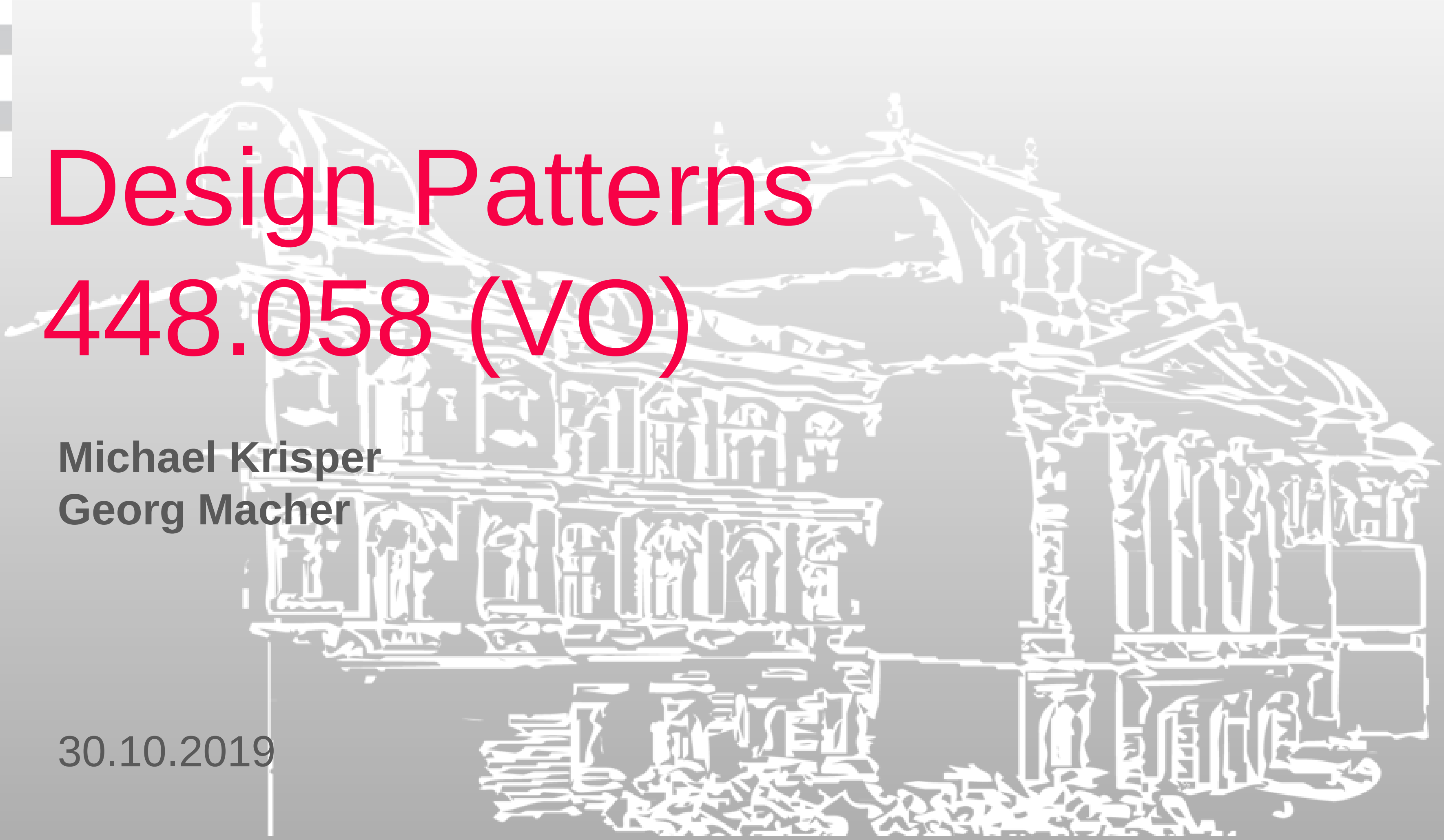


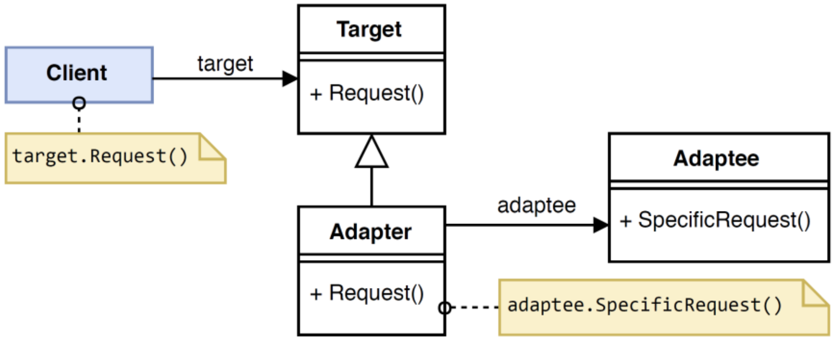
Design Patterns 448.058 (VO)

Michael Krisper
Georg Macher

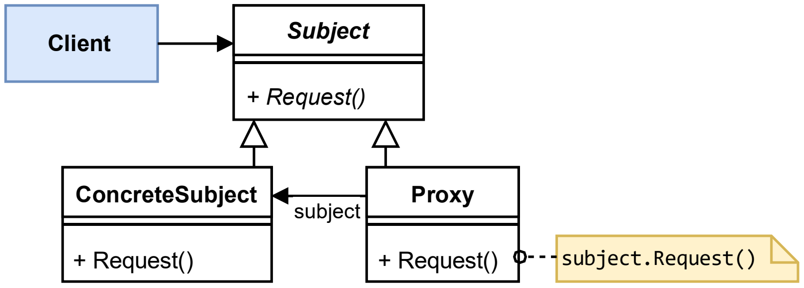
30.10.2019

Revision from last time… wrappers

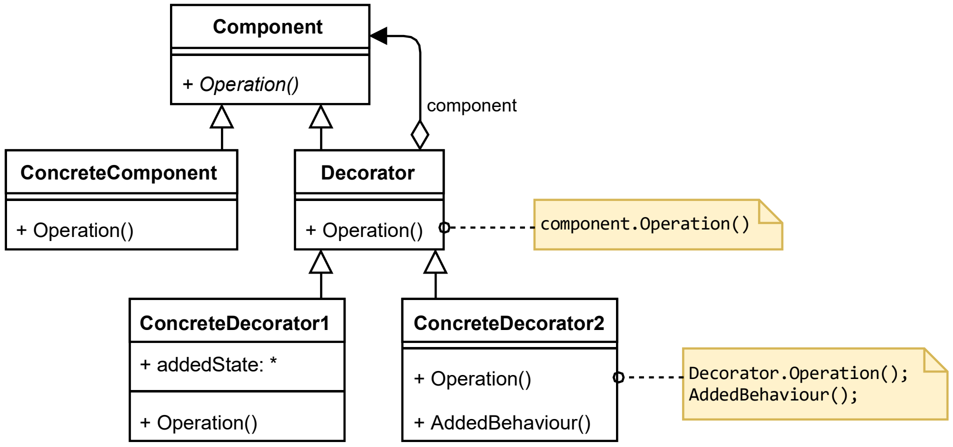
Adapter:
Make object compatible.



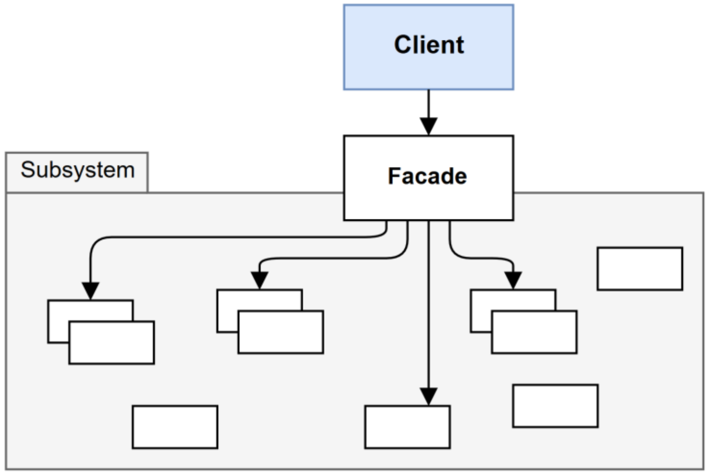
Proxy:
Encapsulate access to objects.



Decorator:
Extend functionality.

