M5: Introduction to Software Patterns

Table of Content

- Software patterns – what does it mean?
- Model-View-Controller Pattern
- Remoting Patterns – Synchronous vs. Asynchronous Communication
- Other useful Design Patterns for Web Applications
  - Adapter
  - Facade
  - Factory Method
  - Abstract Factory
  - Strategy
  - State
  - …

Software Patterns

What is a Pattern?

- *Patterns are reusable solutions to recurring problems*
- A Pattern is a solution to a problem in a context
- A Pattern always has a name
  - Shared vocabulary used by many developers (eases communication)
  - Easy to understand foreign code by naming conventions
- A Pattern is language independent
- A Pattern must be defined as useful by others (Community)
- A Pattern has consequences!

Software Patterns

Description of patterns

- Patterns are usually described in a format that includes
  - a **common name**
  - a **description of the problem** including a concrete example and a specific solution to the concrete problem
  - a **summary** that leads to a general solution
  - a **general solution**
  - **consequences**
  - **related patterns**

- Additional parts used in Gamma and Buschmann
  - **implementation**
  - **sample code**
  - **known uses**
  - **also known as (synonyms)**

Software Patterns

Why Patterns?

- Shared vocabulary
  - Say more with less
- Focus on the problem, not on the implementation
- Makes refactoring easier
- Help developing frameworks
- Help to use/connect to foreign implementations
- Help to predict the quality/properties of an application

Software Patterns

Categories of Patterns

- Architectural patterns
  - Model-View-Controller
  - Three-Tier/Two-Tier
  - Pipes and Filters vs. Blackboard
  - ...
- Design patterns
  - Creational
  - Structural
  - Behavioral
  - Concurrency
- Anti patterns
  - Bad solutions for recurring problems

- Not only software related
  - Analysis, Organizational, Pedagogical, …

Architectural Patterns

- Model-View-Controller Pattern (MVC)

- Context

  Application presents content to users in numerous pages containing various data. Also, the engineering team responsible for designing, implementing, and maintaining the application is composed of individuals with different skill sets.

- Problem

  The application needs to support multiple types of users with multiple types of interfaces. E.g. an online store requires an HTML front for web customers, a WML front for wireless customers, a Swing interface for administrators and an XML-based web service for suppliers.

  No redundant code for each type of client interface should be written.

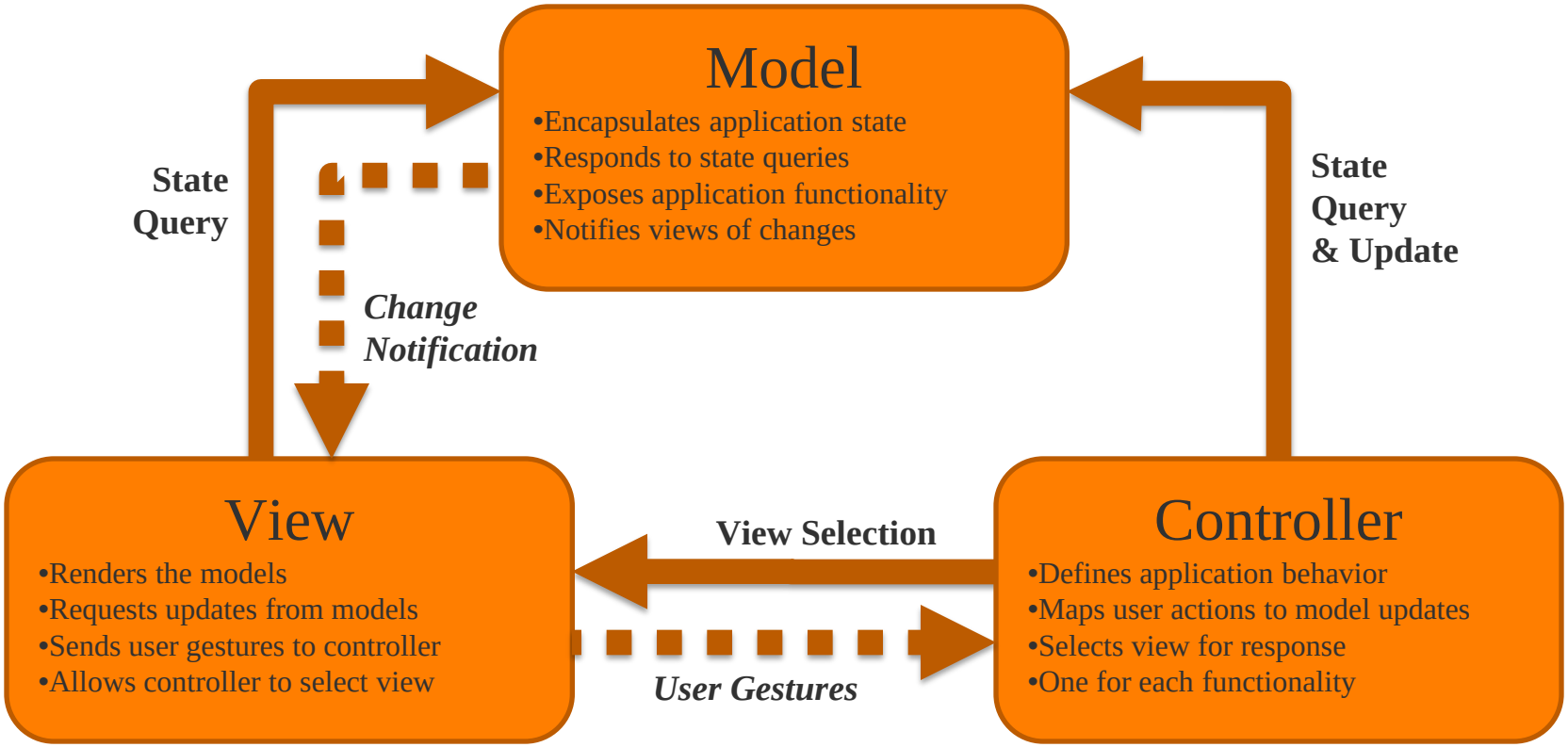
  …

- Problem (cont.)
  - Same data needs to be accessed by all client interfaces
    - e.g. HTML, WML, Swing, XML
  - Same data needs to be updated through different interactions
    - e.g. links in HTML or WML; buttons in Swing; SOAP messages written in XML
  - Supporting multiple types of views and interactions should not impact the components providing the core functionality

- Solution
  - Separate core business model functionality from presentation and control logic
  - Such separation allows multiple views, which makes it easier to implement, test, and maintain multiple clients

Architectural Patterns

Patterns for Web Applications - MVC



- Result Callback (Asynchronous Invocation)

- Context

  Client applications are decoupled from the server application. The client resumes its work immediately after invoking an operation at the server and does not have to wait for the result.

- Problem

  The client application should not block while waiting for a result from a remote operation. Consider, e.g., a form provided by a web application, which validates input fields immediately on the server side. The user should not be interrupted when filling in the form.

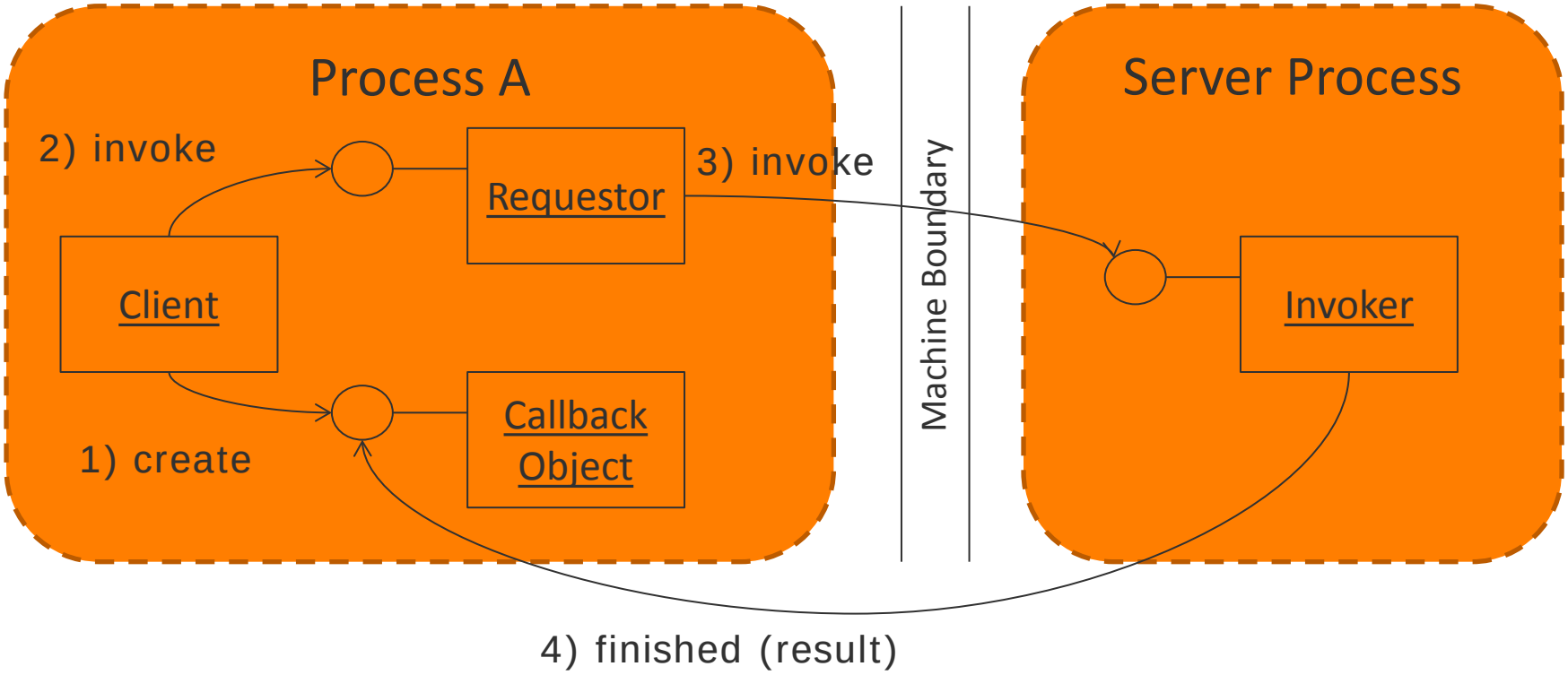
Architectural Patterns

Callback

- Solution
  - Provide a callback-based interface for remote invocations on the client.
  - Instantiate a callback object and invoke the remote operation via a requestor/proxy, which returns immediately after sending the invocation to the server. A predefined operation of the callback object is invoked as soon as the result is available.
- Related Patterns
  - Observer, Publish-Subscribe Messaging

