Design Patterns 448.058 (VO)

Michael Krisper
Georg Macher

16.10.2019

Revision from last time…

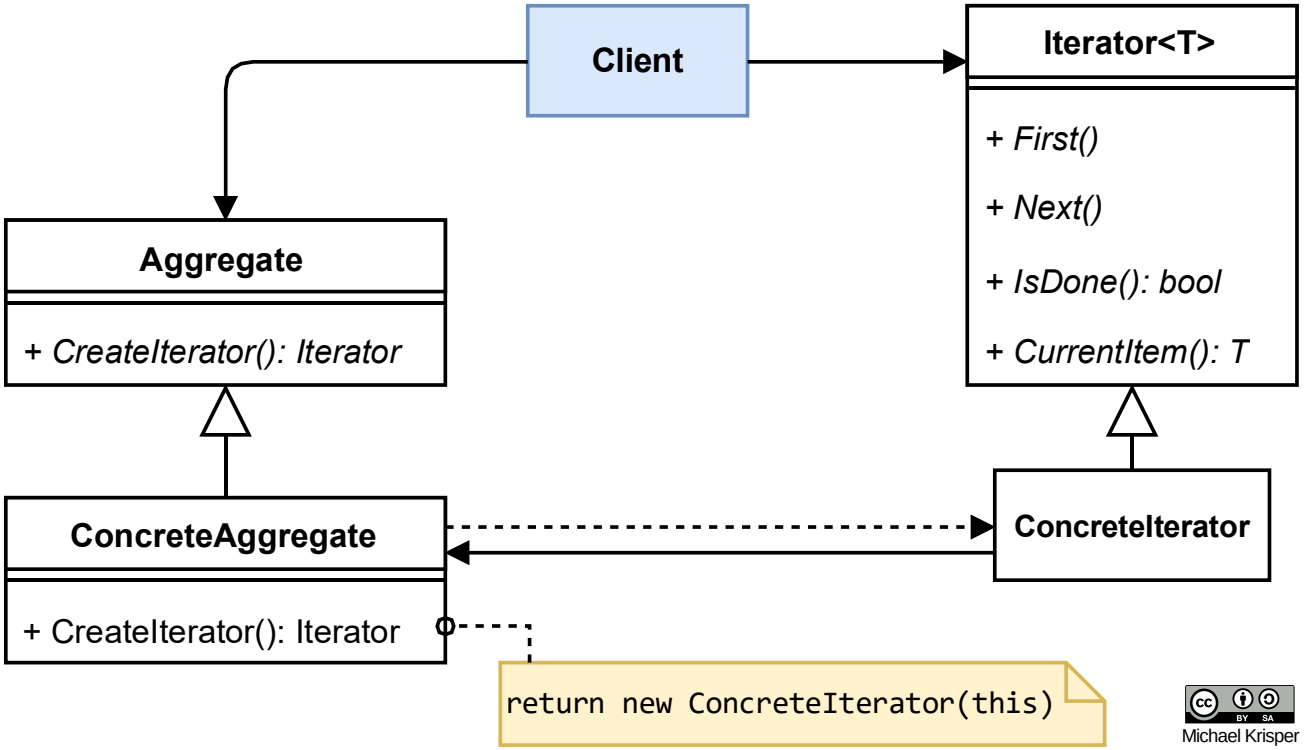
- SOLID Principles
  - **S**ingle Responsibility
  - **O**pen Closed
  - **L**iskov Substitution
  - **I**nterface Segregation
  - **D**ependency Inversion

- Principles of Good Programming:
  - Decomposition
  - Abstraction
  - Decoupling
  - Usability & Simplicity

- Design Pattern
  - Layers
  - Iterator
  - Observer

Iterator

Retrieve items of a collection element by element.