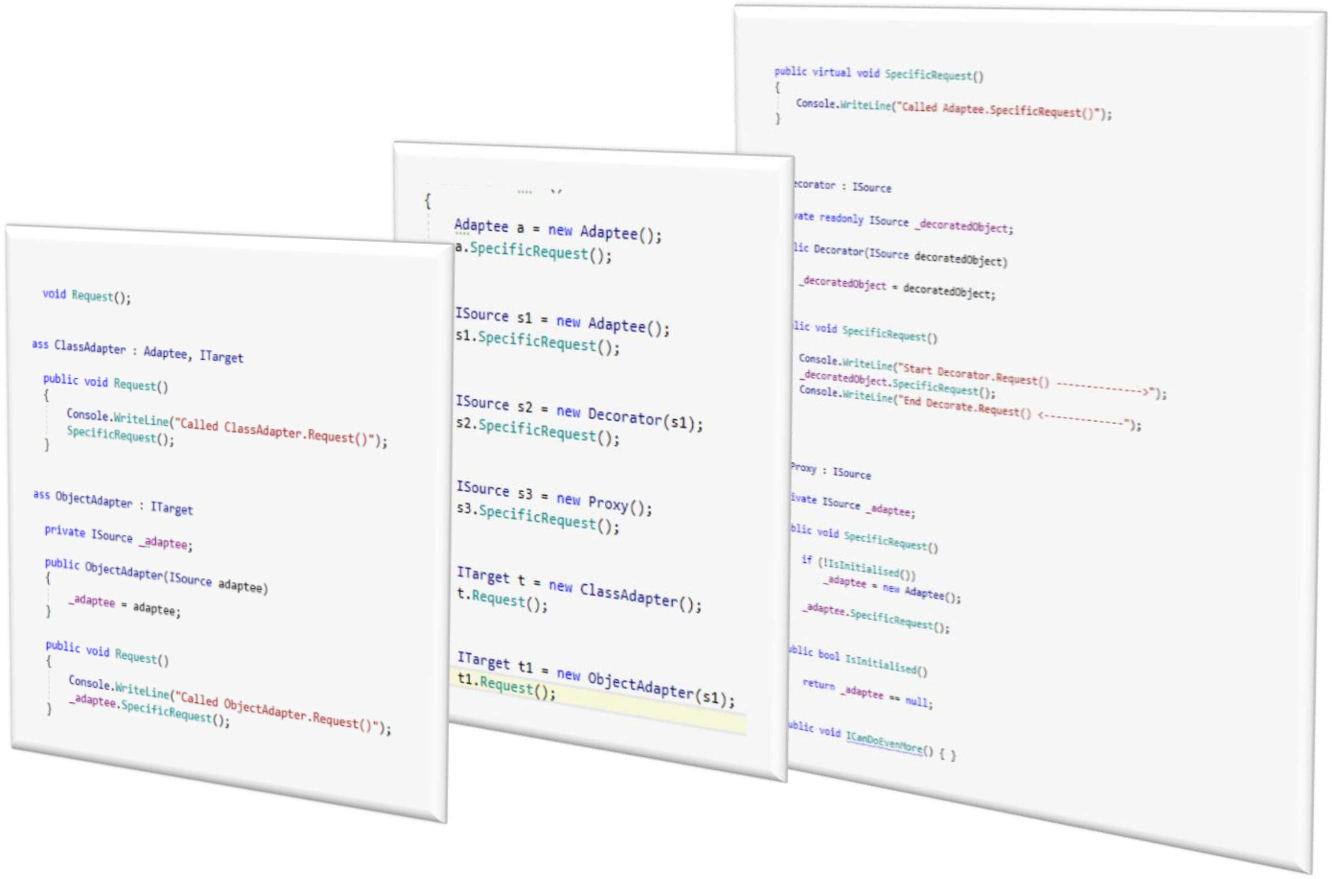


Façade:
Create combined interface.



Revision from last time…

Live Programming Demo…



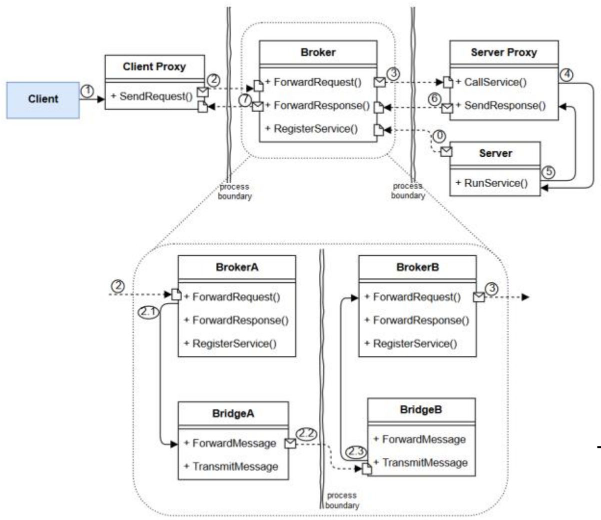
```
void Request();

ass ClassAdapter : Adaptee, ITarget

  public void Request()
  {
      Console.WriteLine("Called ClassAdapter.Request()");
      SpecificRequest();
  }


ass ObjectAdapter : ITarget

  private ISource _adaptee;

  public ObjectAdapter(ISource adaptee)
  {
      _adaptee = adaptee;
  }

  public void Request()
  {
      Console.WriteLine("Called ObjectAdapter.Request()");
      _adaptee.SpecificRequest();
  }
```

```
{
    Adaptee a = new Adaptee();
    a.SpecificRequest();

    ISource s1 = new Adaptee();
    s1.SpecificRequest();

    ISource s2 = new Decorator(s1);
    s2.SpecificRequest();

    ISource s3 = new Proxy();
    s3.SpecificRequest();

    ITarget t = new ClassAdapter();
    t.Request();

    ITarget t1 = new ObjectAdapter(s1);
    t1.Request();
```

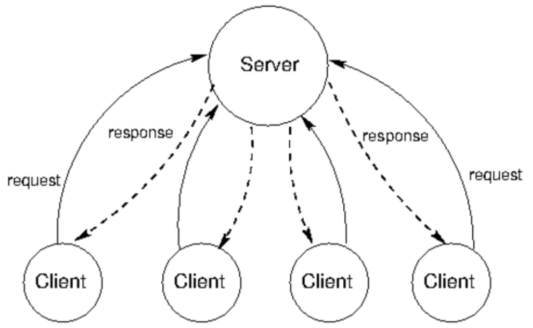
```
public virtual void SpecificRequest()
{
    Console.WriteLine("Called Adaptee.SpecificRequest()");
}


ecorator : ISource

vate readonly ISource _decoratedObject;

lic Decorator(ISource decoratedObject)

 _decoratedObject = decoratedObject;

lic void SpecificRequest()

 Console.WriteLine("Start Decorator.Request() -------------->");
 _decoratedObject.SpecificRequest();
 Console.WriteLine("End Decorate.Request() <--------------");


Proxy : ISource

ivate ISource _adaptee;

blic void SpecificRequest()

  if (!IsInitialised())
      _adaptee = new Adaptee();

  _adaptee.SpecificRequest();

blic bool IsInitialised()

  return _adaptee == null;

blic void ICanDoEvenMore() { }
```

Revision from last time… architectural

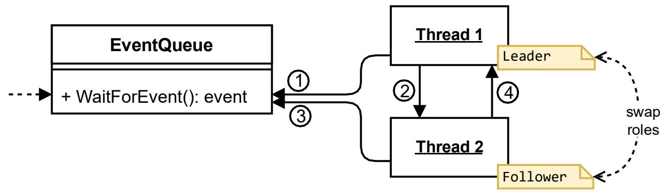
Broker



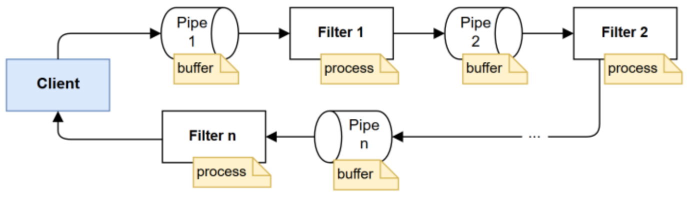
Client-Server



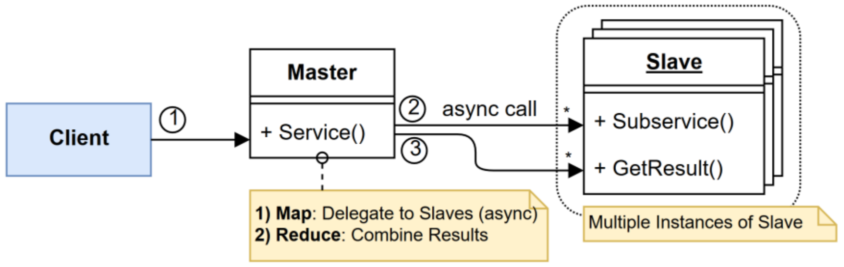
Leader-Follower



Pipes & Filters



Master-Slave


Client
①
Master
+ Service()
② async call
③
*
*
Slave
+ Subservice()
+ GetResult()
1) Map: Delegate to Slaves (async)
2) Reduce: Combine Results
Multiple Instances of Slave