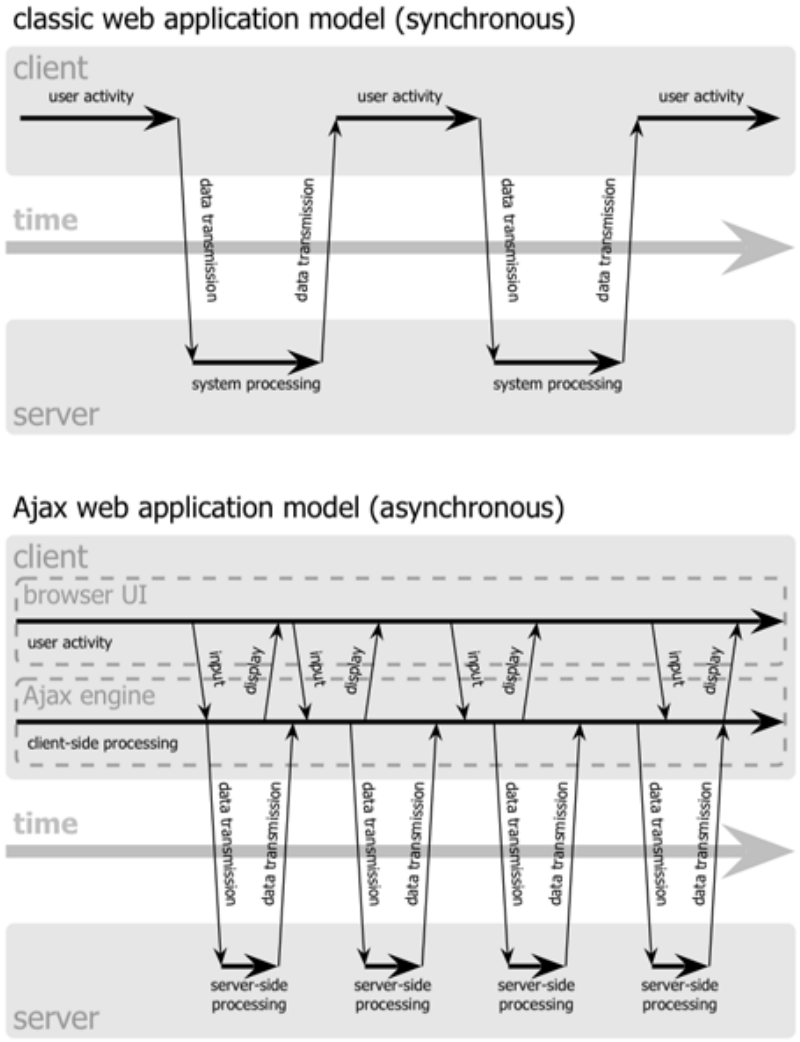
Architectural Patterns

Callback



according to [6]

Consequences of (A)Synchronous interaction patterns

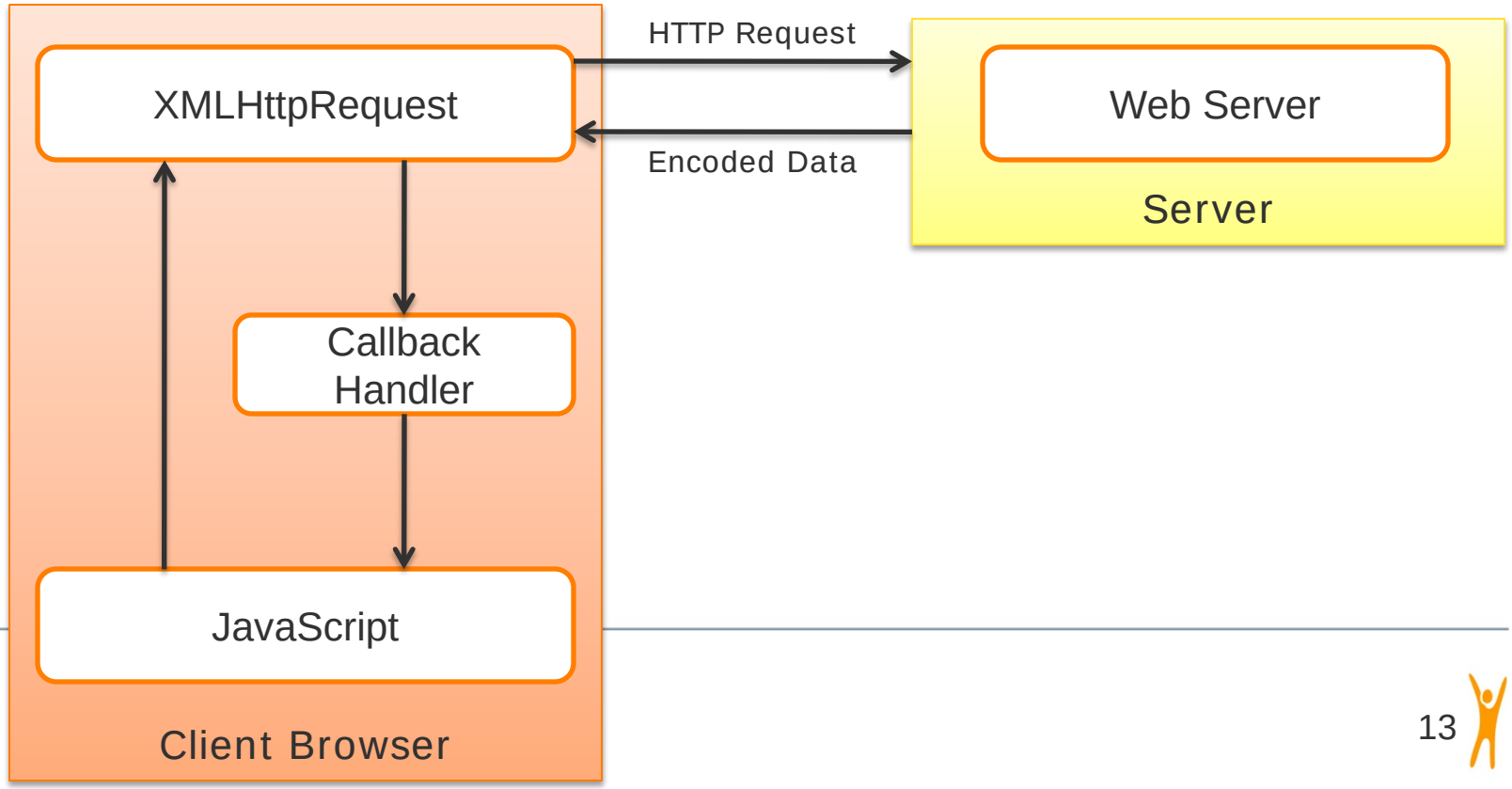


Taken from: A New Approach to Web Applications; Jesse James Garrett; February 18, 2005; http://www.adaptivepath.com/publications/essays/archives/000385.php

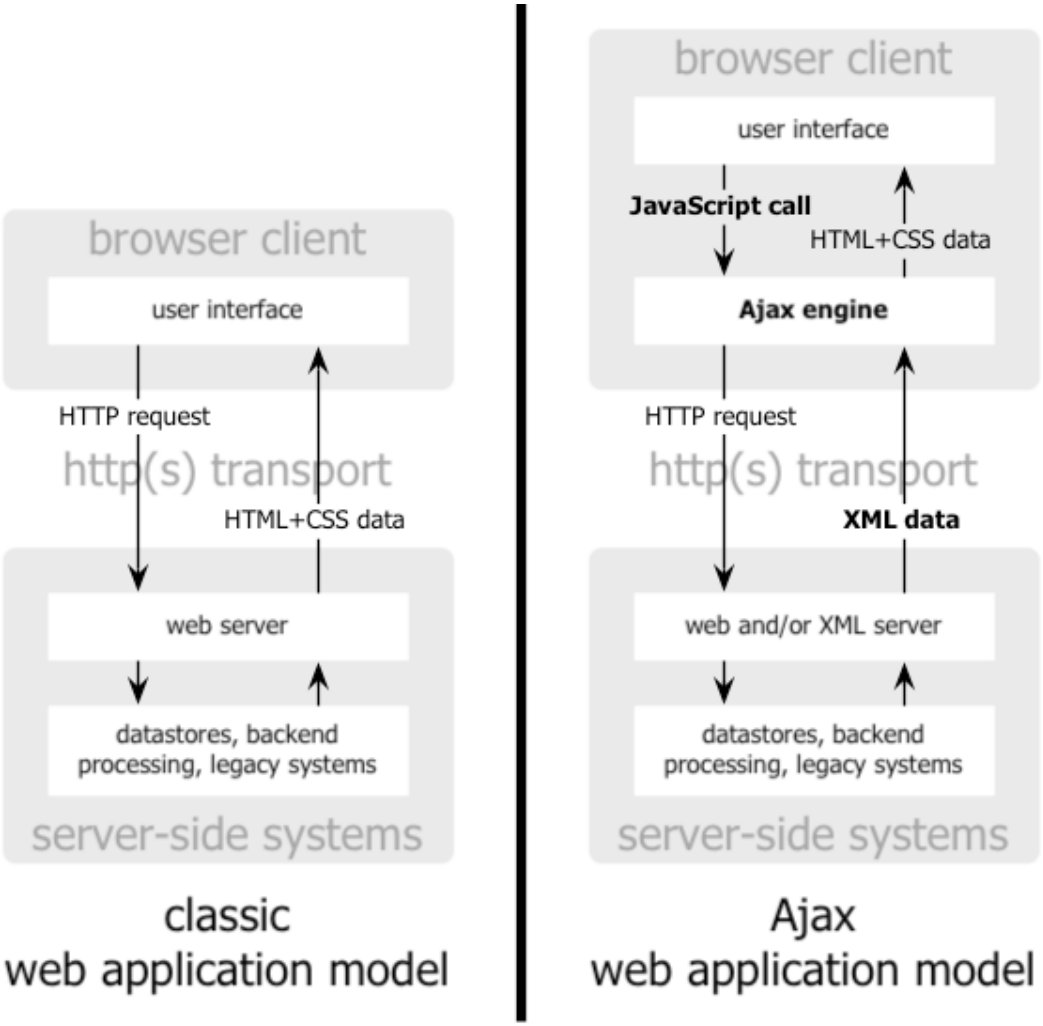
Implementing Callbacks in Web Applications

