:

- Represent the time an object needs to complete a task. When an object is busy executing a process or waiting for a reply message, use a thin rectangle placed vertically on



State (Machine) Diagram

- Describes the behavior of a single object and the flow of control from one state to another
- Specifies a sequence of events that an object goes through during its lifetime in response to events.



State (Machine) Diagram

State is defined as a condition in which an object exists and it changes when some event is triggered

Initial State:

- A filled circle followed by an arrow represents the object's initial state

Final State:

- An arrow pointing to a filled circle nested inside another circle represents the object's final state

Transition:

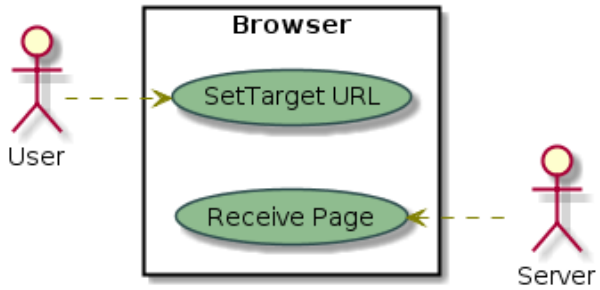
- A solid arrow represents the path between different states of an object. Label the transition with the event that triggered it and the action that results from it. A state can have a transition that points back to itself

Decision points:

- A decision point models a choice that must be made within the sequence of states

Use case diagram

- Ilustrates a unit of functionality provided by the system.
- Visualizes the different types of roles in a system and how those roles interact with the system.
- Identify functions and how roles interact with them
- High level view of the system
- Identify internal and external factors



Use Case Diagram objects

Use case diagrams consist of 4 objects:

Actor:

- Any entity that performs a role in one given system. This could be a person, organization or an external system

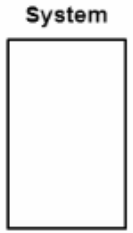
First Actor

Use case:

- Represents a function or an action within the system.

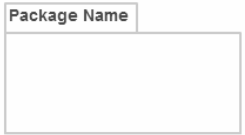
System (optional):

- Defines the scope of the use case, an optional element but useful when your visualizing large systems.



Package (optional):

- Used to group use cases together.



PlantUML

PlantUML is an Open Source project used to draw UML diagram, using a simple and human readable text description.

It's more a *drawing* tool than a *modeling* tool.

http://plantuml.com/

What are 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 align with good programming practices

- Classification
    - Creational
    - Structural
    - Behavioral
    - Concurrency

Criterias for embedded software:

- Performance,
- Deterministic,
- Low amount of reusable objects in 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.