Learning Goals for Today

- MVC / MVP / MVVM / PAC

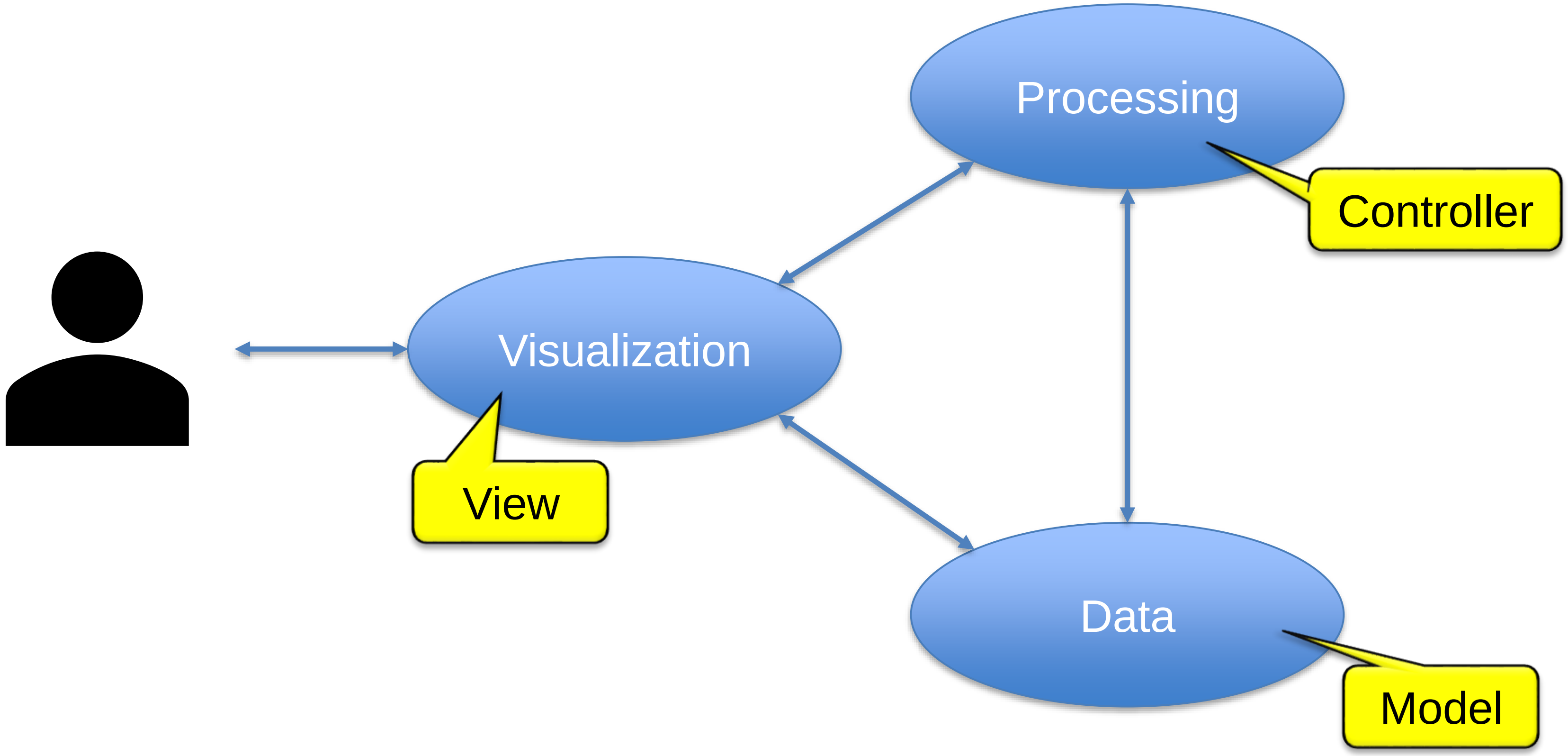
- Understand and describe Creational Patterns:
  - Factory Method
  - Abstract Factory
  - Builder
  - Prototype
  - Singleton

- Understand and describe basic ideas of the following patterns:
  - Memento
  - Flyweight
  - Pooling & Caching

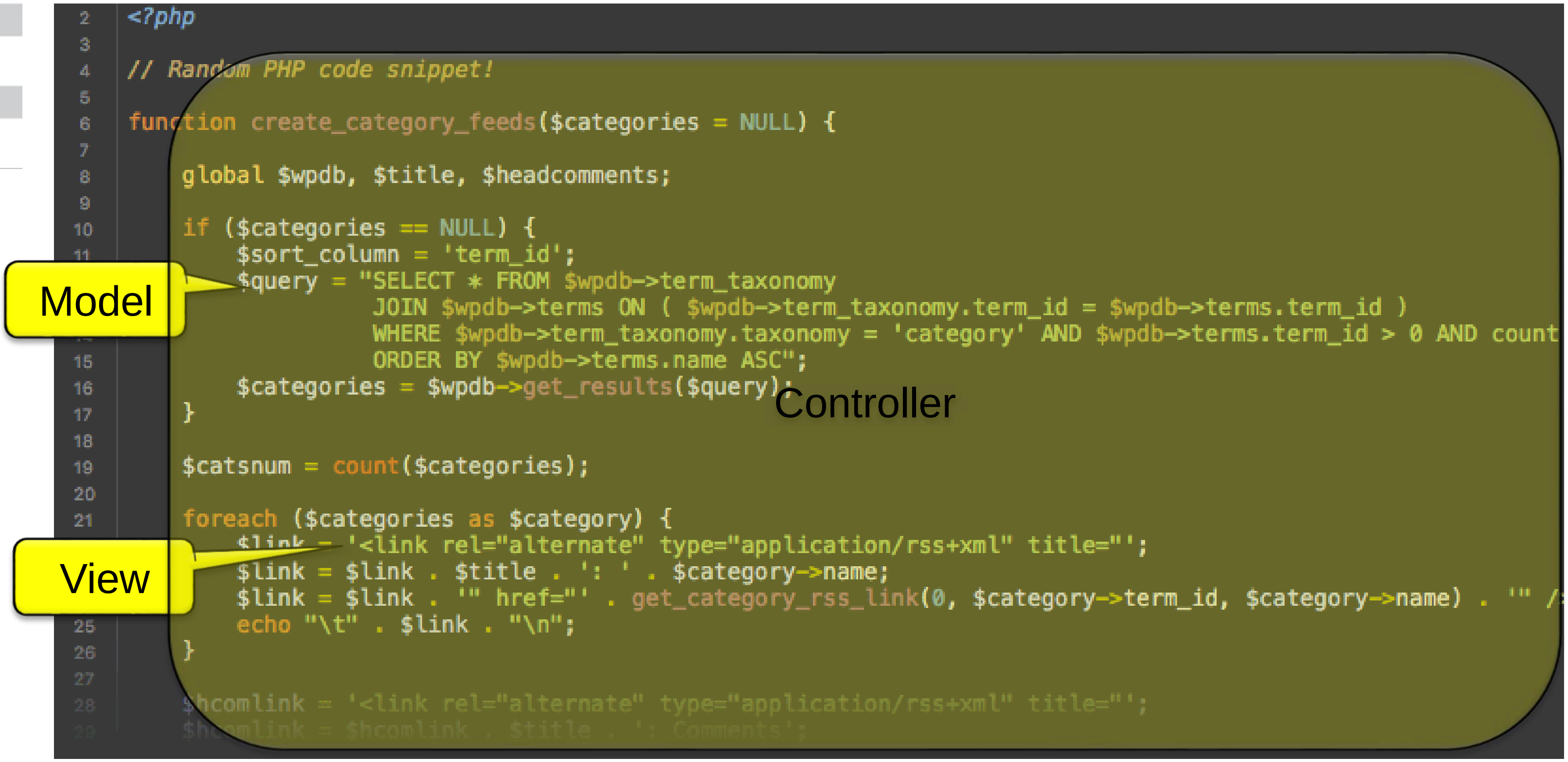
- Explain idea behind “classes at runtime” in dynamic script-languages

Model-View-Controller (MVC) / Model-View-Presenter (MVP) / Model-View-Viewmodel (MVVM)

Separate the responsibilities of visualizing, processing and data management for GUI applications.



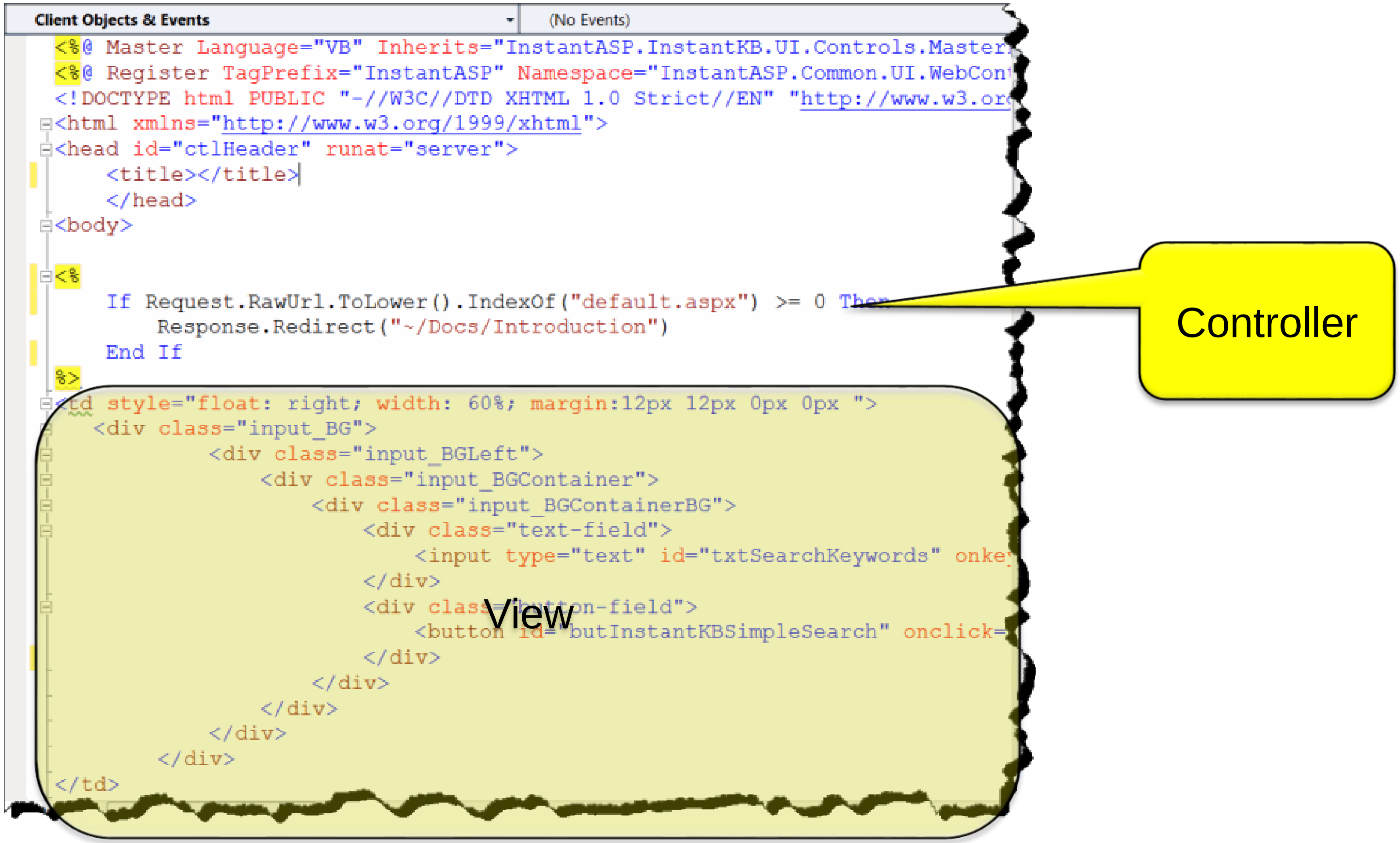
Problem?



```php
<?php

// Random PHP code snippet!

function create_category_feeds($categories = NULL) {

    global $wpdb, $title, $headcomments;

    if ($categories == NULL) {
        $sort_column = 'term_id';
        $query = "SELECT * FROM $wpdb->term_taxonomy
                 JOIN $wpdb->terms ON ( $wpdb->term_taxonomy.term_id = $wpdb->terms.term_id )
                 WHERE $wpdb->term_taxonomy.taxonomy = 'category' AND $wpdb->terms.term_id > 0 AND count
                 ORDER BY $wpdb->terms.name ASC";
        $categories = $wpdb->get_results($query);
    }

    $catsnum = count($categories);

    foreach ($categories as $category) {
        $link = '<link rel="alternate" type="application/rss+xml" title="';
        $link = $link . $title . ': ' . $category->name;
        $link = $link . '" href="' . get_category_rss_link(0, $category->term_id, $category->name) . '" /
        echo "\t" . $link . "\n";
    }

    $hcomlink = '<link rel="alternate" type="application/rss+xml" title="';
    $hcomlink = $hcomlink . $title . ': Comments';
```