Ajax

- Requests over XMLHttpRequest
- Requests can be synchronous or asynchronous
- A callback handler listens for responses
- JavaScript integrates responded data into DOM



Traditional model vs. Ajax model



Taken from: A New Approach to Web Applications; Jesse James Garrett; February 18, 2005; http://www.adaptivepath.com/publications/essays/archives/000385.php

Ajax

XMLHttpRequest

- Implementation by browsers differ from W3C specification
  - Microsoft: ActiveX
  - Others: XMLHttpRequest
- Invented by Microsoft in 1999 with IE5
- W3C Working Draft
  - Part of the “Rich Web Clients Activity”
    - http://www.w3.org/2006/rwc/Activity
  - Started in April 2006
  - Current version: W3C Candidate Recommendation 3 August 2010
  - http://www.w3.org/TR/XMLHttpRequest/

Ajax

XMLHttpRequest

- Methods
  - open(method, url, asynchronous, user, password)

    Initializes an HTTP-Request

    method = CONNECT, DELETE, **GET**, HEAD, OPTIONS, **POST**, PUT, TRACE, or TRACK

    asynchron = true | false
  - send(*data*)

    Executes the request initialized with *open()*
  - abort()

    Cancels current communication
  - getResponseHeader(header)

    Returns the HTTP-header
  - getAllResponseHeaders()

    Returns all HTTP-headers of the response, delimited by CR/LF

    Or *null*/empty string if *readyState* is 3 or 4
  - setRequestHeader(*header*, *value*)

    Sets a HTTP header

Ajax

XMLHttpRequest

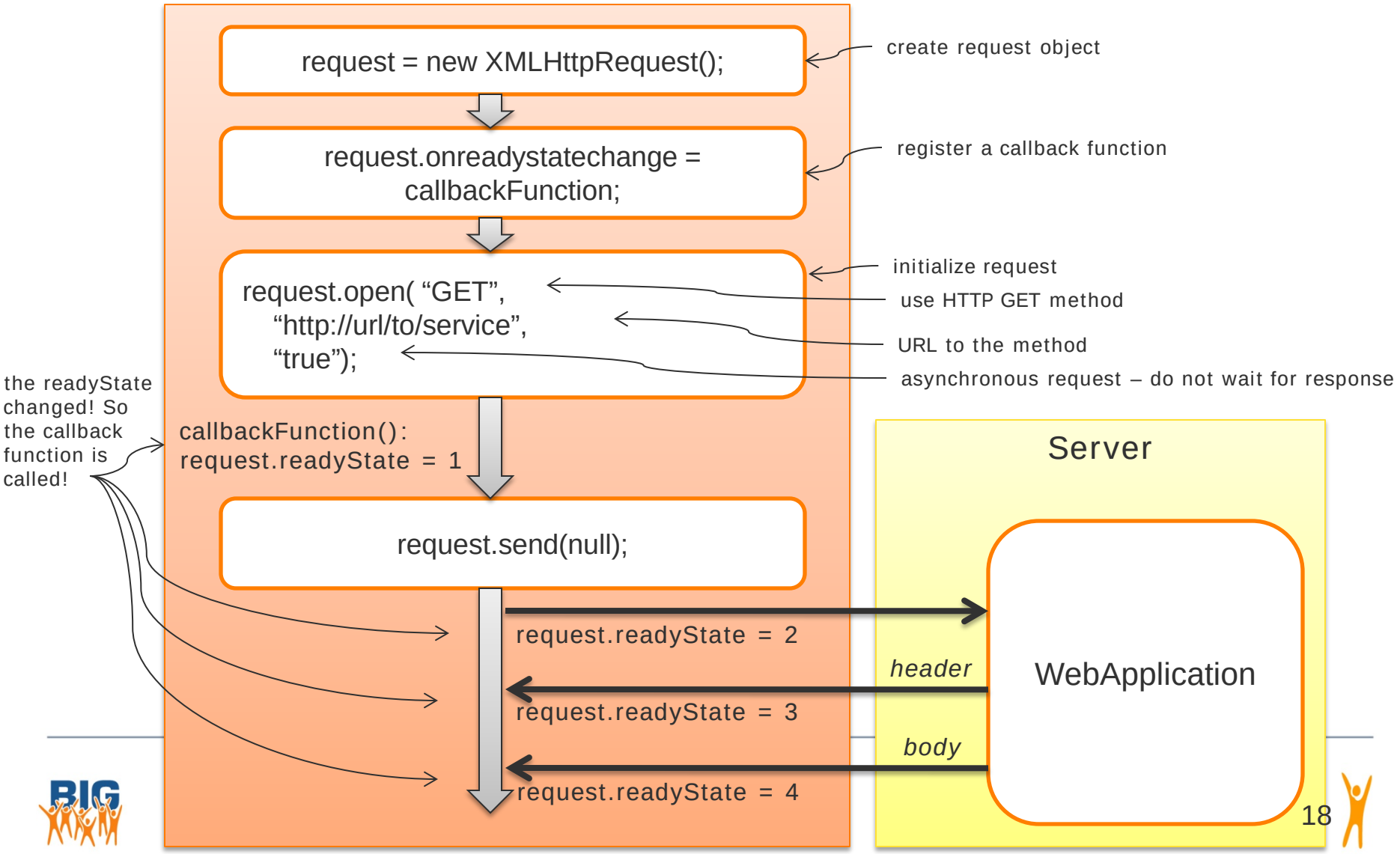
- Attributes
  - *onreadystatechange* – Callback, is called for *every* change of *readyState*
  - *readystate* - current state of request

Nr	W3	description
0	UNSENT	
1	OPENED	open() has been successfully called
2	HEADERS_RECEIVED	user agent (browser) successfully acknowledged request
3	LOADING	after HTTP header received, before message body receiving
4	DONE	data transfer completed

  - *responseText* – HTTP-body as text
  - *responseXML* – HTTP-body as DOM
  - *status* – HTTP-state code of the server's response
  - *statusText* – HTTP-state text of the server's response

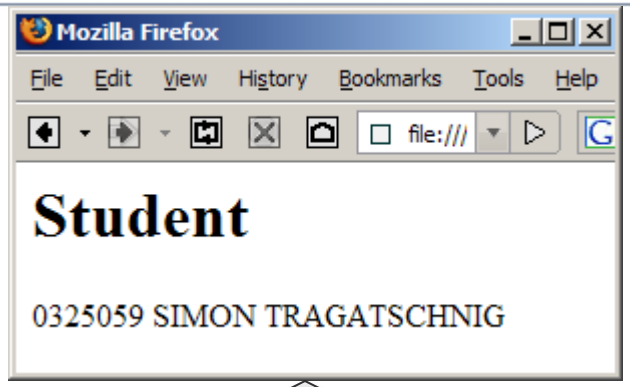
Ajax

XMLHttpRequest



Ajax

Little example



the response's data

```
<?xml version="1.0"
      encoding="UTF-8" ?>
<student>
  <mnr>0325059</mnr>
  <firstname>SIMON</firstname>
  <lastname>TRAGATSCHNIG</lastname>
</student>
```

here is the "hidden magic"

here should be the MNr

and here the student's name

```
<html>
  <head>
    <script type="text/javascript" src="script.js" />
  </head>
  <body onload="getStudent();">
    <h1>Student</h1>
    <p>
      <span id="mnr">no mnr</span>
      <span id="name">no name</span>
    </p>
  </body>
</html>
```

Ajax

Little example

- script.js – how to create the XMLHttpRequest for most browsers

```
function createXHR() {
    var xmlhttp;

    if (window.XMLHttpRequest)
    {
        xmlhttp = new XMLHttpRequest();
    }
    else
    {
        xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
    }

    return xmlhttp;
}
```

code for IE7+, Firefox, Chrome, Opera, Safari

this should work with MS IE5 and IE6

Ajax

Little example

- script.js – how to send the request

the function called by the HTML-event *onload*

create the request

initialize the request

register callback function

send the request

```
var request = null;

function getStudent() {
    request = createXHR();

    request.open("GET", “service.xml”, true);

    request.onreadystatechange = stateChanged;

    request.send(null);
}
```

Details of the initialization

```
request.open("GET", “service.xml”, true);
```

HTTP method

URL to application - in this case we refer relatively to a XML-file

an asynchronous call

Ajax

Little example

- script.js – how to handle the callbacks

```
function stateChanged() {
  switch(request.readyState) {
    case 0: break; // unsent
    case 1: break; // opened
    case 2: break; // sent
    case 3: break; // loading
    case 4: // done
      if(request.status == 0) {
         //everything ok
         /*
           add content to DOM
           see next slide …
         */
      }
      break;
  }
}
```

check the state of
the server's response:

- *0* for successful local file requests
- *200* for request to web server
- other HTTP status codes

Ajax

Little example

script.js – add content to DOM

```
<p>
  <span id="mnr">no mnr</span>
  <span id="name">no name</span>
</p>
```

the HTML file

the JavaScript

get span for mnr, name

set the text node's value of the span elements

get the text node's value of the response elements

```
var mnr = document.getElementById("mnr");
var name = document.getElementById("name");

mnr.firstChild.nodeValue =
  request.responseXML.getElementsByTagName("mnr")[0].
  firstChild.nodeValue;

name.firstChild.nodeValue =
  request.responseXML.getElementsByTagName("firstname")[0].
  firstChild.nodeValue + " " +
  request.responseXML.getElementsByTagName("lastname")[0].
  firstChild.nodeValue;
```

```
<student>
  <mnr>0325059</mnr>
  <firstname>SIMON</firstname>
  <lastname>TRAGATSCHNIG</lastname>
</student>
```

the server's response

Fundamental Design patterns

Object Oriented Paradigm

- OO Basics
  - Abstraction
  - Encapsulation
  - Polymorphism
  - Inheritance
- OO Principles
  - Encapsulate what varies
  - Favor composition over inheritance
  - Program to interfaces, not implementations
  - Depend upon abstractions. Do not depend upon concrete classes
  - Classes should be open for extension, but closed for modification
  - Don’t call us, we’ll call you
  - Strive for loosely coupled designs between objects that interact
  - Only talk to your friends
  - A class should have only one reason to change

