Design Patterns 448.058 (VO)

Michael Krisper
Georg Macher

09.10.2019

Revision from last time…

- Organisation of the Courses: VO, UE
  - VO: Exam on 29.01.2020
  - UE: Projects + Presentations in groups á 2-3 students

- Learning Theory
  - 20 Minutes Attention Span
  - **Repeat and Exercise** for Long Term Memory
  - Learning Goals: Imitate, Apply, Derive, Develop

- Design Patterns Theory
  - What is a Design Pattern?
  - Pattern Form:
    Name, Context, Problem, Forces, Solution, Consequences
  - Pattern Languages

- Layers

Design Patterns

“A proven solution template for a recurring problem.”

Purpose:

- Easier knowledge transfer
- Resuing existing ideas
- Better communication / Common vocabulary
- Generalizing the solution

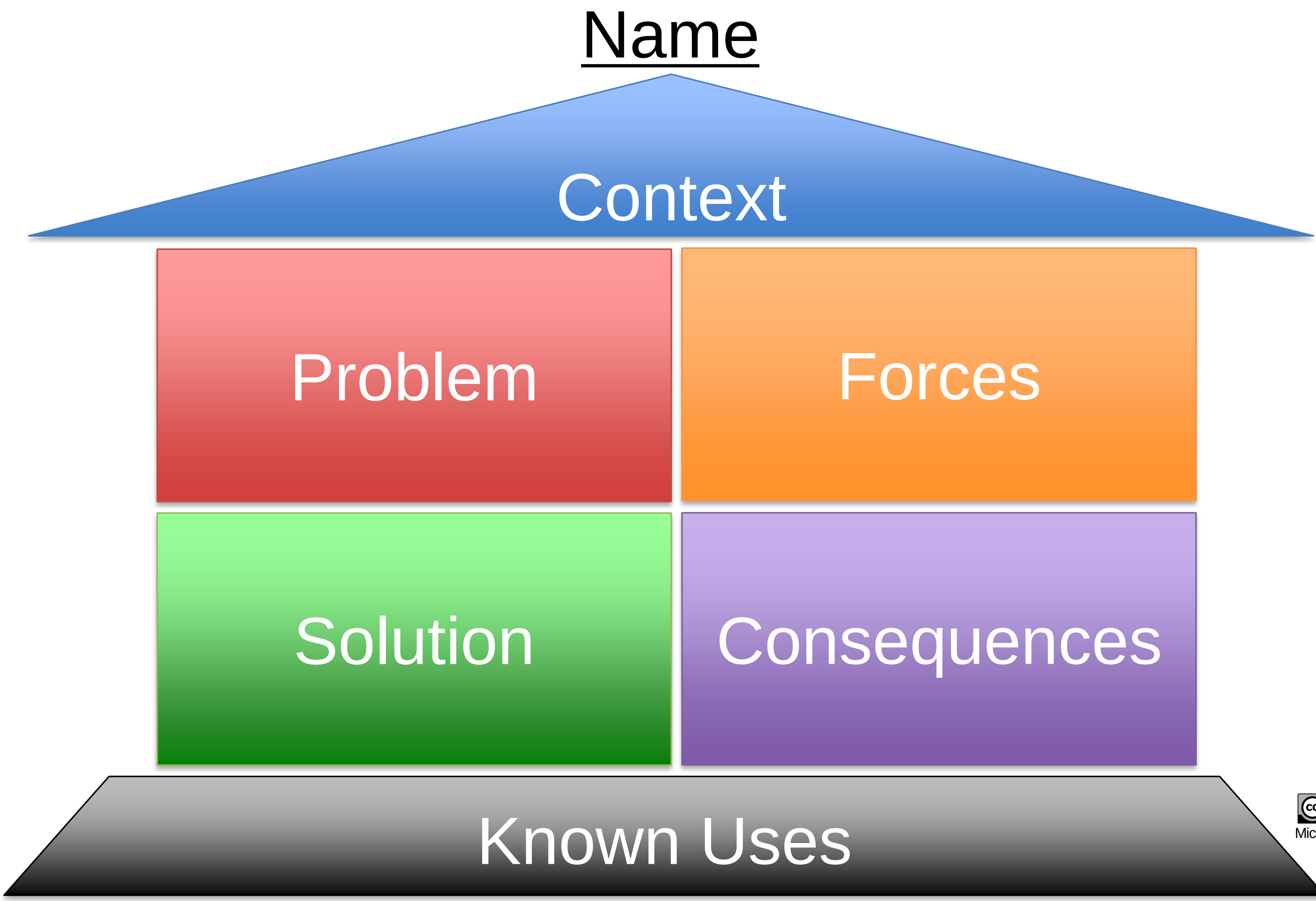
Software Design Patterns:

- Architectural Patterns
- Design Patterns
- Idioms

Pattern format

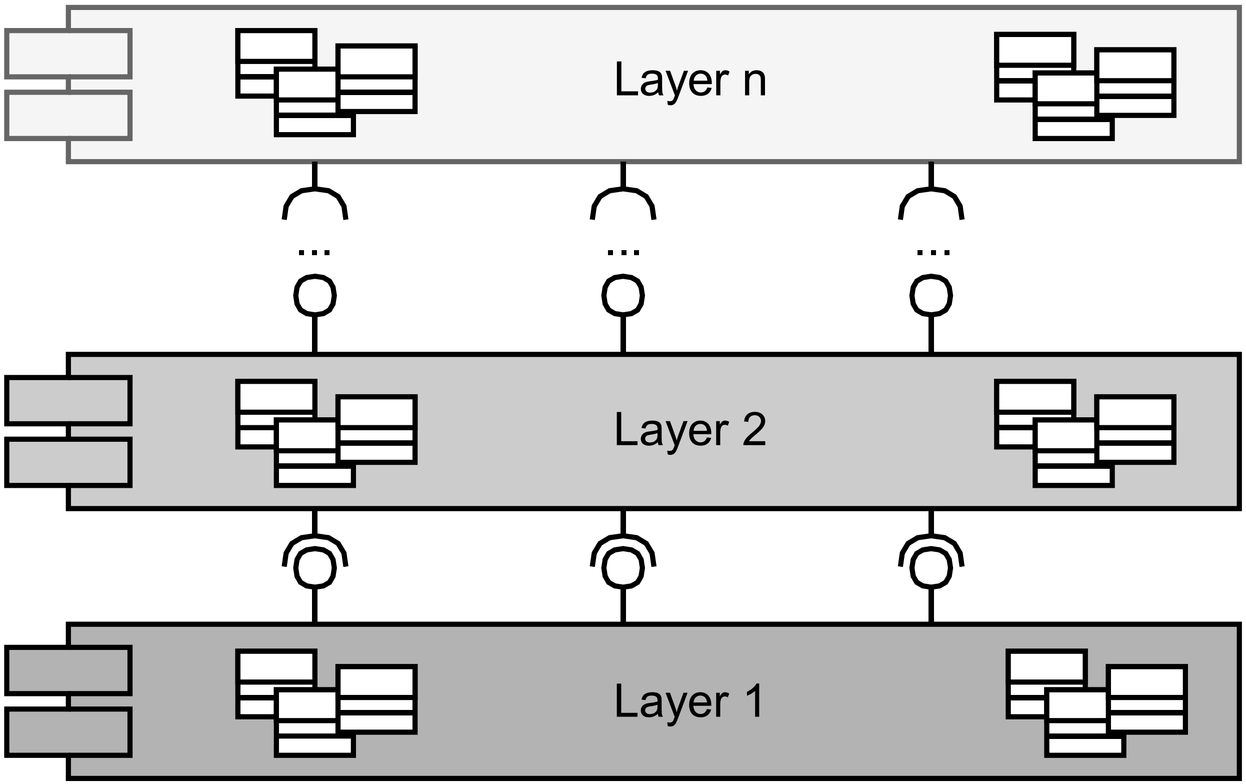
- **Name**: A catchy name for the pattern
- **Context**: The situation where the problem occurs
- **Problem**: General Problem Description
- **Forces**: Requirements and Constraints - Why does the problem hurt in this context?
- **Solution**: Generic Description of a proven solution.
  Static Structures, Dynamic Behaviour, Actionable Steps
- **Consequences:**
  - What are the benefits and drawbacks? Pro and Contra?
  - What are the liabilities, limitations and tradeoffs?
  - How are the forces resolved?
- **Known-Uses**: Real Life Examples