⇨ Completely mixed Responsibilities. Fully coupled. Bad.

Problem?



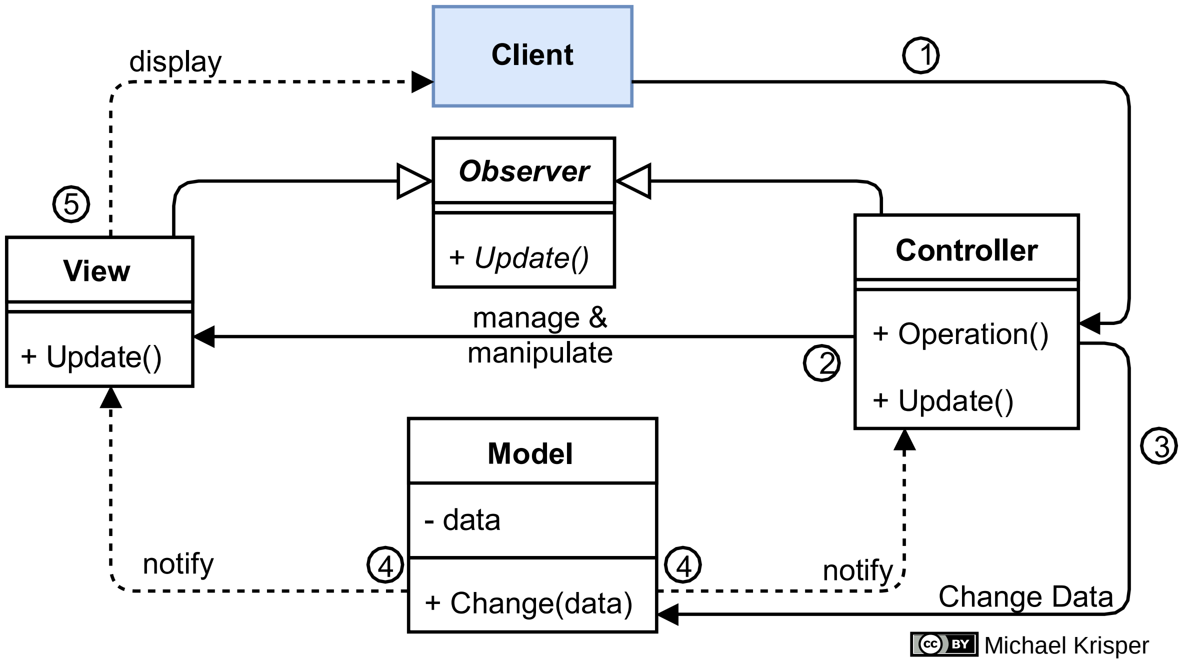
Client Objects & Events (No Events)

```
<%@ Master Language="VB" Inherits="InstantASP.InstantKB.UI.Controls.Master
<%@ Register TagPrefix="InstantASP" Namespace="InstantASP.Common.UI.WebCon
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.or
<html xmlns="http://www.w3.org/1999/xhtml">
<head id="ctlHeader" runat="server">
    <title></title>
    </head>
<body>

<%
    If Request.RawUrl.ToLower().IndexOf("default.aspx") >= 0 Then
        Response.Redirect("~/Docs/Introduction")
    End If
%>
<td style="float: right; width: 60%; margin:12px 12px 0px 0px ">
   <div class="input_BG">
            <div class="input_BGLeft">
                <div class="input_BGContainer">
                    <div class="input_BGContainerBG">
                        <div class="text-field">
                            <input type="text" id="txtSearchKeywords" onke
                        </div>
                        <div class="button-field">
                            <button id="butInstantKBSimpleSearch" onclick=
                        </div>
                    </div>
                </div>
            </div>
        </div>
</td>
```

⇨ Principle of least surprise broken. You never know what is implemented in GUI code.

Model-View-Controller (MVC)

Separate the responsibilities of visualizing, processing and data management for GUI applications.



MVC / MVP / MVVM
Some Variations…

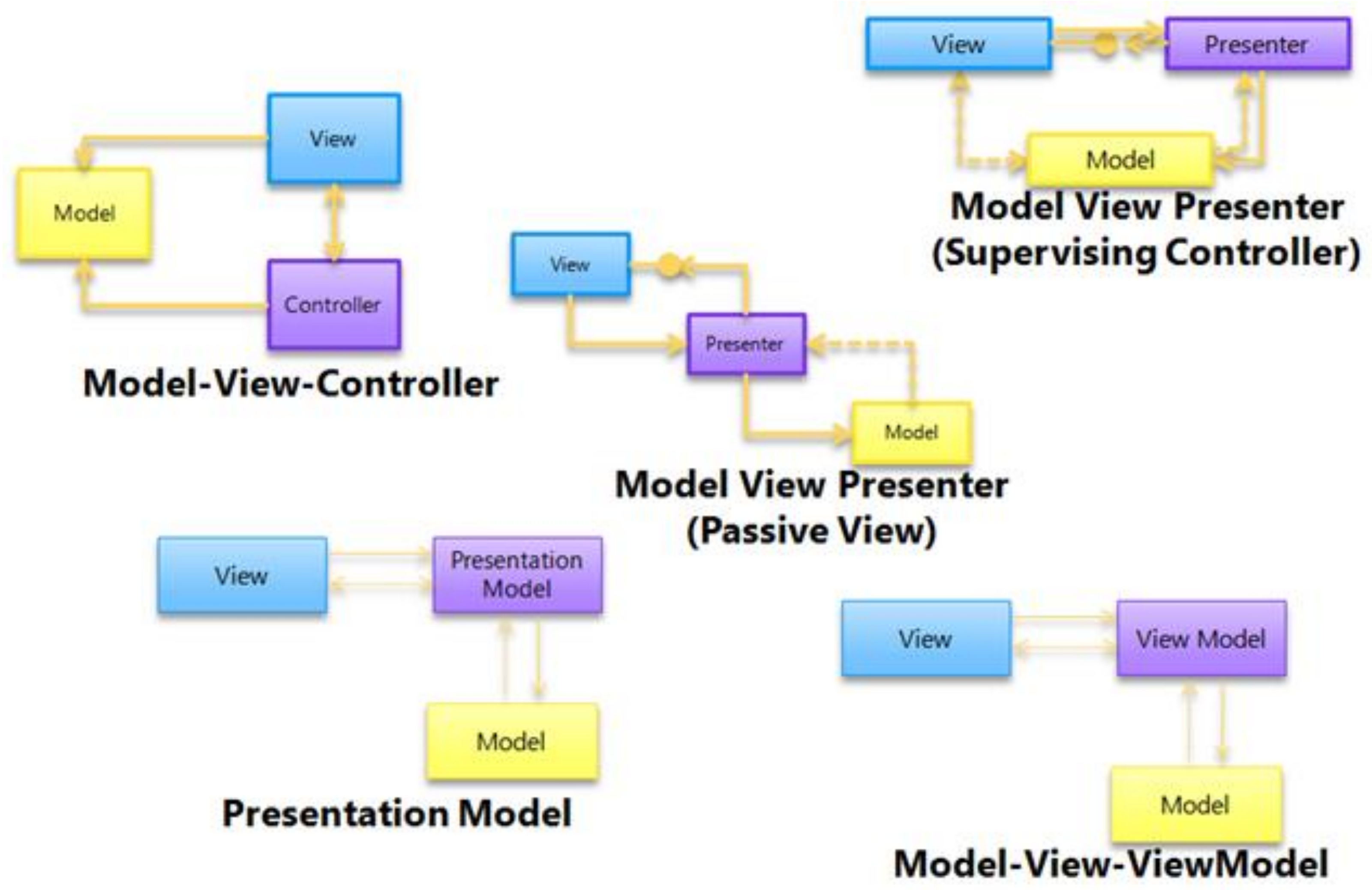


Figure by Erwin Van der Valk, 2009, https://blogs.msdn.microsoft.com/erwinvandervalk/2009/08/14/the-difference-between-model-view-viewmodel-and-other-separated-presentation-patterns/