Design Patterns

Overview

		Mission		
		Creational	Structural	Behavioral
Scope	Class-Based	Factory	Adapter	Interpreter Template
	Object-Based	Abstract Factory Factory Method Builder Prototype Singleton	Adapter Bridge Decorator Facade Flyweight Composite Proxy	Command Observer Visitor Iterator Memento Strategy Mediator State Chain of Responsibility

from [1],
the underlined patterns will be discussed in detail

Structural Design Patterns

Adapter Pattern

- Context
  An existing object should be used in your own application

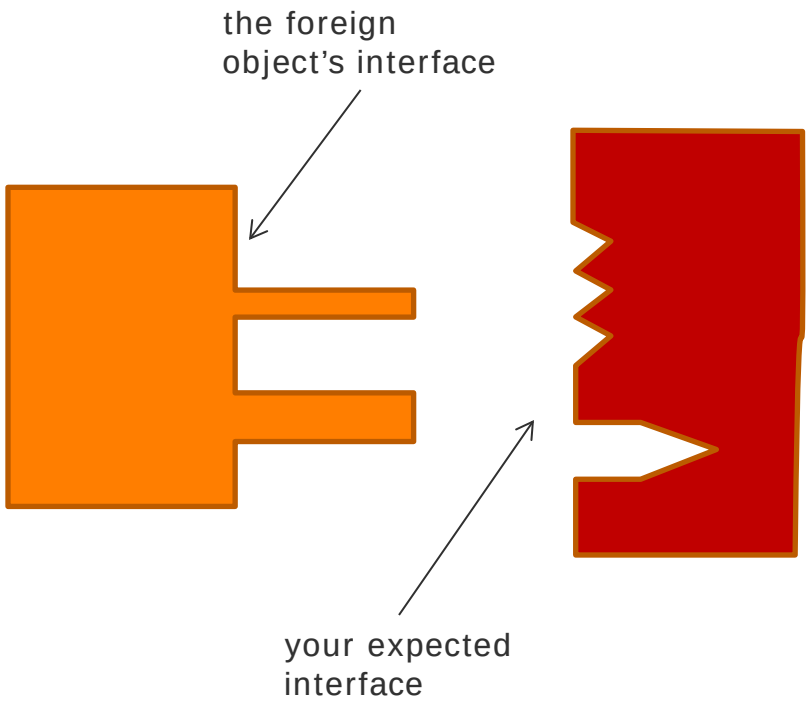
- Problems
  The foreign object's interface does not fit to your expected interfaces

- Solution
  - Convert the foreign object's interface to another interface
  - Wrap the foreign object with a new interface

Structural Design Patterns

Adapter Pattern

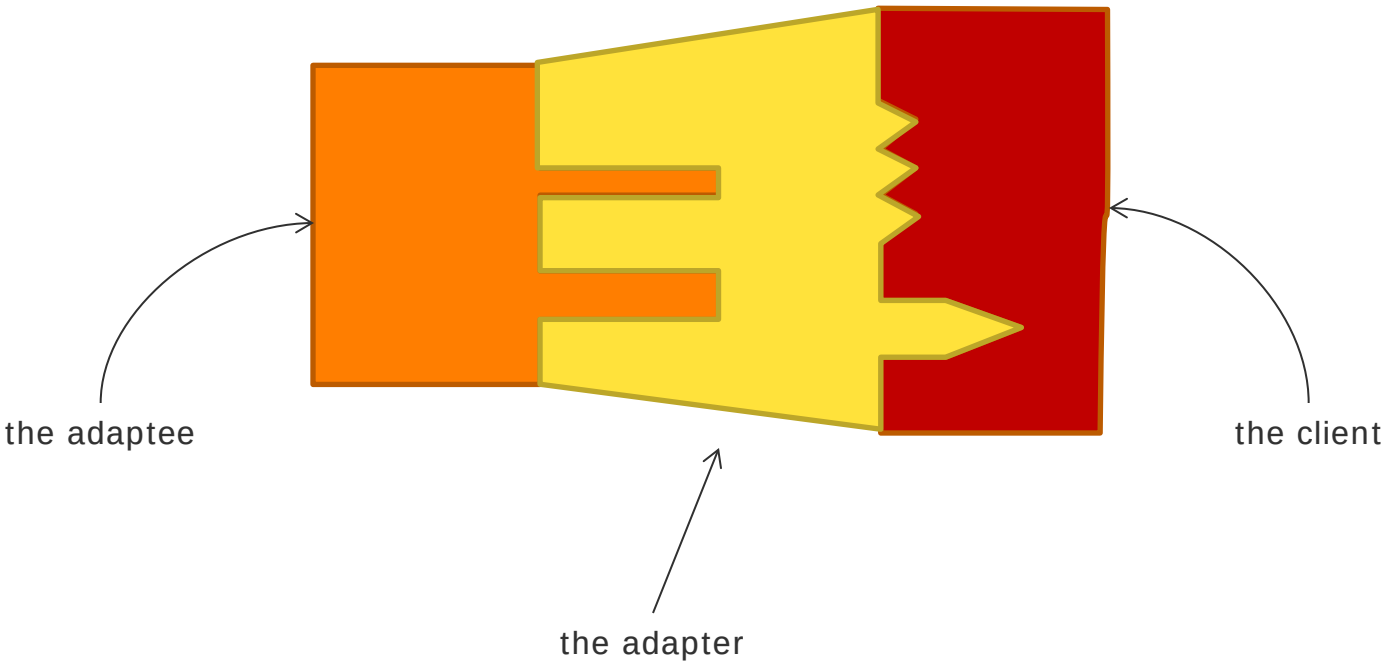
- How can they get connected?



Structural Design Patterns

Adapter Pattern

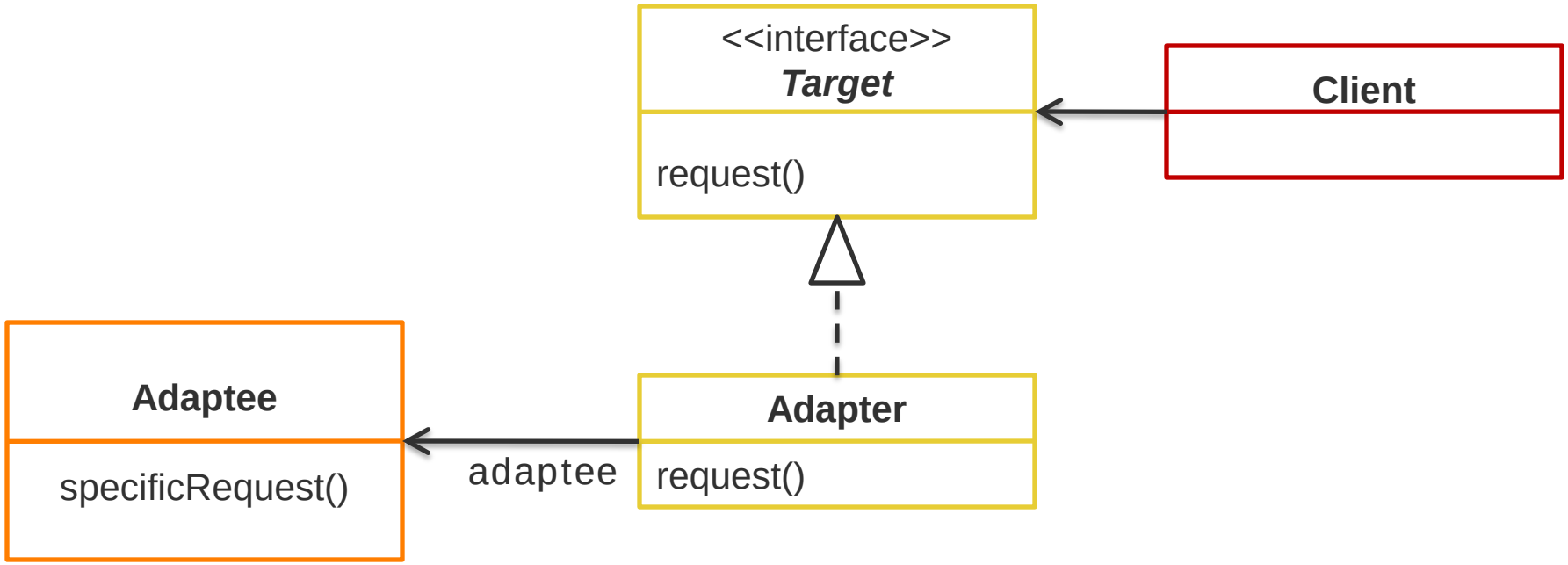
- How can they connect together?
  - ➔ use an adapter!