The Design Pattern House



Layers

Split your system into layers based on abstraction levels



Layers

Context: Large systems that require decomposition

Problem:

- Many functions and responsibilities
- Hard to understand structure, many dependencies

Forces:

- Changes should be limited to one component
- Clear boundaries of responsibility
- Interfaces should be stable
- Parts should be exchangeable
- Parts should be reusable
- Smaller groups for easier understandability, maintainability

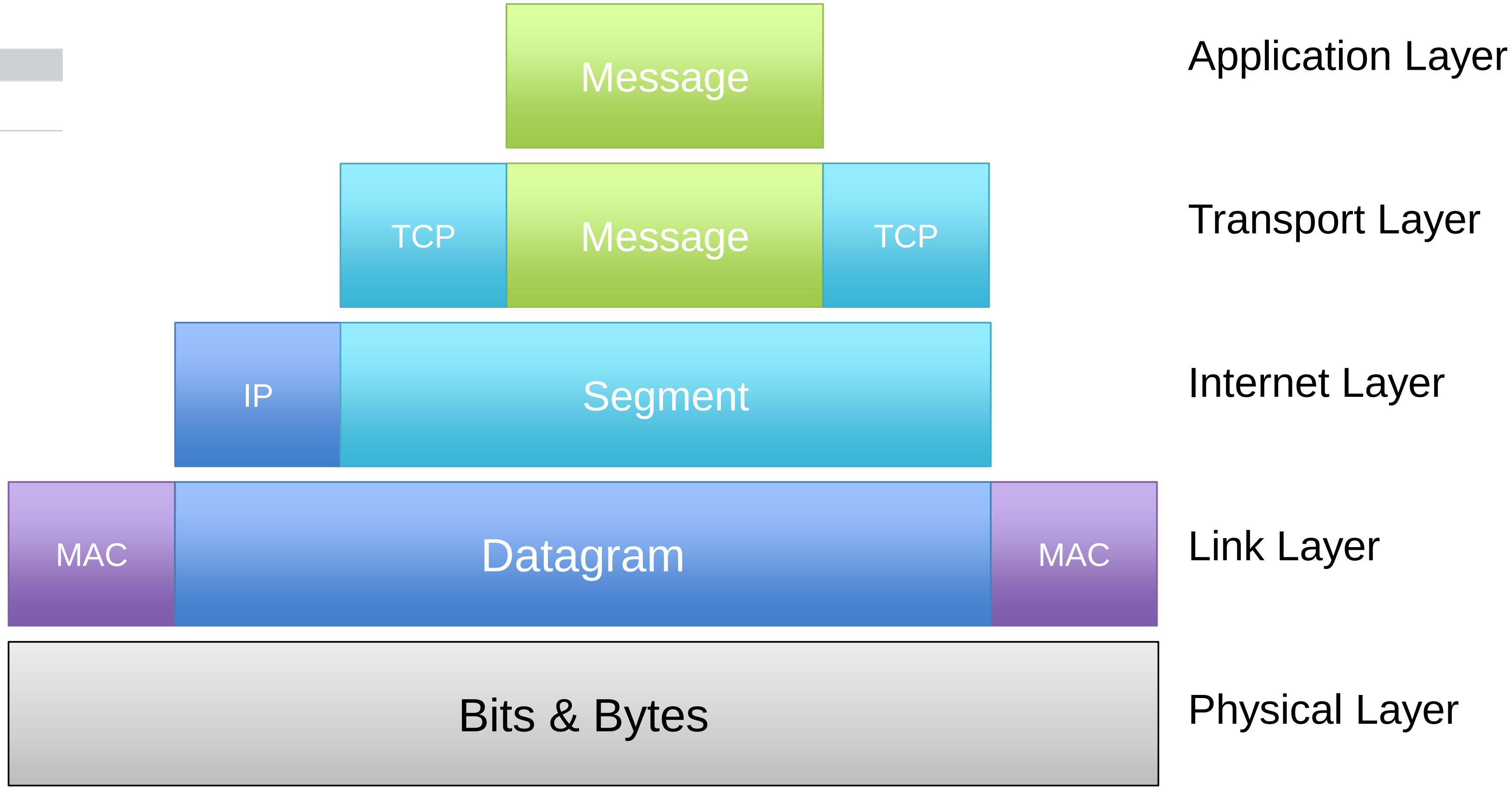
Solution:

- Structure the function into appropriate number of layers, based on their abstraction levels
- Every layer uses defined services of sublayer
- Every layer provides defined services to upper layer

Consequences:

+ Dependencies/Changes are kept local
+ Defined Interfaces between Layers
+ Layers are exchangeable & reusable
- Lower efficiency
- No fine grained control of sublayers
- Changes cascade and are costly
- Right granularity is difficult to find

Layers – Example & Live Demo



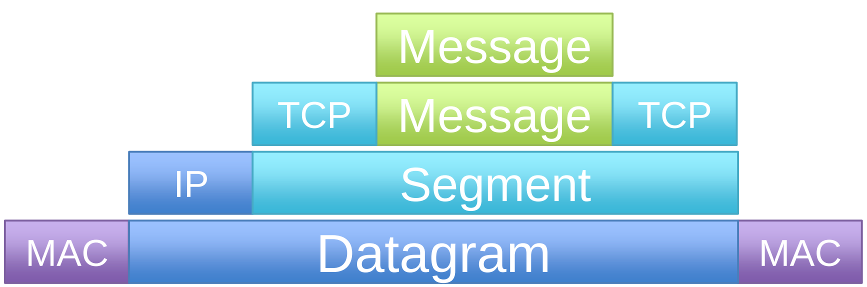
Layers – Example & Live Demo

Interfaces

```
interface ITransportLayer
{
    void SendMessage(string message);
    string ReceiveMessage();
    void SetLowerLayer(IInternetLayer layer);
}

interface IInternetLayer
{
    void SendSegment(string segment);
    string ReceiveSegment();
    void SetLowerLayer(ILinkLayer layer);
}

interface ILinkLayer
{
    void SendDatagram(string datagram);
    string ReceiveDatagram();
}
```



Implementation

```
class TCP : ITransportLayer
{
    public void SetLowerLayer(IInternetLayer sublayer)
        => _sublayer = sublayer;
    private IInternetLayer _sublayer;

    public void SendMessage(string message)
    {
        var segment = "TCPHeader|" + message + "|TCPFooter";
        _sublayer.SendSegment(segment);
    }

    public string ReceiveMessage()
    {
        var segment = _sublayer.ReceiveSegment();
        var message = segment.Substring(10, segment.Length - 20);
        return message;
    }
}
```

Usage

```
static void Main(string[] args)
{
    ITransportLayer transport = new TCP();
    IInternetLayer internet = new IP();
    ILinkLayer link = new MAC();
    internet.SetLowerLayer(link);
    transport.SetLowerLayer(internet);

    var message = "Hello Design Patterns!";
    transport.SendMessage(message);

    var received = transport.ReceiveMessage();
    Console.WriteLine("Application received: " + {received});
}
```

Layers – Implementation Issues

- Who composes the layers at runtime?
- How are the interfaces defined?
- Layers are Black Boxes
- Workarounds / Skip layers?
- Stateless / Stateful Implementations?
- Caches
- Split up Calls vs. Combine calls (Optimization?)
- Exception-Handling throughout layers?

Learning Goals for Today

- Understand and describe **SOLID Principles:**
  - Single Responsibility
  - Open Closed Principle
  - Liskov Substitution
  - Interface Segregation
  - Dependency Inversion
- Understand and describe **Principles of Good Programming**:
  - Decomposition
  - Abstraction
  - Decoupling
  - Usability & Simplicity
- Understand and describe some design patterns:
  - Layers
  - Iterator
  - Observer
- Derive the Problem, Forces, Solution, and Consequences of these patterns

SOLID Principles (in OOP)

- ***S*ingle Responsibility**: A class should have one, and **only one**, **reason to change**.
- ***O*pen Closed**: You should be able to **extend** a class's behavior, **without modifying** it.
- ***L*iskov Substitution**: Derived classes must be **substitutable** for their base **classes**.
- ***I*nterface Segregation**: Make **fine grained interfaces** that are client specific.
- ***D*ependency Inversion**: **Depend on abstractions**, not on concrete implementations.