MVC vs. MVP vs. MVVM

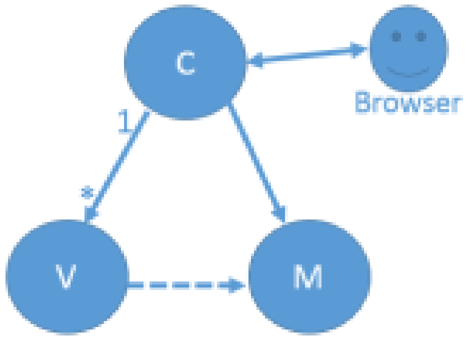
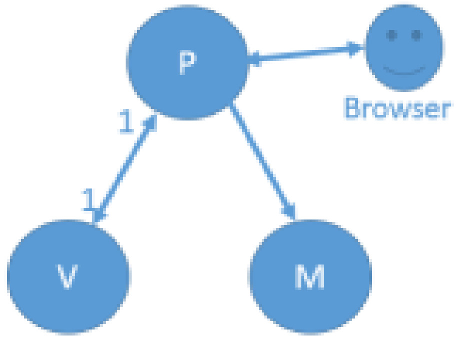
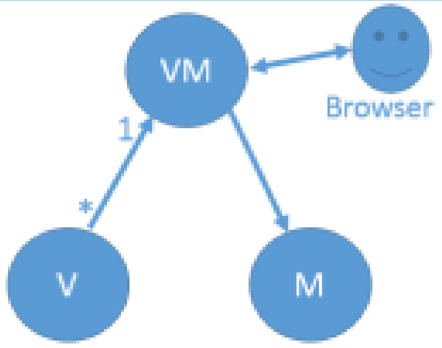
MVC	MVP	MVVM
		
• Controller is the entry point to the application	• View is the entry point to the application	• View is the entry point to the application
• One to Many relationship between Controller and View	• One to One mapping between View and Presenter	• One to Many relationship between View and ViewModel
• View does not have reference to the Controller	• View have the reference to the Presenter	• View have the reference to the View Model
• View is very well aware of the Model	• View is not aware of the Model	• View is not aware of the Model
• Smalltalk, ASP.Net MVC	• Windows forms	• Silverlight, WPF, HTML5 with Knockout/AngularJS

Figure by Arslan Butt, 09.09.2014, http://bestcodeway.blogspot.com/2014/09/mvc-vs-mvvm-vs-mvp.html

MVC / MVP / MVVM

Context: Important dataset that needs to be provided to be processed.

Problem: Tight coupling of data and representation. I want to separate data and representation.

Forces:

- Independent change of data and views
- Separation of concerns
- Different lifecycles / update rates
- Different expertise

Solution:

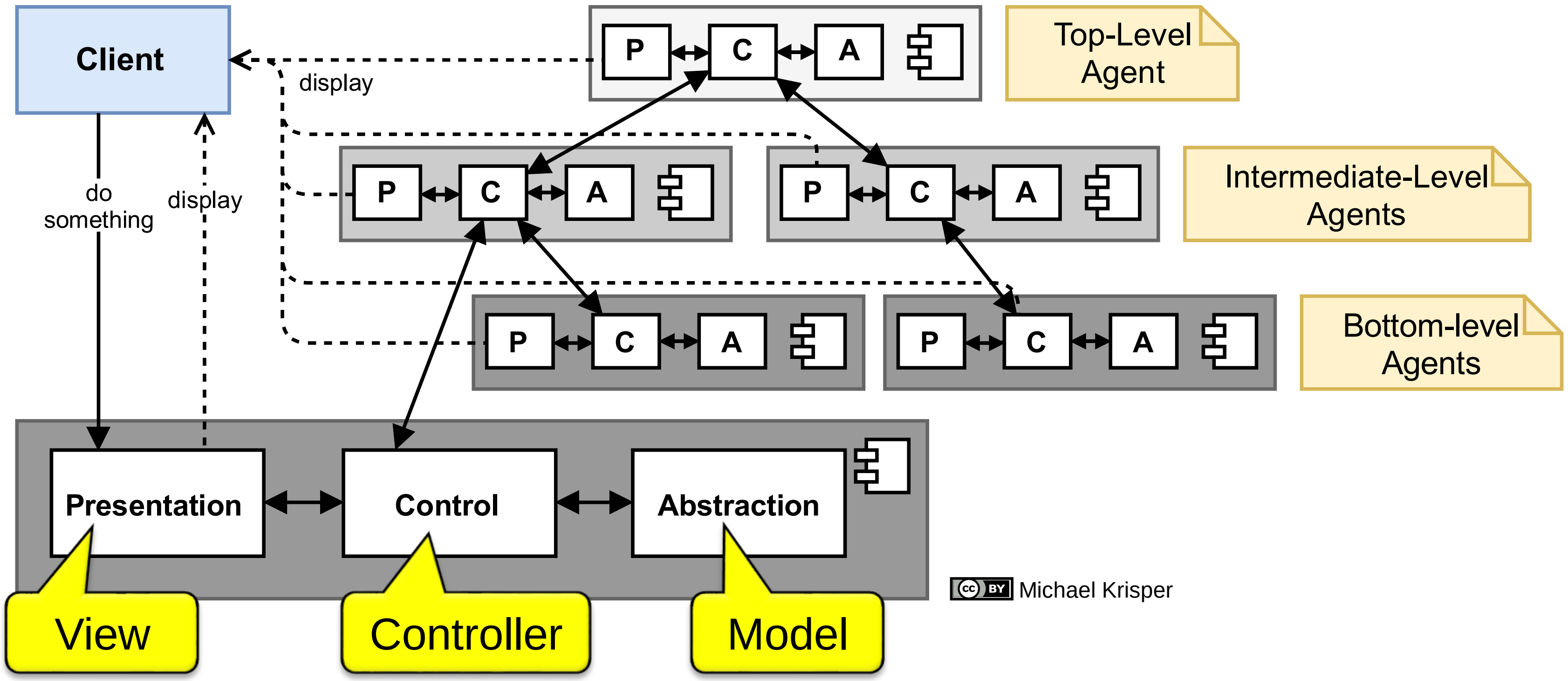
- Decouple components for data, visualisation, and control
- Dedicated part for representation (view)
- Part for manipulation of data (controller)
- Independent model for storage of data (model)

Consequences:

+ Increased reusability of code
+ Separable for different development teams
+ Independence between data and representation (decoupling)
- Complexity increase
- Unit testing more complex

Presentation-Abstraction-Control (PAC)

Decompose GUI generation into smaller agents, each consisting of three parts: presentation, abstraction and control.



Creational Patterns

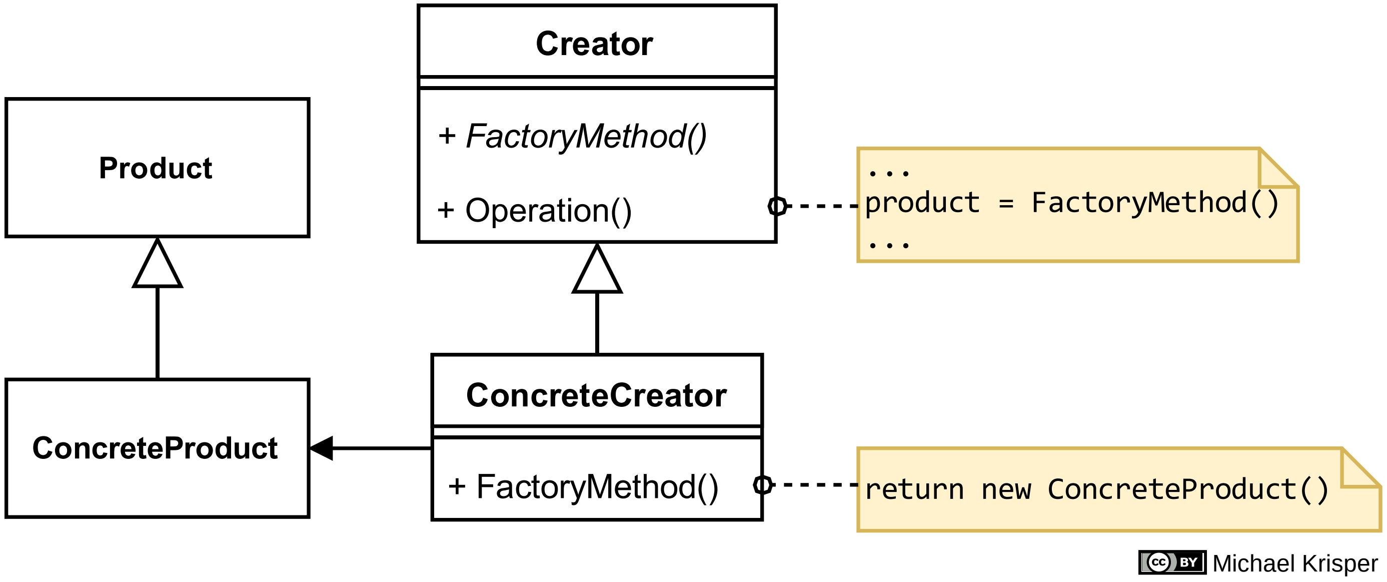
How to create objects in a decoupled and flexible way?