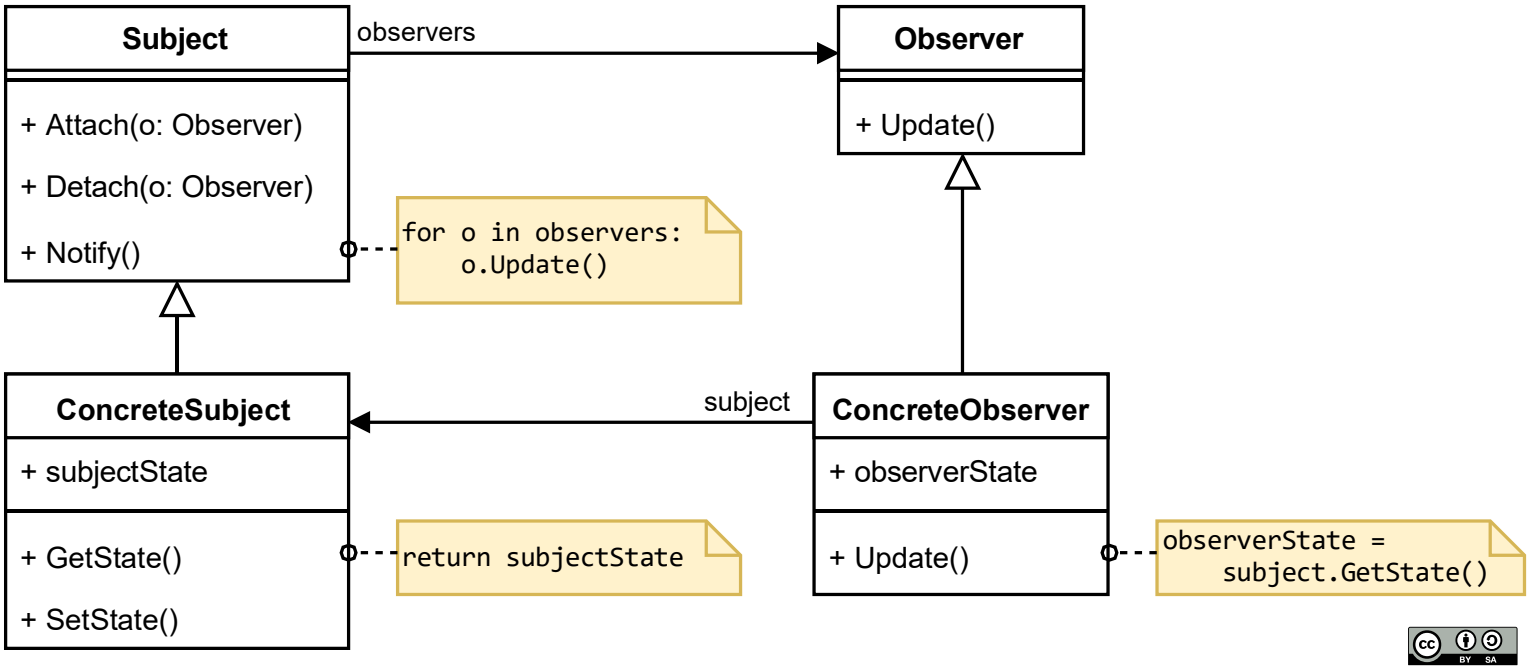


Learning Goals for Today

- Understand and describe some design patterns:
  - Observer
  - Factory Method
  - Strategy
  - Command
- Explain the core ideas of these patterns in own words
- Derive the Problem, Forces, Solution, and Consequences of these patterns

Observer

Inform registered observers about changes.



Observer

Context: data is distributed over multiple related objects.

Problem: Maintain consistency between related objects.

Forces:

- When one object changes, others should be held consistent.
- Polling is very costly or not possible.
- The other objects are not known at compile-time and should not be tightly coupled.
- Reuse even in isolation should be possible.

Solution:

- Define means to manage observers for a subject (register, unregister).
- On changes: notify all observers that a change happened.
- Give the observers the possibility to access the changed data.

Consequences:

\+ Decouple subjects and observers.
\+ Reuse subjects and observers.
\+ Polling is not needed anymore.
\+ Support for m:n communication.
\- Unexpected updates / Frequent updates / Cascading updates.
~ Synchronous vs. Asynchronous updates!
~ Who initiates the update?