Principles of Good Programming

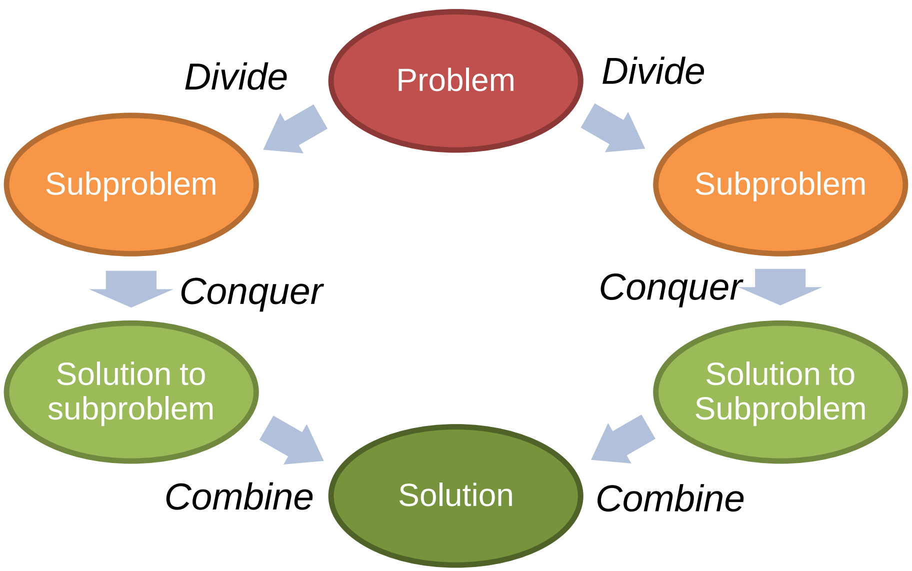
- **Decomposition**
  make a problem manageable
  decompose it into sub-problems
- **Abstraction**
  wrap around a problem
  abstract away the details
- **Decoupling**
  reduce dependencies, late binding
  shift binding time to “later”
- **Usability & Simplicity**
  make things easy to use right, hard to use wrong
  adhere to expectations, make usage intuitive

Principles of Good Programming

- **Decomposition** (make a problem manageable):
  - Layers, Pipes and Filters, Model-View-Controller
- **Abstraction** (wrap around):
  - Adapter, Decorator, Façade, Proxy
- **Decoupling** (reduce dependencies, late binding):
  - Observer, Mediator, Broker, Factory Method
- **Usability & Simplicity** (easy to use right, hard to use wrong):
  - Monitor, Thread-Specific Storage, Counted Pointer

Decomposition

- Split up a problem until it gets manageable
- Divide and Conquer
- Separation of Concerns
- Orthogonality (Separation of Concepts)
- Single responsibility
- Curly's Law (do just one thing and stick to that)



Abstraction

- Hide implementation details
- Wrap another layer around a problem.
- Liskov substitution
  Substitute Parent-Classes by Sub-Classes
- Fundamental theorem of software engineering:
  "We can solve any problem by introducing an extra level of indirection."
  (David Wheeler)

Decoupling

- Minimise coupling / Maximize cohesion
- Separation of Concerns
- Shift Binding time to “later”
- Composition over inheritance
- Inversion of control
  - Hollywood principle:
    “*Don’t call us, we call you!*”