Structural Design Patterns

Adapter Pattern

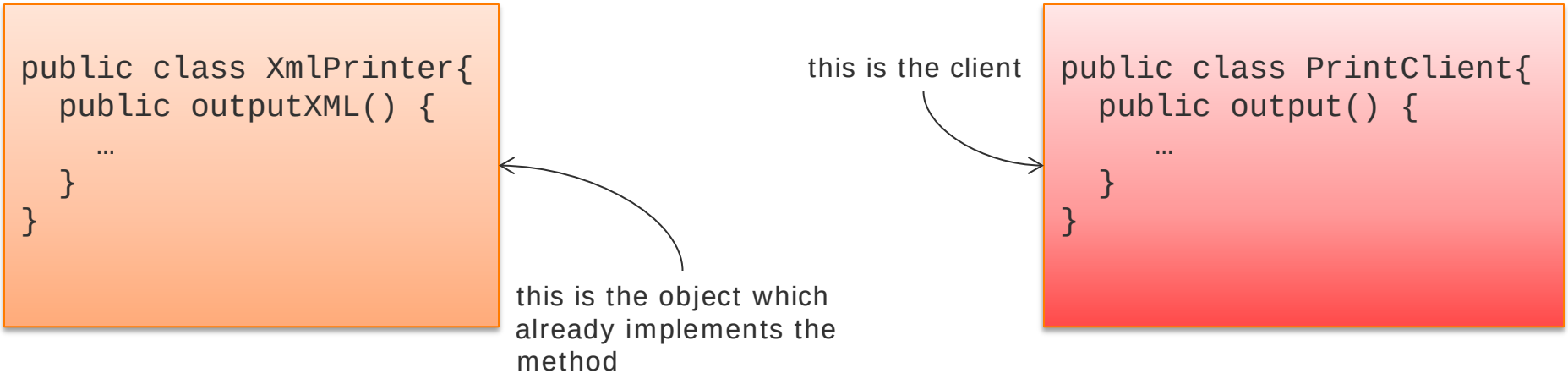
- Definition
  Converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. [1]



Structural Design Patterns

Adapter Pattern

- Let’s try to realize

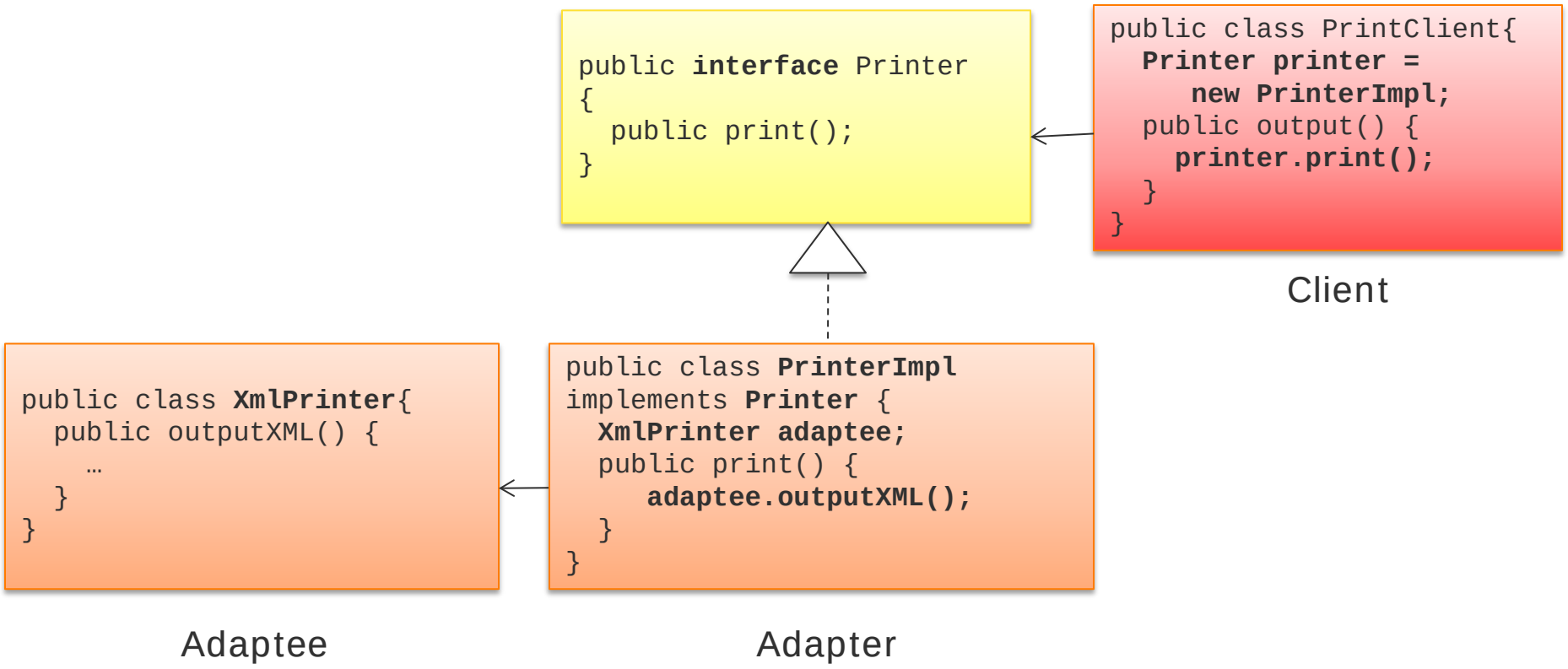


- How can they interact in a sophisticated way?

Structural Design Patterns

Adapter Pattern

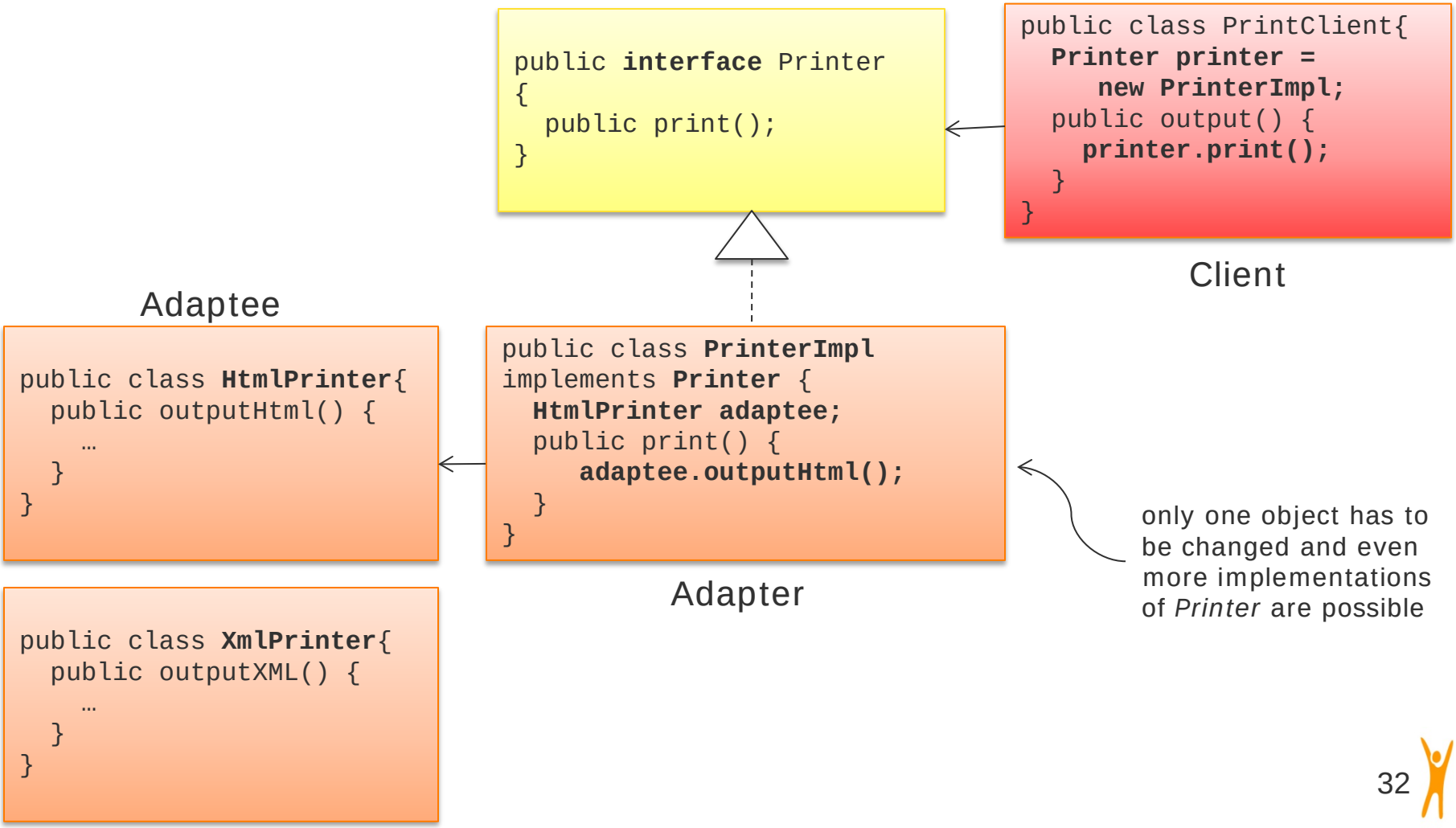
- What happens if I want to use the object “HtmlPrinter” instead of “XmlPrinter”?



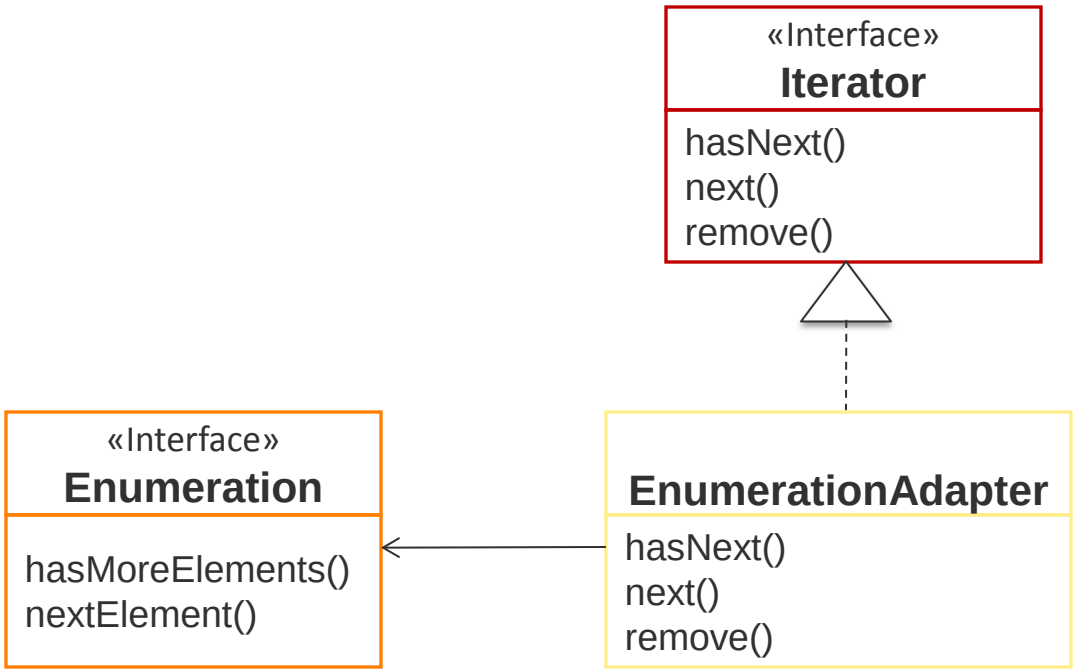
Structural Design Patterns

Adapter Pattern

- Let's change *XmlPrinter* to *HtmlPrinter*



- Enumeration/Iterator Integration



Structural Design Patterns

Facade Pattern

- Context
  A set of objects (subsystem) providing a certain functionality should be usable by a simple interface