- Who creates the object?
- Dependencies?
- How are parameters set?

If I see a “**new**” in your application code, I kill you!
– Prof. Sven Havemann, Graz University of Technology, 2012

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:

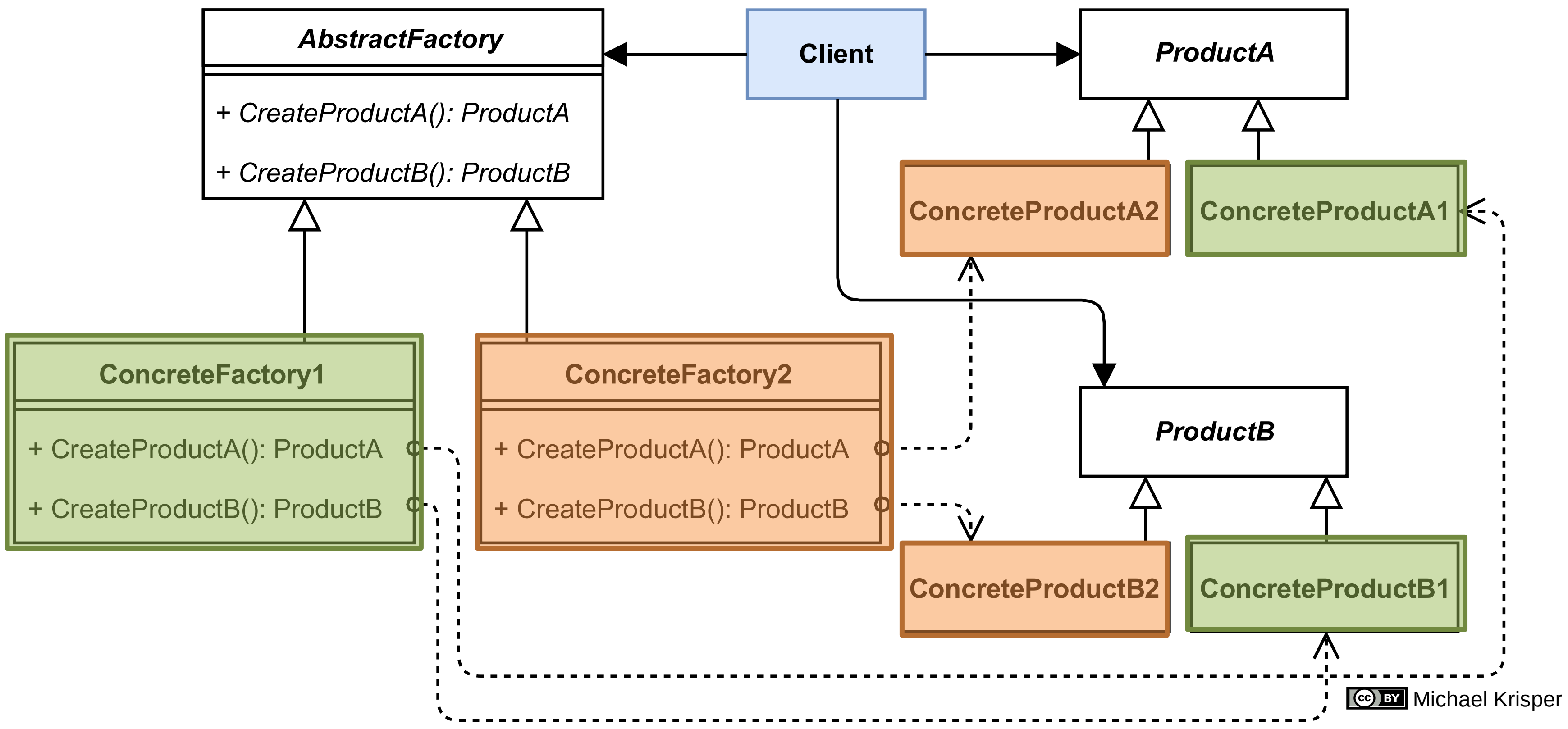
- Define an interface of capabilities your objects must implement.
- Define some means (method or own class) to create the actual object.
- Let the actual object implement the needed interface.

Consequences:

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

Abstract Factory

Create whole families of related objects



AbstractFactory
+ CreateProductA(): ProductA
+ CreateProductB(): ProductB
Client
ProductA
ConcreteProductA2
ConcreteProductA1
ConcreteFactory1
+ CreateProductA(): ProductA
+ CreateProductB(): ProductB
ConcreteFactory2
+ CreateProductA(): ProductA
+ CreateProductB(): ProductB
ProductB
ConcreteProductB2
ConcreteProductB1
Michael Krisper

Abstract Factory

Context:

Having multiple related families of similar objects

Problem:

How to create only matching objects?

Forces:

- Only create objects which fit together
- Choose object family at runtime
- Reveal just the interfaces, not the implementations

Solution:

- Define **Interface** for **Products**.
- Define **Interface** for **Factories**.
- Implement both accordingly.
- **Select the needed factory** at runtime to create the needed products.

Consequences:

+ Makes exchanging product families easy

+ Promotes consistency among products

+ Isolates concrete classes

~ When is the product family selected? Who selects?

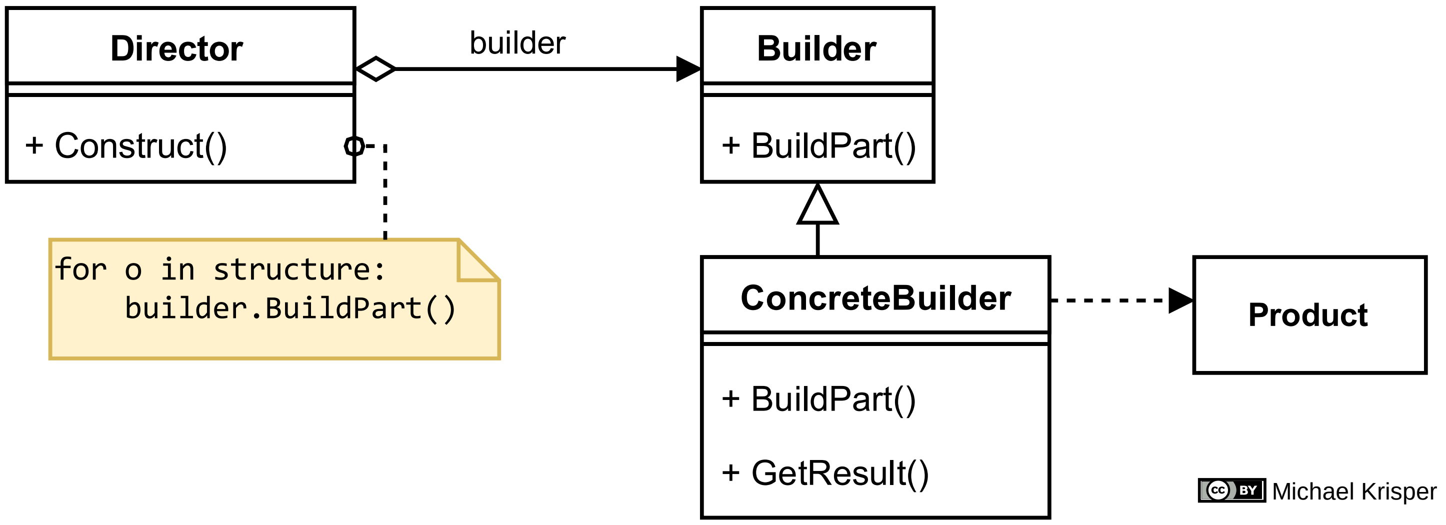
~ Factories as singletons?

~ Use prototypes as templates?

- Supporting new kinds of products is difficult

Builder

Split up creation into multiple steps



```
for o in structure:
    builder.BuildPart()
```

Builder

Context:

Creation of complex objects

Problem:

How to create complex objects in an easy and comfortable way?