Factory Method / Abstract Factory

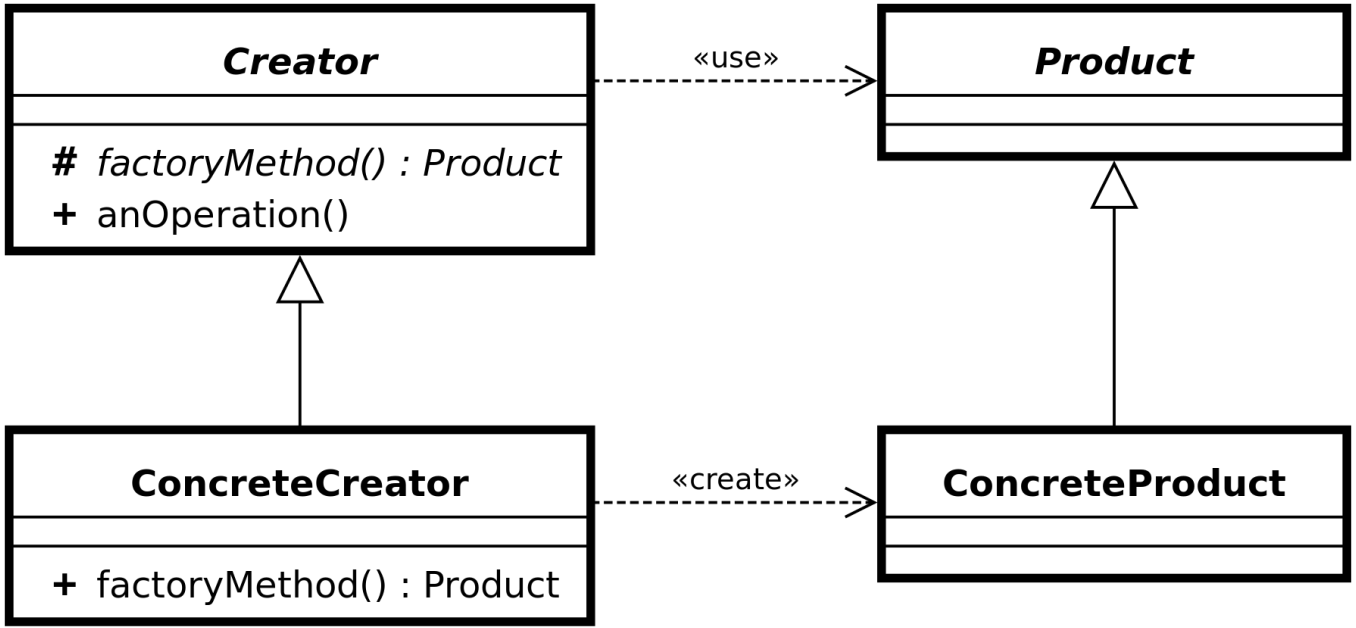
Factory Method/Abstract Factory

- Creational patterns
- Encapsulate creating objects and hide this process (support dependency injection)
- Allows to avoid explicit usage of *new* and sometimes *if* or *#ifdef*
- Encapsulates business logic related with object creation
- May accept parameters and create objects depending on them
- Abstract Factory = set of Factory Methods
- Factory Method creates one object, Abstract Factory – a family of objects

Factory Method/Abstract Factory

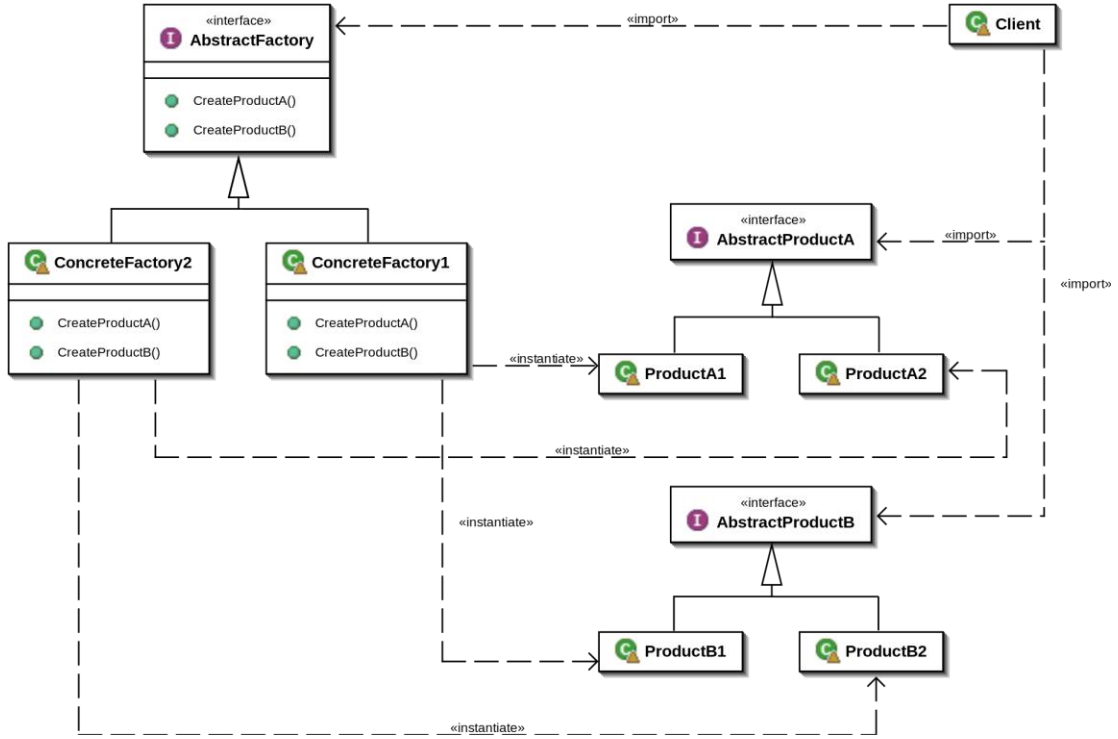
- Decouples code for object hierarchy construction from runtime logic code, thus helps enforce Single Responsibility Rule,
- Enables testing: TestSuites are factories that replace real dependencies with mocks,
- The Factory Method separates product construction code from the code that actually uses the product,
- Can manage objects lifetime,
- Enables implementation of other design patterns by removing static dependencies between classes.