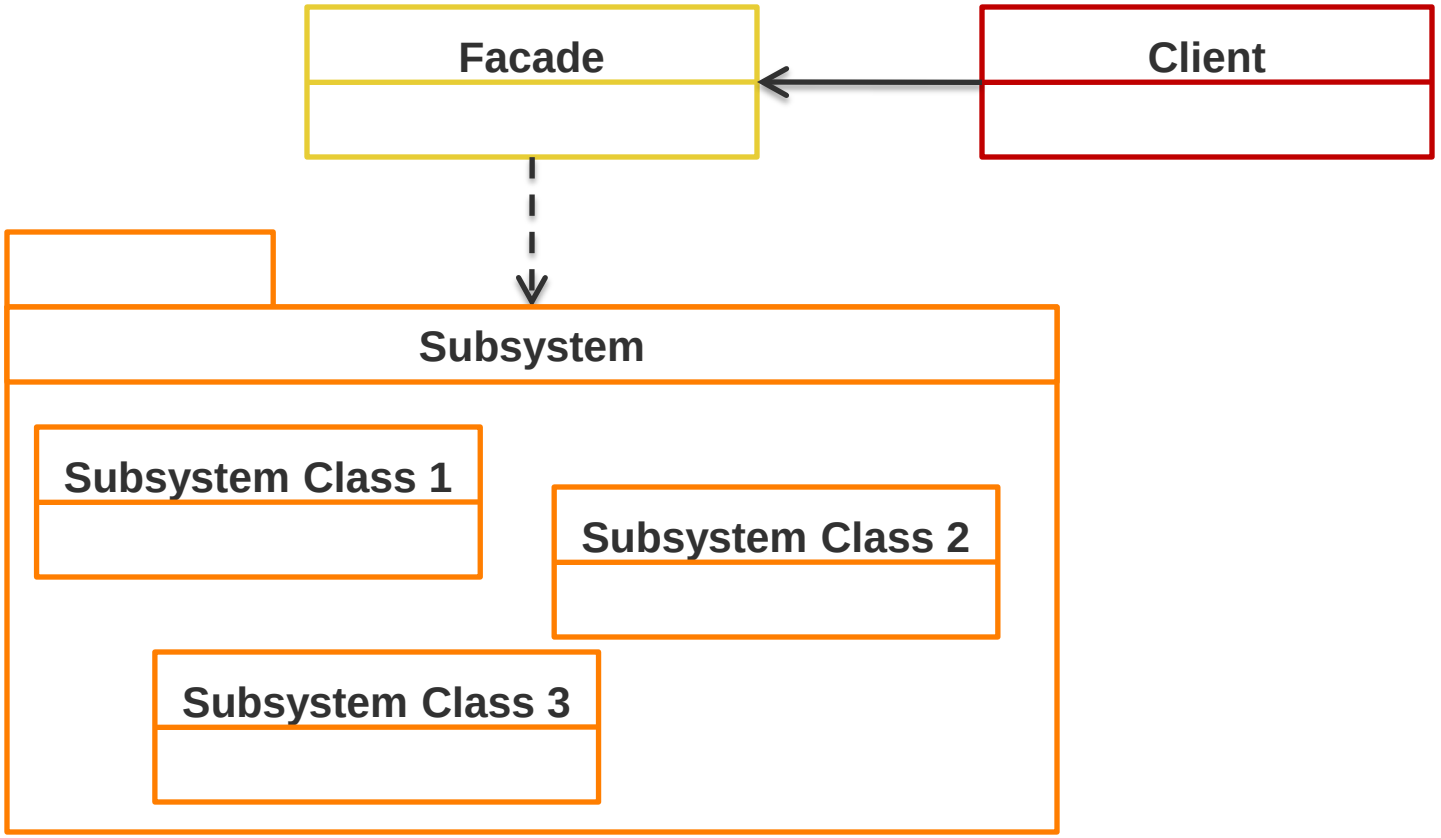
- Problems
  The subsystem is too complex to handle it easily

- Solution
  Simplify the interface to the overall functionality of the complex subsystem

Structural Design Patterns

Facade Pattern

- Definition
  The Facade-Pattern provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use. [1]

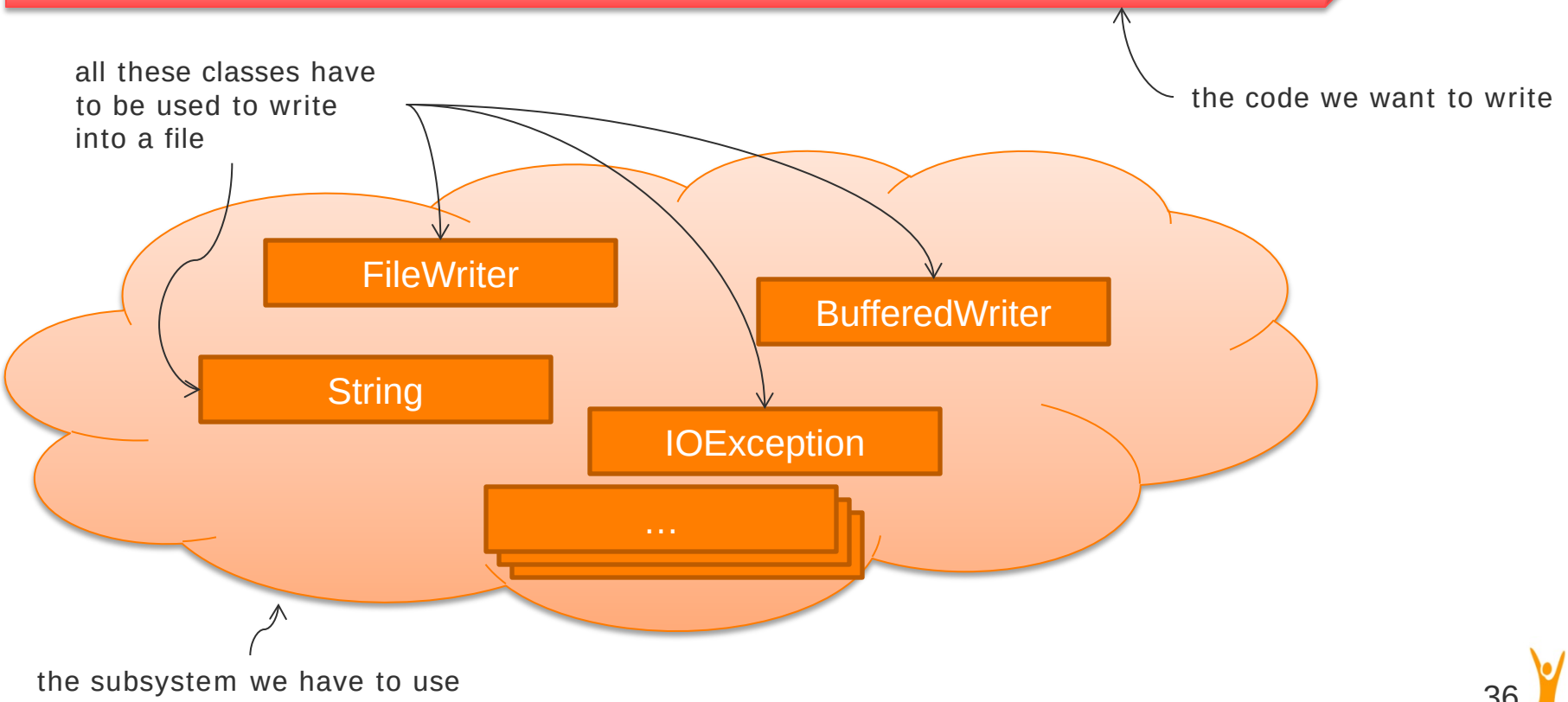


Structural Design Patterns

Facade Pattern

- Let's try to realize a simple class to write into a file

```
MySimpleFileWriter.write("myfile.txt", "my first file");
```



Structural Design Patterns

Facade Pattern

- Let's try to realize a simple class to write into a file

the client

```
…
MySimpleFileWriter.write("myfile.txt", "my first file");
…
```

the facade

```
public class MySimpleFileWriter {
   public static void write(String filename, String content) {
      try {
        FileWriter fw = new FileWriter(filename);
        BufferedWriter out = new BufferedWriter(fw);
        out.write(content);
        out.close();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

Creational Design Patterns

- What is the problem with ***new*** operator?

- Example – Pizza offer varies!