Forces:

- Manage many different construction options
- Creation of objects should be independent of assembling

Solution:

- Split creation from assembling
- Define Interface for creating individual parts & assembling
- Implement methods for parts

Consequences:

+ Allows many combinations of parts

+ Isolates code for construction and representation

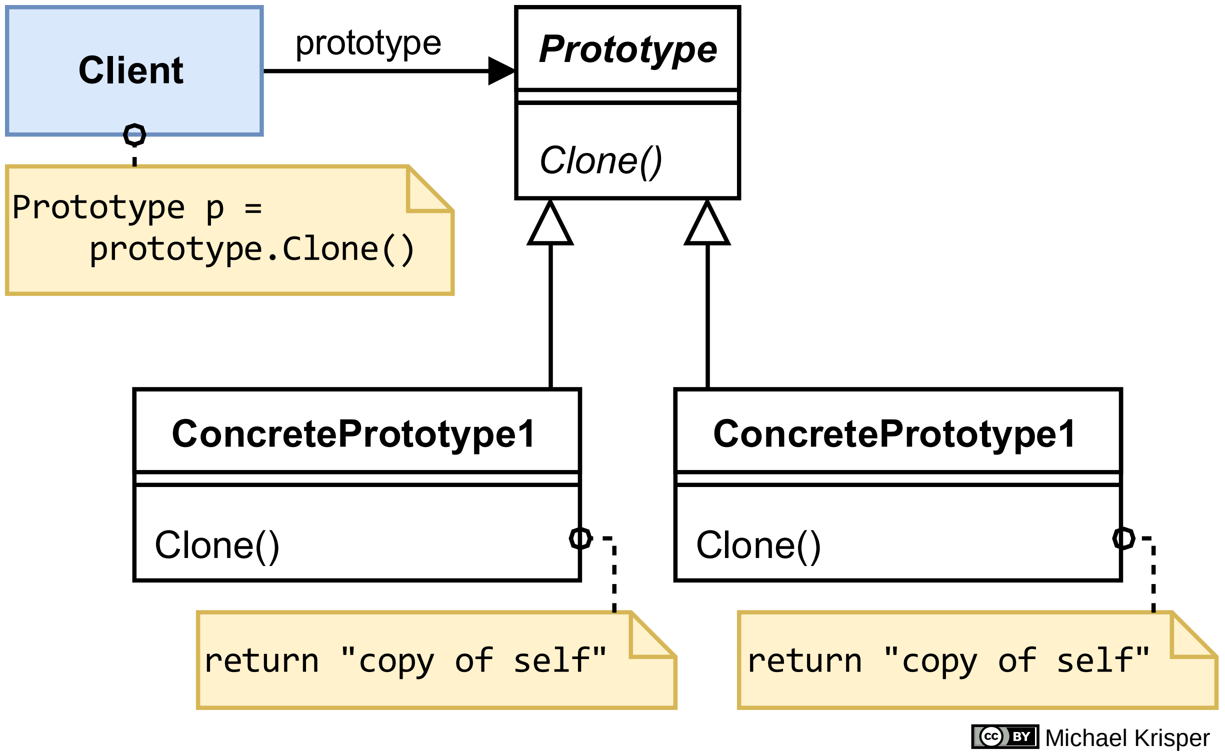
+ Allows finer control of construction

- Construction is not a simple “new” anymore

- How to ensure that parts are correctly configured?

Prototype

Create objects by cloning from templates



Prototype

Context:

Creation of objects whose classes and properties are not known until run-time

Problem:

How to dynamically implement and use objects without knowing its properties?

Forces:

- Object Members are defined at runtime
- Avoid building complex class hierarchies and factories
- Avoid long taking instantiations

Solution:

- Declare cloning interface
- Implement cloning interface
- (Add mechanism for dynamically setting/getting members and calling methods → Dictionary!)

Consequences:

+ Dynamic objects can be created at runtime
+ Class system is bypassed
+ No complex inheritance hierarchy
+ Long taking initialisation are done only once

~ Usage of prototype manager? (registry)

~ Shallow vs deep copy?

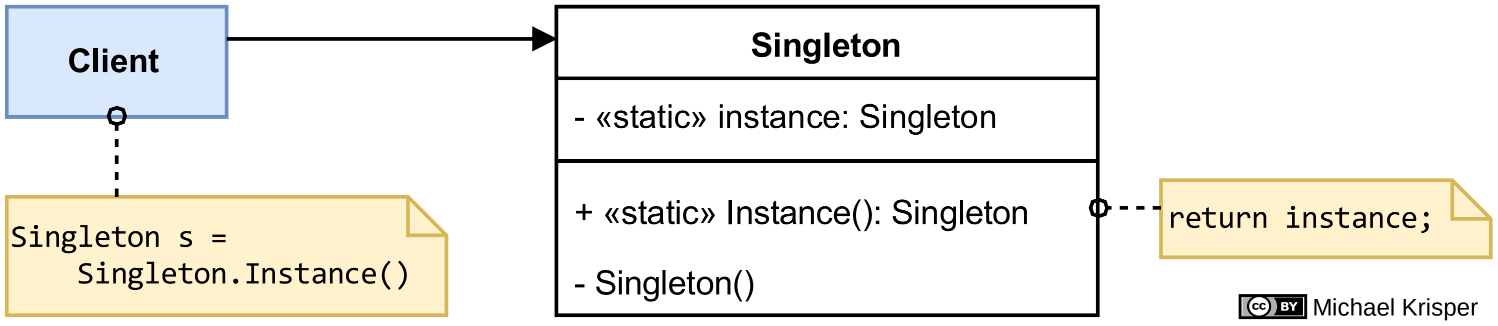
~ How to access members?

- No type safety!

- No compile-time errors!

Singleton

Allow only one instance of an object



Singleton

Context:

Creation of exactly one instance

Problem:

Ensure a class only has one instance, provide a global point of access

Forces:

- There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point
- When the sole instance should be extensible by subclassing, clients should be able to use and extended instance without modifying their code

Solution:

- Hide the constructor of a class (protected or private)
- Add a static Factory Method to create exactly one instance stored as static member
- Consequent creations only return the already created instance.
- Prohibit deep copying of the object

Consequences:

- Controlled access to sole instance
- Reduced name space
- Permits refinement of operations and representation (subclassing)
- Permits a variable number of instances
- More flexible than static class operations

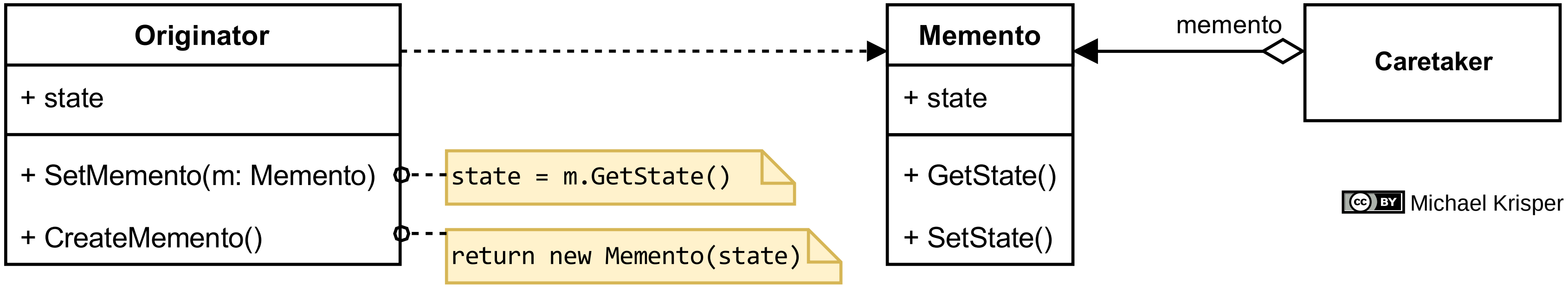
Singleton Example

```
class Singleton
{
    private static readonly Singleton _instance = new Singleton();

    protected Singleton() { }

    public static Singleton Instance()
    {
        return _instance;
    }
}
```

```
void Main()
{
    var s1 = Singleton.Instance();
    var s2 = Singleton.Instance();
    Console.WriteLine($"Singletons are equal: {s1.Equals(s2)}");
}
```

Memento

Store & Load the internal state of an object



```
state = m.GetState()
```

```
return new Memento(state)
```

Problem

How can an object be persisted?

Forces

- State of object should be storable/restorable.
- Do not break encapsulation

Solution:

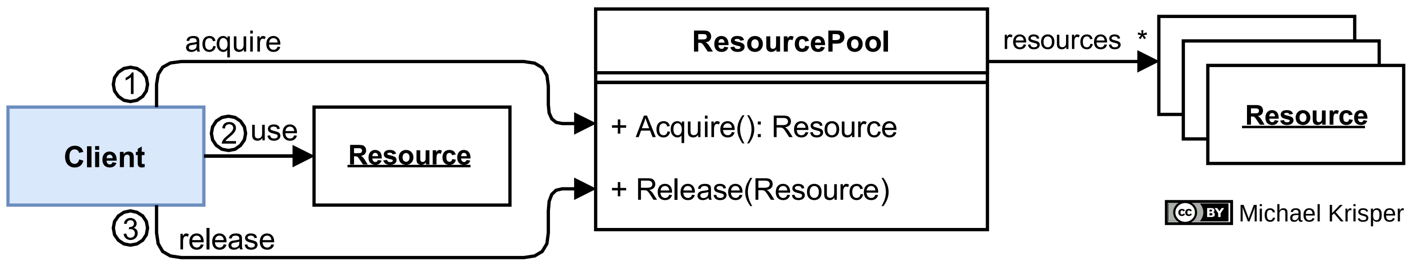
- Create a Memento-Class: Data class for storing the state.
- Implement method for returning a Memento.
- Implement method for reading a Memento.