Factory Method



https://sourcemaking.com/

Abstract Factory



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
+ Follow Open/Closed Principle

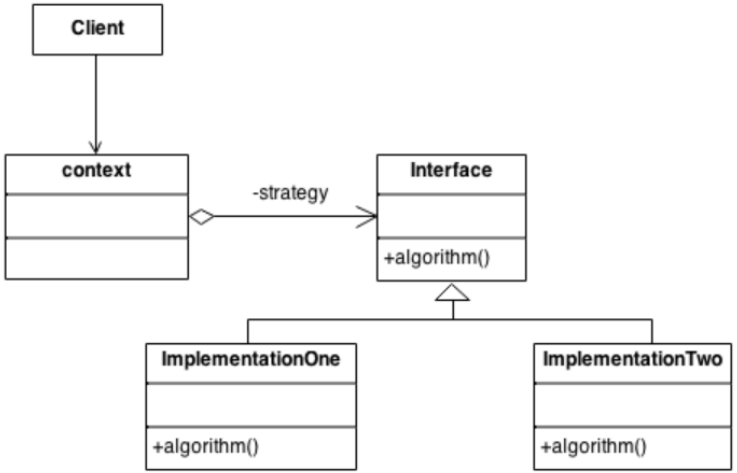
- The code may become complicated

- Use the Factory Method when you want to provide users of your library or framework with a way to extend its internal components.
- Use the Factory Method when you don’t know beforehand the exact types and dependencies of the objects your code should work with.

Strategy

Strategy pattern

- Behavioral design pattern
- Allows to select an algorithm or specific behavior **at runtime** (based on composition, not inheritance)
- Provides good decoupling and independent testing of algorithms



https://sourcemaking.com/

Strategy pattern

- 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 reference, pointers or smart pointers.
- Holding strategy by reference is permanent for the lifetime of object, strategies hold by pointers can be changed during object lifetime
- Object can have multiple strategy hierarchies

Strategy pattern – Pros and Cons

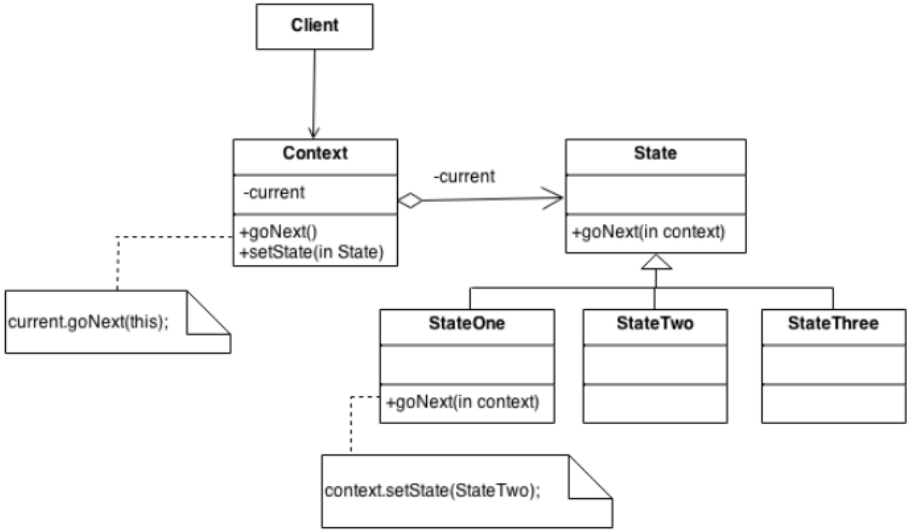
+ You can swap algorithms used inside an object at runtime.
+ You can isolate the implementation details of an algorithm from the code that uses it.
+ You can replace inheritance with composition.
+ Open/Closed Principle. You can introduce new strategies without having to change the context.