```
Pizza orderPizza(String typ){
   Pizza pizza;
   if (typ.equals(“Salami“)) {
      pizza = new SalamiPizza();
   }else if (typ.equals(“Spinat“)){
      pizza = new SpinatPizza();
   }else if (typ.equals(“Thunfisch“)){
      pizza = new ThunfischPizza();
   }
    …
   pizza.prepare();
   pizza.pack();
   …
}
```

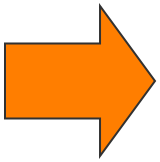
This part is subject to permanent change

Constant Part

Creational Design Patterns

- Encapsulate Object Creation into Factory

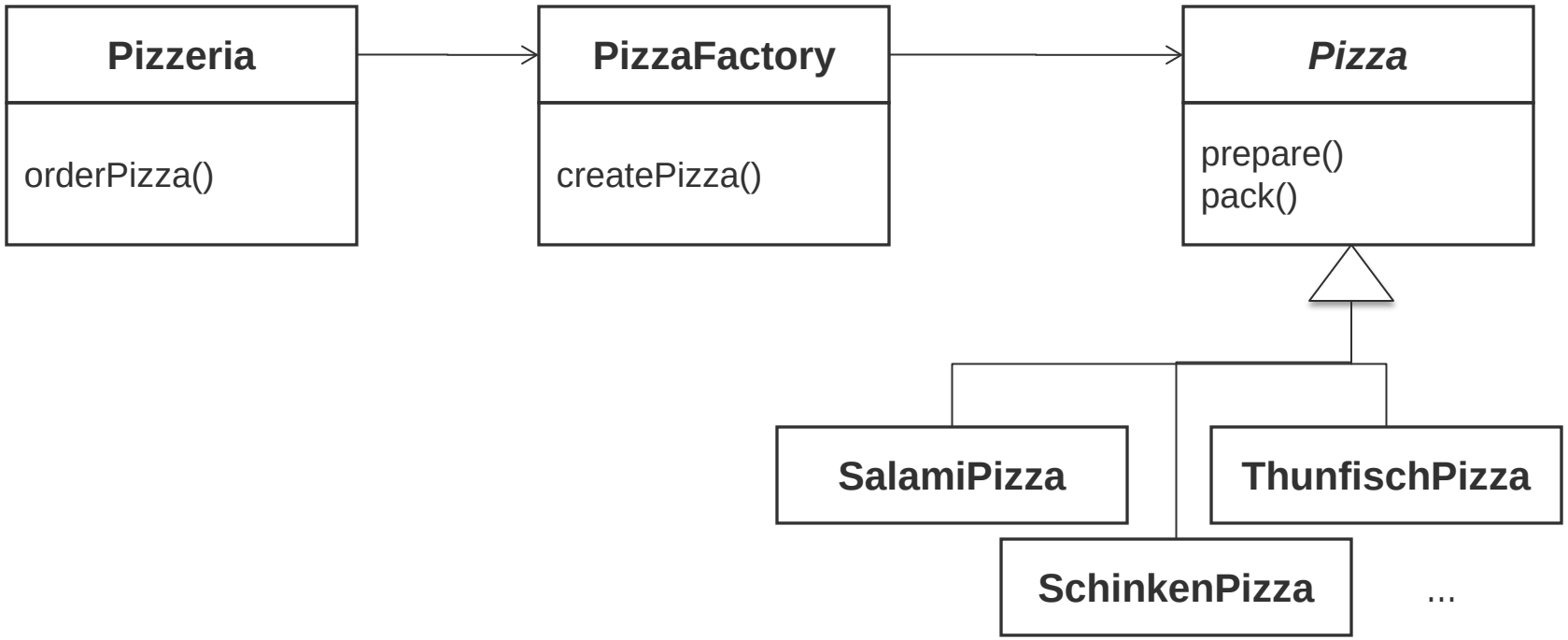
```
public class Pizzeria{

Pizza orderPizza(String typ){
   Pizza pizza;
   if (typ.equals(“Salami“)) {
          pizza = new SalamiPizza();
   }else if (typ.equals(“Spinat“)){
          pizza = new SpinatPizza();
   }else if (typ.equals(“Thunfisch“)){
          pizza = new ThunfischPizza();
   }
          …
   pizza.prepare();
   pizza.pack();
   …
}

}
```



```
public class PizzaFactory{

public Pizza createPizza(String typ)
   Pizza pizza = null;
   if (typ.equals(“Salami“)) {
          pizza = new SalamiPizza();
   }else if (typ.equals(“Spinat“)){
          pizza = new SpinatPizza();
   }else if(typ.equals(“Thunfisch“)){
          pizza = new ThunfischPizza();
   }…

   return pizza
}

public class Pizzeria{
  PizzaFactory factory;

  Pizza orderPizza(String typ){
       Pizza pizza = factory.createPizza(typ);
       pizza.prepare();
       pizza.pack();
  }
}
```

Creational Design Patterns

Factory * Patterns



Creational Design Patterns

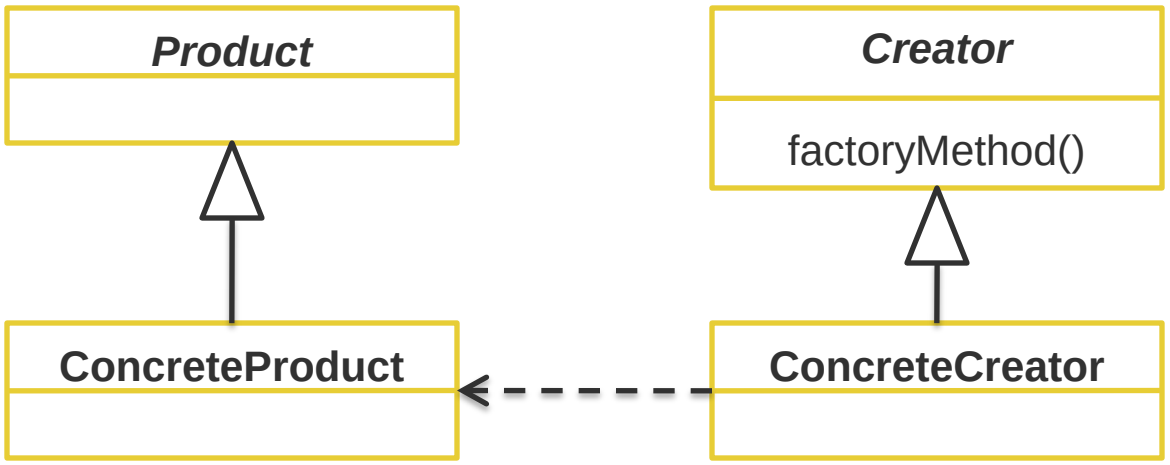
Factory Method Pattern

- Context
  - Create an object under the principle “Program to an interface, not to an implementation”. Because there is an interface, all implementations of it would be a group of related classes.
- Problems
  - Because we “Program to an interface, not to an implementation” an object only knows interfaces of other objects and nothing about how and which object to be instantiated. The creator class only knows when and not what kind of object to create.
  - With new a concrete class is specified
- Solution
  - An object has to be responsible to create the right instance at runtime.

Creational Design Patterns

Factory Method Pattern

- Definition
  The Factory Method Pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses. [1]



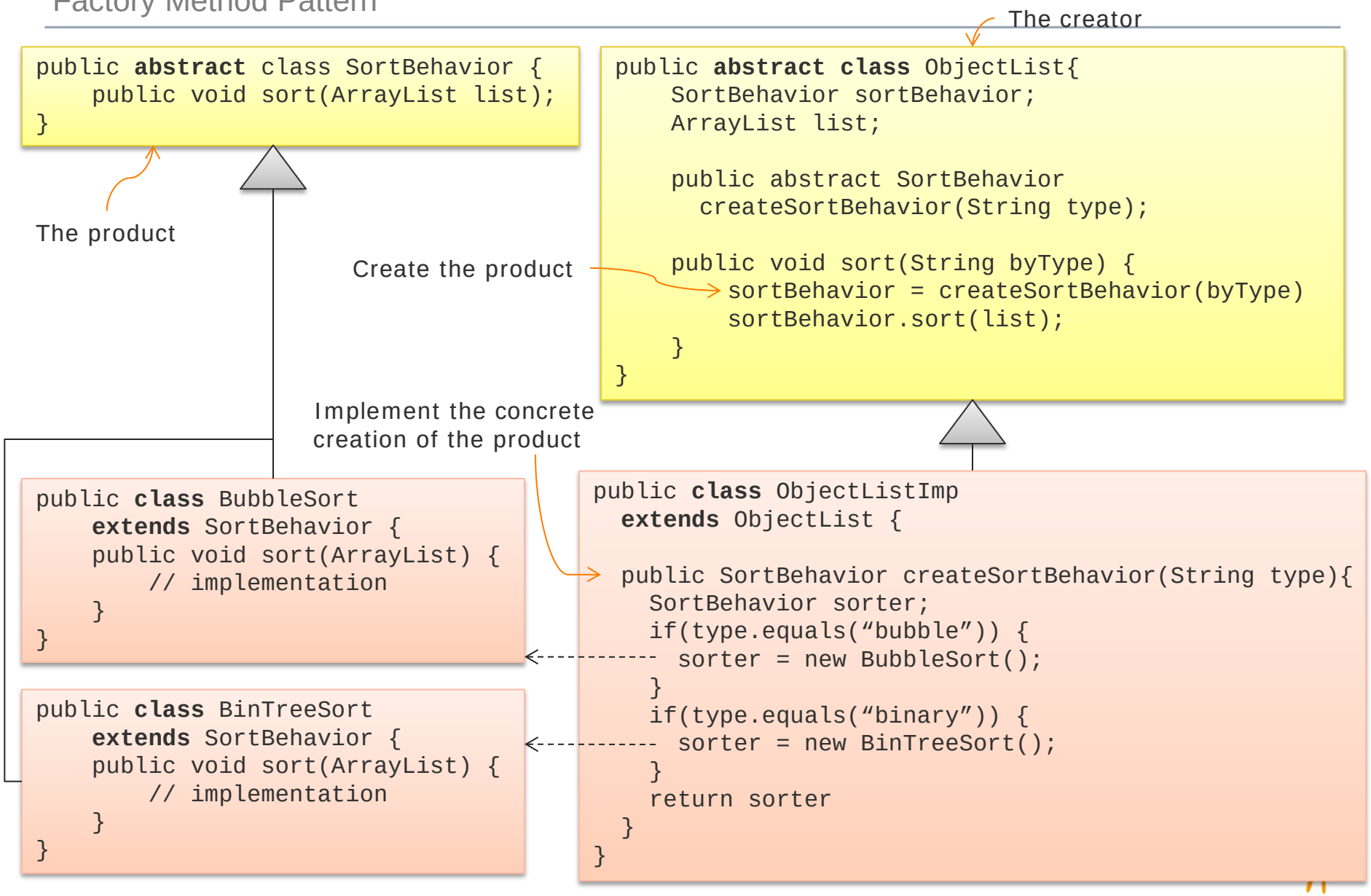
Creational Design Patterns

Factory Method Pattern

- Let’s try to realize
  - A factory creating an *ObjectList* where the sorting algorithm will be selected by a string
- Product
  - Our well-known *SortBehavior* interface
- Concrete products
  - *BubbleSort*, *BinTreeSort*, …
- Creator
  - The *ObjectList* interface with method *createSortBehavior() : SortBehavior*
- Concrete creators
  - The implementation of the interface: *ObjectListImp*

Creational Design Patterns

Factory Method Pattern



Creational Design Patterns

Factory Method Pattern

- Use this implementation

sorts with Bubble Sort

sorts with Binary Tree Sort