Observer - Example

```
abstract class Subject {
  private readonly List<Observer> _observers = new List<Observer>();
  public void Attach(Observer observer) => _observers.Add(observer);
  public void Detach(Observer observer) => _observers.Remove(observer);
  public void Notify() => _observers.ForEach(o => o.Update());
}
```

```
abstract class Observer {
  public abstract void Update();
}
```

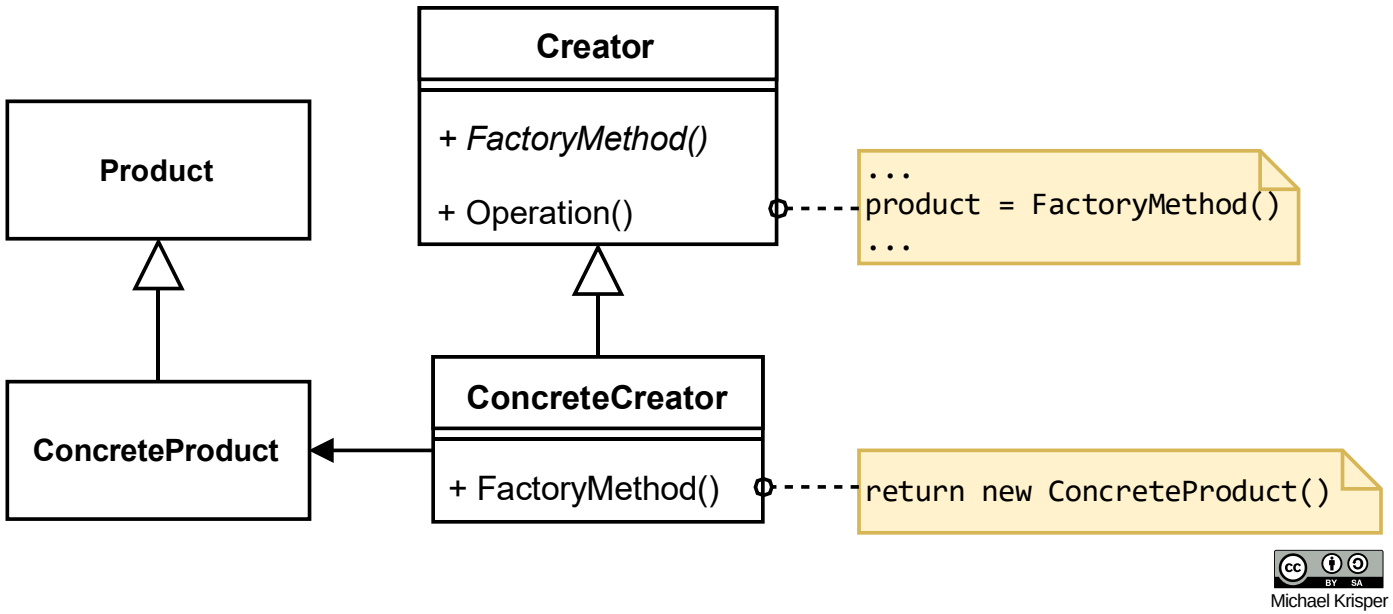
```
class ConcreteSubject : Subject {
  private string _state;
  public string State {
    get { _state; }
    set {
      _state = value;
      Notify();
    }
  }
}
```

```
class ConcreteObserver : Observer {
  public ConcreteSubject Subject;
  private readonly string _name;
  public ConcreteObserver(ConcreteSubject subject, string name) {
    Subject = subject;
    subject.Attach(this);
    _name = name;
  }
  public override void Update() {
    Console.WriteLine($"Observer {_name} was informed
                        that subject changed: {Subject.State}");
  }
}
```

```
void Main() {
  var s = new ConcreteSubject();
  var o1 = new ConcreteObserver(s, "O1");
  s.State = "Change 1";
}
```

Factory Method

Delegate the creation of objects to someone else.



Factory Method

Context: Creation of an object, whose class is not known until runtime.