- If you only have a couple of algorithms and they rarely change, there's no real reason to overcomplicate the program with new classes and interfaces that come along with the pattern.
- Clients must be aware of the differences between strategies to be able to select a proper one.
- A lot of modern programming languages have functional type support that lets you implement different versions of an algorithm inside a set of anonymous functions. Then you could use these functions exactly as you'd have used the strategy objects, but without bloating your code with extra classes and interfaces.

State

State pattern

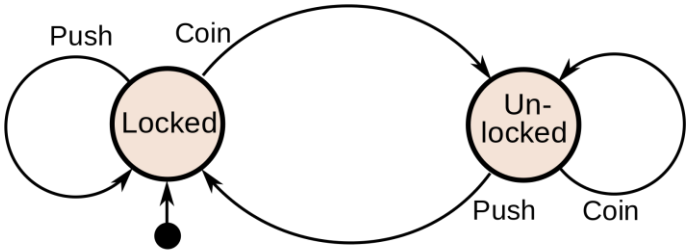
- Behavioral design pattern
- Class diagram looks the same as for Strategy – the difference is in the intention
- Provides good decoupling and independent testing of algorithms



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 - Summary

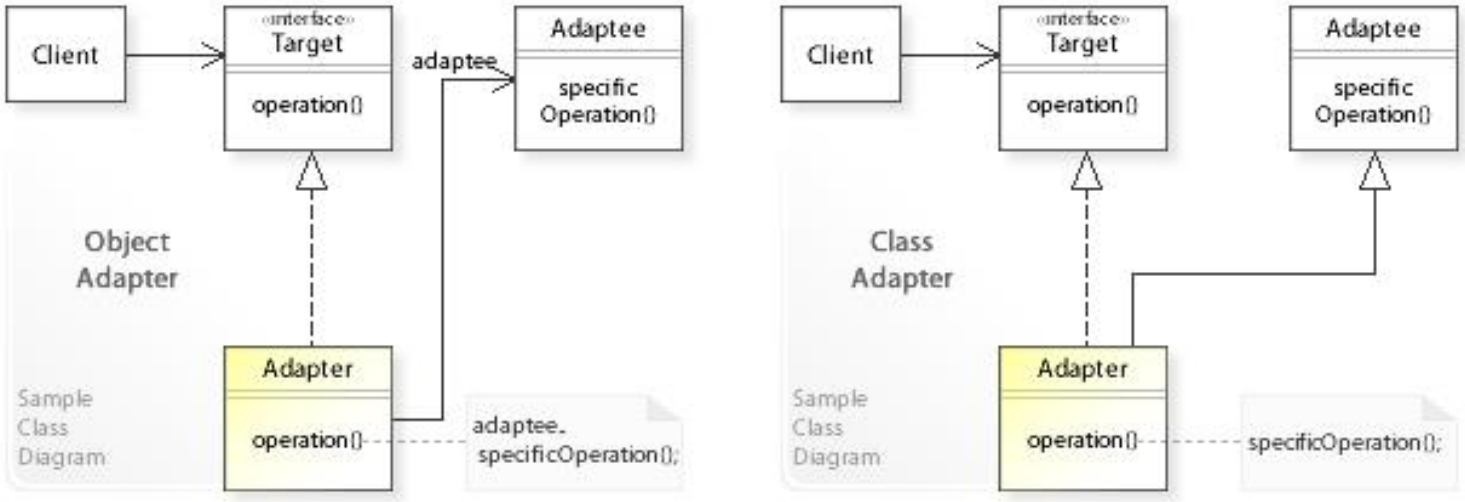
- Combines state of a class with code to be run in given state,
- Uses polymorphism instead of conditional statements in order to pick appropriate actions in the given state
- Improves readability of transitions between states,
- Class can have multiple state hierarchies.