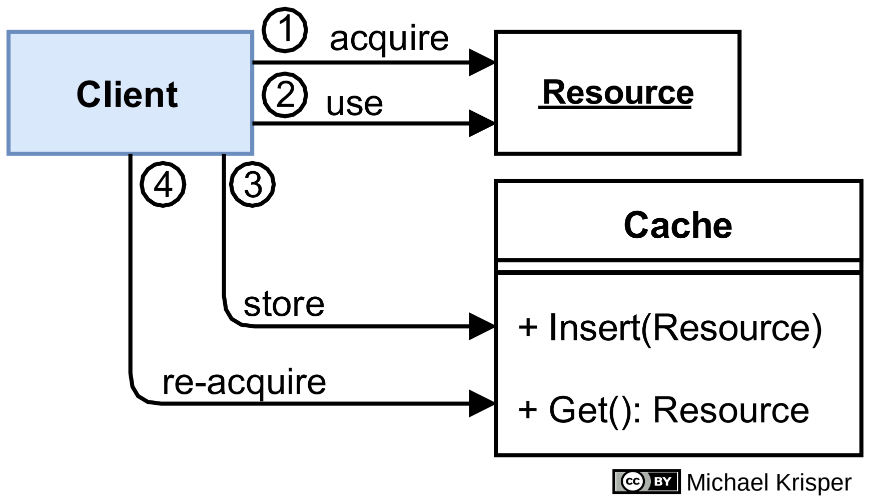
Consequences:

+ State can be persisted without exposing all internal members.
+ Persisted state can be used to restore the object.
+ Snapshots are possible.
+ Combines very well with Command Pattern
- If data format is known, data could be manipulated “offline”. (make sure to add some checksum or digitally sign the memento)

Pooling & Caching

Reuse resources for “later”

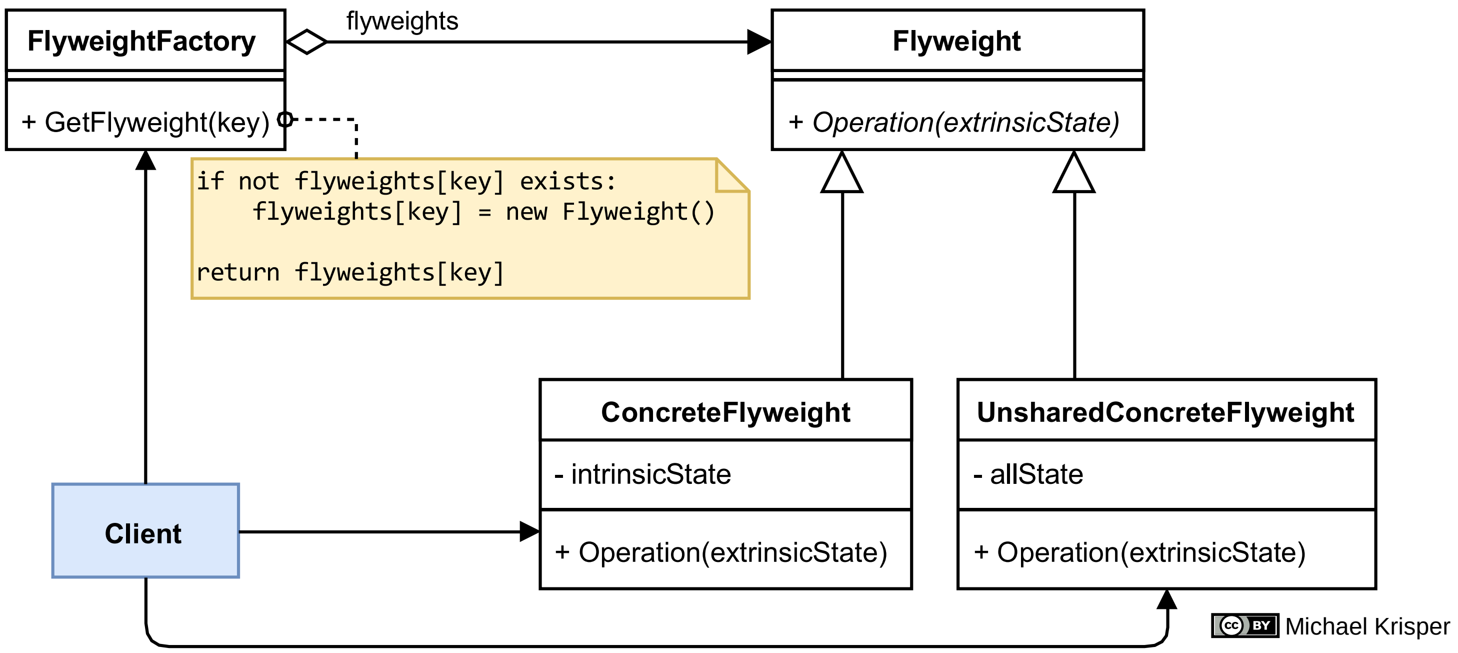




What could the problem, solution, and consequences be?
Take a few minutes thinking time
Finish with Discussion

Flyweight

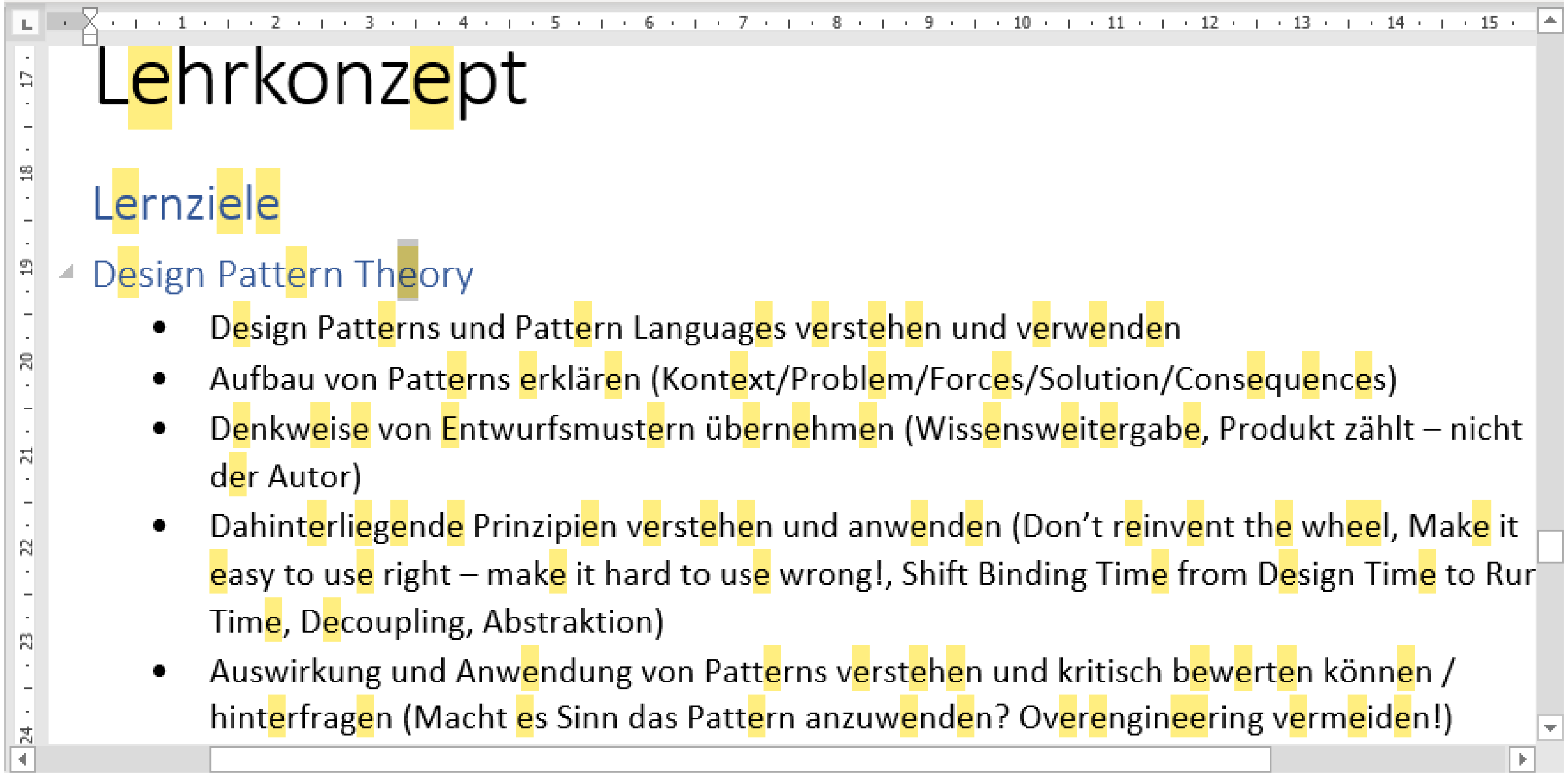
Share global state and vary differences only when needed.



```
if not flyweights[key] exists:
    flyweights[key] = new Flyweight()

return flyweights[key]
```

What could the problem, solution, and consequences be?
Take a few minutes thinking time
Finish with Discussion

Flyweight - Example



Lehrkonzept

Lernziele

Design Pattern Theory

- Design Patterns und Pattern Languages verstehen und verwenden
- Aufbau von Patterns erklären (Kontext/Problem/Forces/Solution/Consequences)
- Denkweise von Entwurfsmustern übernehmen (Wissensweitergabe, Produkt zählt – nicht der Autor)
- Dahinterliegende Prinzipien verstehen und anwenden (Don't reinvent the wheel, Make it easy to use right – make it hard to use wrong!, Shift Binding Time from Design Time to Rur Time, Decoupling, Abstraktion)
- Auswirkung und Anwendung von Patterns verstehen und kritisch bewerten können / hinterfragen (Macht es Sinn das Pattern anzuwenden? Overengineering vermeiden!)

Summary

Patterns:

- Factory Method
- Abstract Factory
- Builder
- Prototype
- Singleton
- Memento
- Flyweight
- Pooling & Caching