Problem: How to create an object for which the concrete class is not known.

Forces:

- We **don't care which object** is created, as long as it provides the **same functionality**.
- We **can't anticipate** the class we want to create at coding time.
- We want to **shift the decision** to someone else.

Solution:

- Define an interface of capabilities your objects must implement.
- Define some means (method or own class) to create the actual object somewhere else.
- Let the actual object implement the needed interface.

Consequences:

\+ Isolates Framework and Application code
\+ Flexibility (Compiletime/Runtime)
\+ Lesser Dependencies
\+ Connects parallel class hierarchies
\+ Decoupling of Implementation and Usage
\+ Abstraction of actual instances
~ Hides constructors
\- Needs an interface/abstraction layer!

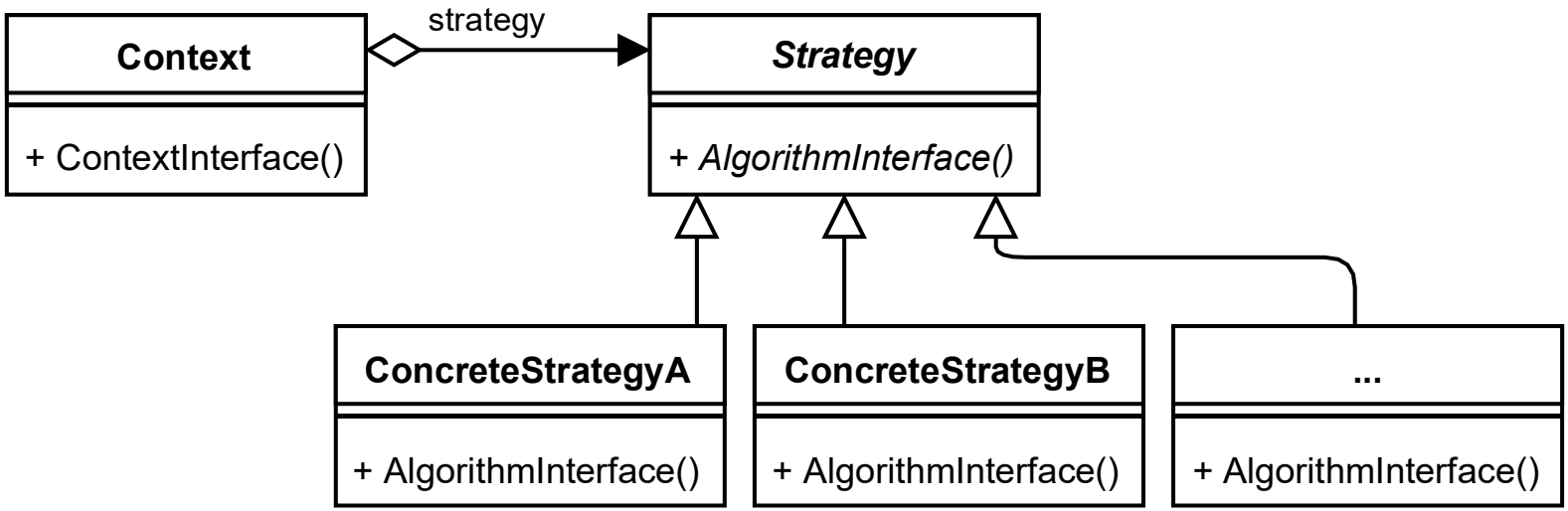
Factory Method – Implementation Issues

- Naming Convention (e.g. `XyzFactory`)
- Constructor Parameters?
- Universal-God-Interface vs. Duck Typing
- How to avoid direct constructions?
  (Private Constructor?)

- Abstract Creator (subclasses must implement)
  vs.
  Concrete Creator (default implementation, but subclasses can override)

Strategy

Substitute behaviour later.



Strategy

Context:

- Many related classes which differ only in their behaviour.
- Methods with complex behaviour based on many conditionals.

Problem:

How to manage the different behaviours and simplify the architecture?