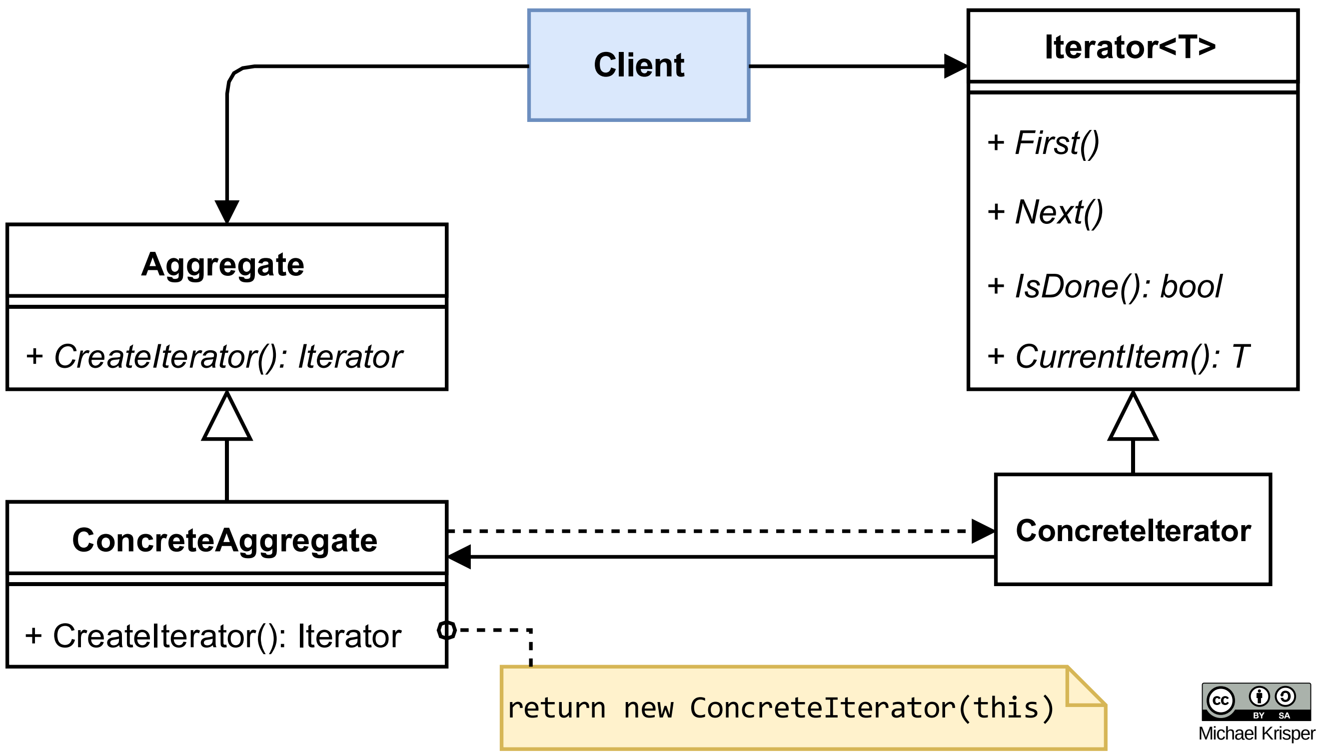
- Open close - encapsulate what changes
- Embrace change
- Law of Demeter: Only use *“direct”* dependencies

Usability & Simplicity

- YAGNI – You ain't gonna need it!
- DRY – Don't repeat yourself!
- Principle of least astonishment
  - Don't make me think
  - Easy to use right, hard to use wrong
  - Code for the Maintainer
  - Command / Query Separation
  - Interface segregation
- Ockham's razor
  - Do the simplest thing possible
  - KISS – Keep it simple, stupid!
- Avoid premature optimization (Knuth, 1974)

Iterator

Retrieve items of a collection element by element.



Iterator

Context: Access elements of a collection

Problem:

- Different collections have different means of retrieving its elements.

Forces:

- Access the elements without exposing its internal representation
- Provide uniform interface for traversing different structures
- Support multiple traversals modes
- Sometimes we want to “hide” traversal from the client (foreach)

Solution:

- Define an interface for repeatedly accessing the “next” element of an aggregate
- Implement a concrete iterator for each needed type of aggregate

Consequences:

+ Every collection can be accessed in a uniform way.
+ Multiple iterations are possible at the same time.
+ Traversal algorithm can vary
- Lower efficiency
- Robustness is not guaranteed (insertions, deletions)
- “Hides” underlying data structure

Iterator – Known Uses

- Many programming languages use iterators for looping over collections (C++, C#, Java, Python, …)
- Foreach Loop also uses iterator
- Enumerables & Generators are also variants of iterators

```
int myint[] = {1, 2, 3, 4, 5};
for (int i : myint)
  std::cout << i << std::endl;
```

foreach uses implicit iteration

C++ uses “end”-pointer to test the end of iteration

```
vector<int> ar = { 1, 2, 3, 4,     };
vector<int>::iterator ptr;
for (ptr = ar.begin(); ptr < ar.end(); ptr++)
   cout << *ptr << std::endl;
```

Iterator – Known Uses

Python uses Exception instead of “IsDone()” or “End”:

Definition:

```
class Counter:
    def __init__(self, limit):
        self.current = 0
        self.limit = limit

    def __iter__(self):
        return self

    def __next__(self):
        if self.current > self.limit:
            raise StopIteration
        else:
            self.current += 1
            return self.current-1
```