Adapter

Adapter

- Structural design pattern
- Allows two classes with incompatible interfaces work together
- Should not contain logic
- In align with “open-close” principle
- Two possible ways of implementation – object and class

Adapter



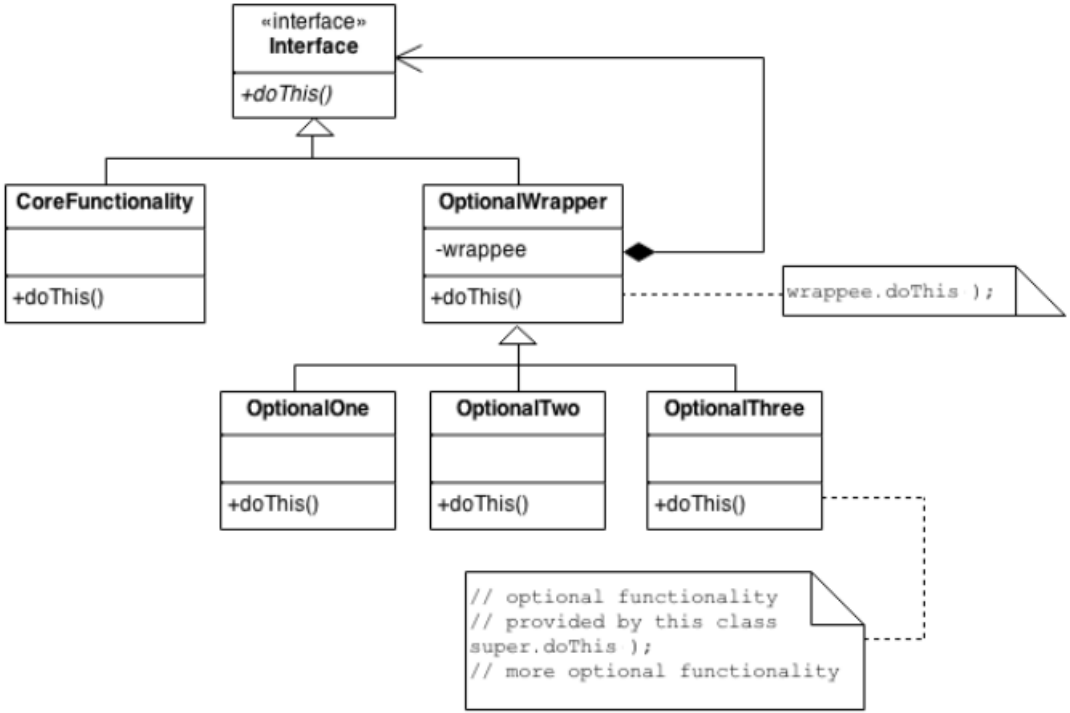
https://en.wikipedia.org/wiki/Adapter_pattern

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”)
- Changes “skin” of an object, and not “guts”

Decorator

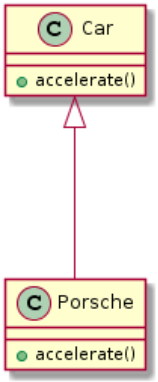


https://sourcemaking.com/

Exercise: Car Factory & Strategy

Exercise: Car Factory & Strategy

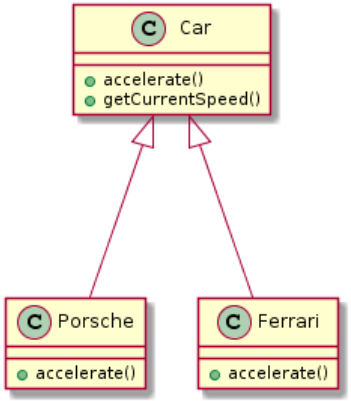
- Porsche:
  - Has 200 HP
  - Has Cx of 0.66
  - Accelerates each time by HP/2
  - Has top speed limited by HP/Cx
- Test top speed on the test track that accepts Car.