Forces:

- You need different variants for an algorithm.
- The behaviour should be exchangeable at runtime.
- You want to split up behaviour of classes to simplify it.

Solution:

- Define interfaces for algorithms
- Encapsulate the algorithms to make them interchangeable.
- Let the algorithm vary independently from the clients.

Consequences:

\+ Split up behaviour and decision logic.

\+ Elimination of Subclasses just for different behaviour (composition over inheritance!)

\+ Reuse: Behaviour of one class can be reused for others.

\- Communication overhead

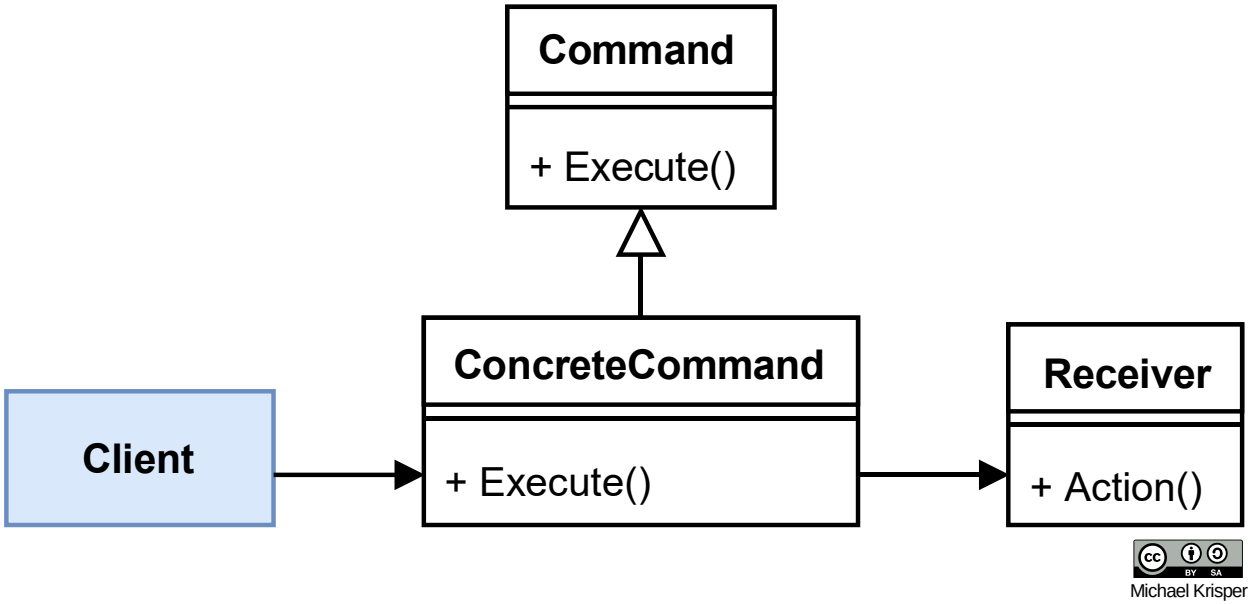
\- Access to private fields?

\- Increased number of objects (every behaviour is an own object)

~ Who assembles the concrete strategies at runtime?

Command

Encapsulate a request. Decouple invocation from execution.



Command

Context:

Invoking some behaviour of an object

Problem:

We just want to invoke an operation, regardless of its concrete implementation and executing context.

Forces:

- Avoid coupling of the invoker and the context of the request.
- We do not know the exact implementation of a request
- A request should be undoable

Solution:

- Define an interface for commands with a very simple interface (just Execute()).
- Encapsulate the behaviour in concrete commands implementing this interface and containing all needed parameters as members.
- Implement means to let the client initialise the parameters.

Consequences:

+ A request does not depend upon the creating class anymore.
+ A request can be executed in isolation.
+ Undo/Redo-Operations become possible
+ Switching the receiver at runtime becomes possible
+ Behaviour can be reused for multiple receivers.
- Increased number of objects
- References to all needed parameters must be stored

Summary

Patterns:

- Observer
- Factory Method
- Strategy
- Command