Usage:

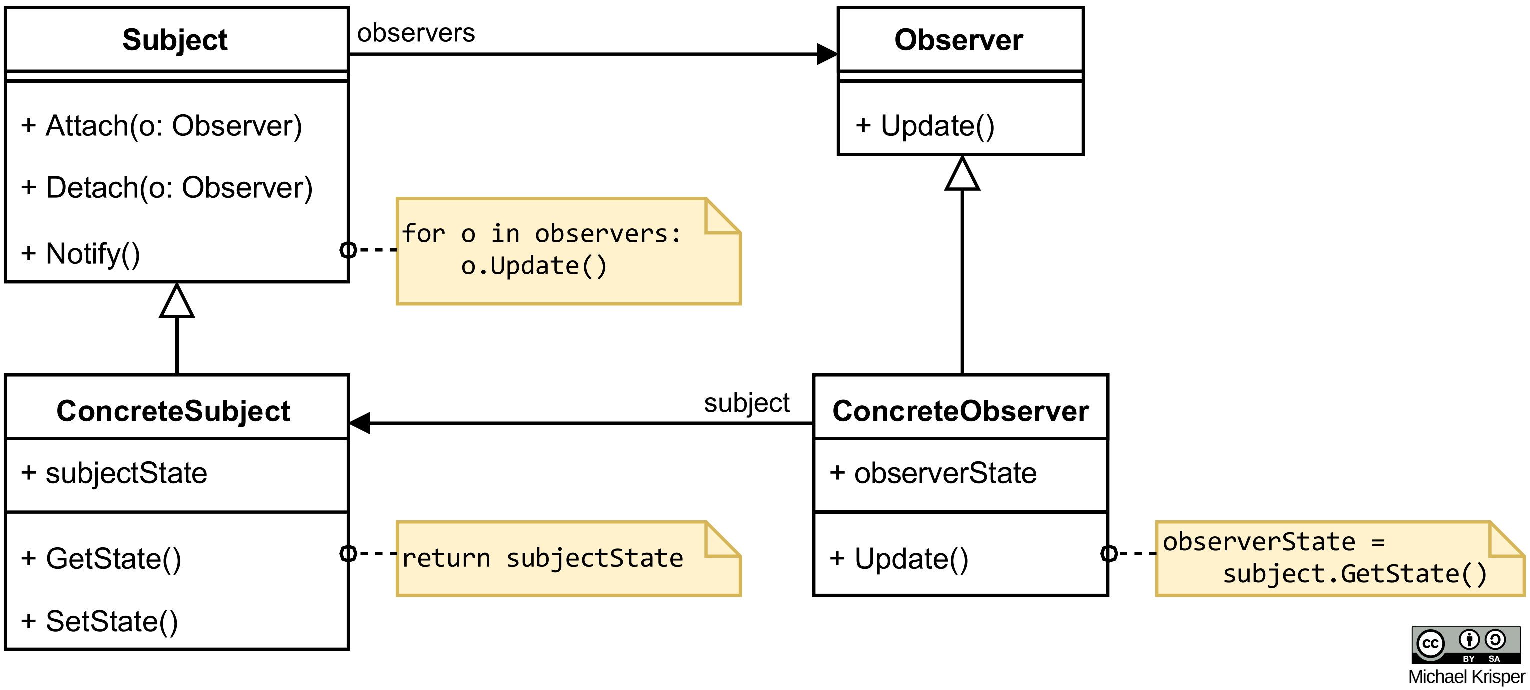
```
for c in Counter(2):
    print c


it = Counter(2)
print(next(it))
print(next(it))
print(next(it))
print(next(it))
```

Python uses StopIteration-Exception to signify the end of iteration

Observer

Inform registered observers about changes.



- Events and Signals in many programming languages and operating systems e.g. Events like OnClick, OnEnter, OnKeyDown in C#, Java, JavaScript, …
- Message Queue Systems: MQTT, Apache Kafka, RabbitMQ

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?

Summary

SOLID
- Single Responsibility
- Open Closed Principle
- Liskov Substitution
- Interface Segregation
- Dependency Inversion

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

Patterns:
- Layers
- Iterator
- Observer