https://sourcemaking.com/

Exercise: Car Factory & Strategy

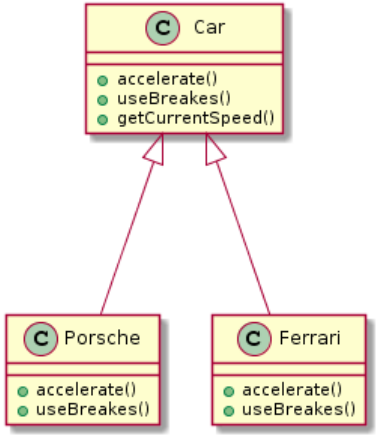
- Ferrari:
  - Has 300HP
  - Accelerates according to sequence HP/2, HP/4, HP/8 …
  - Has unlimited top speed
- Test it on the test track together with Porsche.



https://sourcemaking.com/

Exercise: Car Factory & Strategy

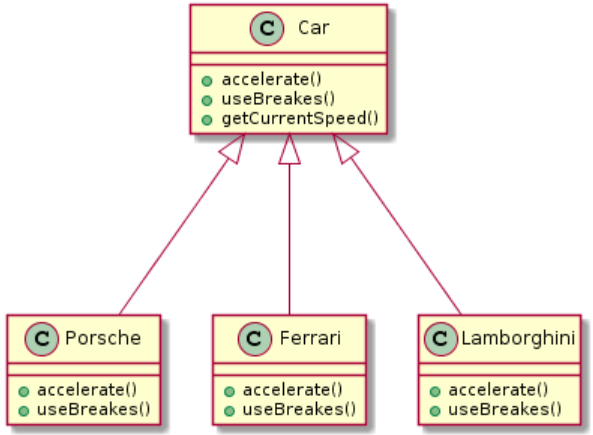
- Add breakes:
  - Ferrari stops on a dime (useBrakes() sets currentSpeed to 0)
  - Porsche useBrakes()
    - If currentSpeed < 50 sets currentSpeed to 0
    - Otherwise sets currentSpeed to currentSpeed/2
- Test brakes of both cars on the test track.



https://sourcemaking.com/

Exercise: Car Factory & Strategy

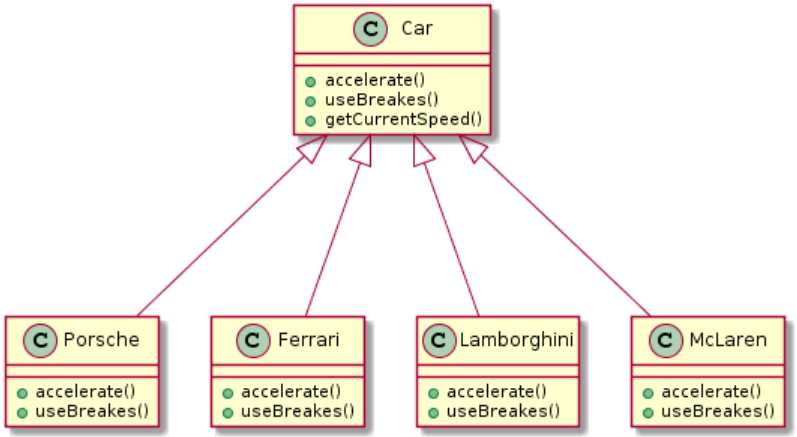
- Lamborghini:
  - Has 400HP
  - Accelerates like Porsche
  - Brakes like Ferrari
  - Has top speed limited by 315
- Test it on the test track together with other cars.



https://sourcemaking.com/

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.



https://sourcemaking.com/


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