```
public class TestObjectList{
    public static void main(String args[]) {
        ObjectList list = new ObjectListImp;

        list.sort("bubble");

        list.sort("binary");
    }
}
```

Creational Design Patterns

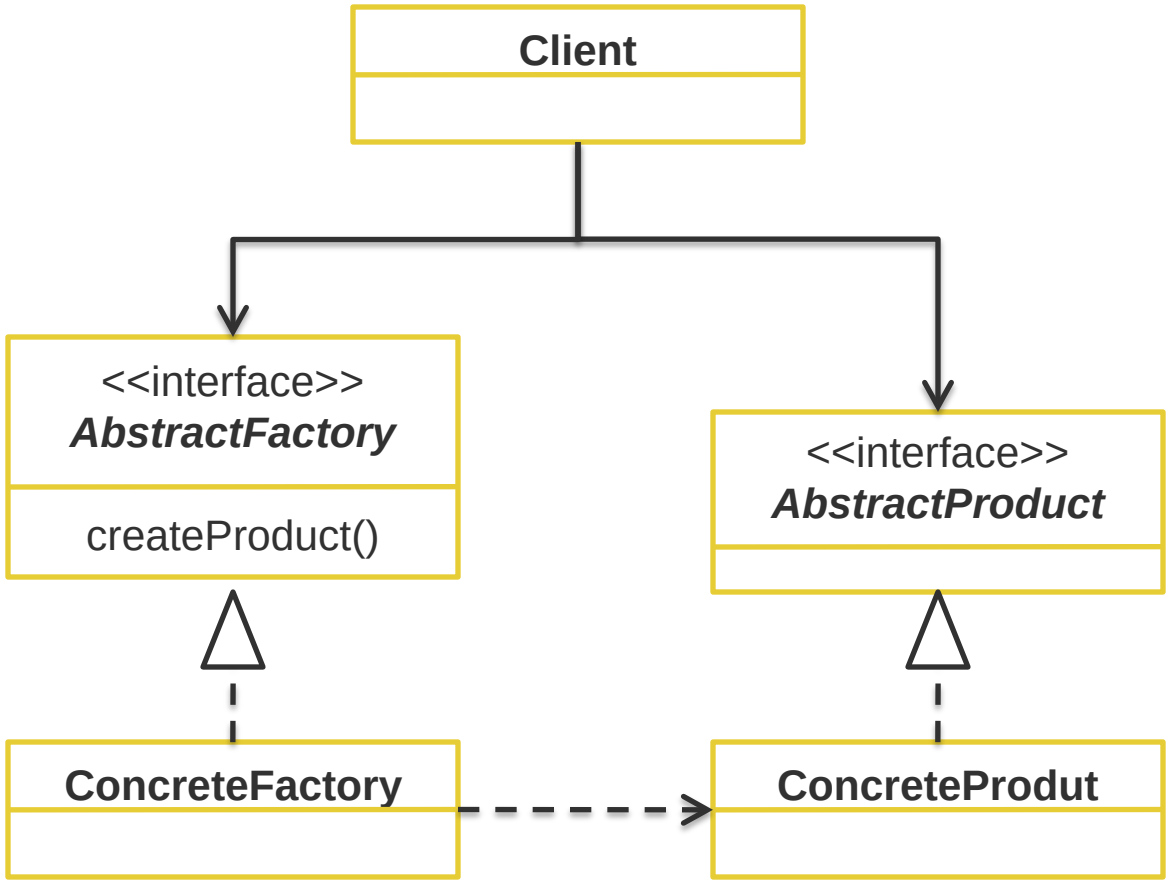
Abstract Factory Pattern

- Context
  - We want to create families of related or dependent objects, but without specifying their concrete classes.
  - (e.g. Sorter ◁– character | numeric ◁– Bubble | Binary Tree)
- Problems
  - Dependency Inversion Principle:
    Depend upon abstractions. Do not depend upon concrete classes.
- Solution
  - Group similar objects
  - Use the Abstract Factory-Patterns

Creational Design Patterns

Abstract Factory Pattern

- Definition
  The Abstract Factory-Pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes. [1]



Creational Design Patterns

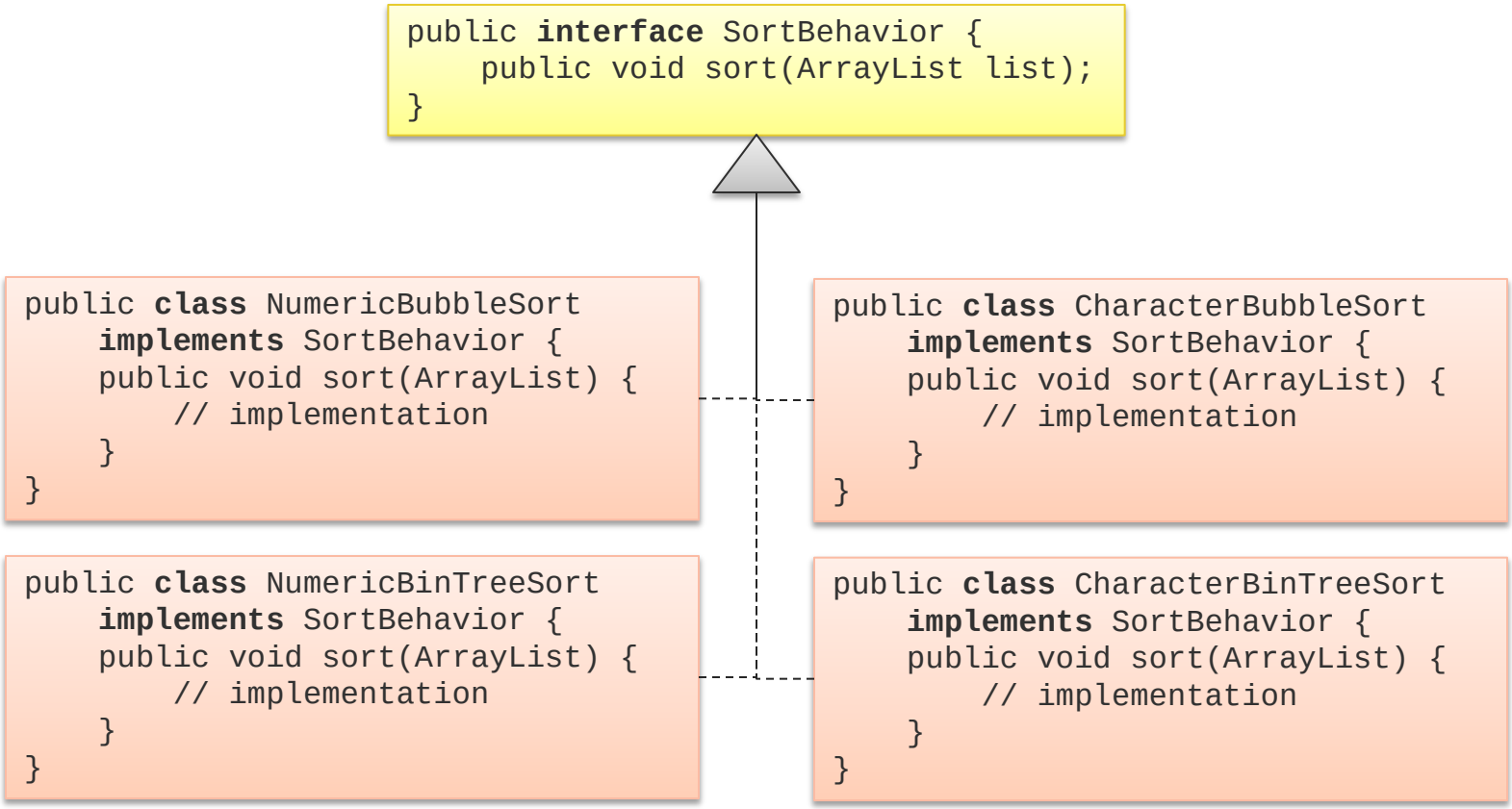
Abstract Factory Pattern

- Let’s try to realize
  - An abstract Factory for handling numeric and character based sort algorithms
- Product
  - SortBehavior
- Concrete Products
  - CharacterBubbleSort
  - CharacterBinTreeSort
  - NumericBubbleSort
  - NumericBinTreeSort
- Factory
  - SorterFactory
- Concrete Factories
  - CharacterSorterFactory
  - NumericSorterFactory

Creational Design Patterns

Abstract Factory Pattern

- The products

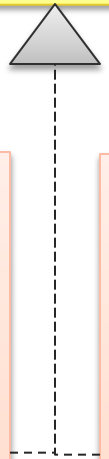


Creational Design Patterns

Abstract Factory Pattern

- The factories

```
public interface SorterFactory {
    public SorterFactory create(String type);
}
```



```
public class NumericSorterFactory
  implements SorterFactory {

  public SorterFactory create(String type)
  {
      SortBehavior sorter;
      if(type.equals("bubble")) {
        sorter = new NumericBubbleSort();
      }
      if(type.equals("binary")) {
        sorter = new NumericBinTreeSort();
      }
      return sorter
  }
}
```

```
public class CharacterSorterFactory
  implements SorterFactory {

  public SorterFactory create(String type)
  {
      SortBehavior sorter;
      if(type.equals("bubble")) {
        sorter = new CharacterBubbleSort();
      }
      if(type.equals("binary")) {
        sorter = new CharacterBinTreeSort();
      }
      return sorter
  }
}
```

- The client

```
public class ObjectList {
    SortBehavior sortBehavior;
    ArrayList list;

    public void sort(String type, SorterFactory factory) {
        sortBehavior = factory.create(type);
        sortBehavior.sort(list);
    }
}
```

  - The ObjectList-implementation never uses concrete instances (no *new*)!

- Use this implementation

sorts characters with Bubble Sort

sorts numbers with Binary Tree Sort

```
public class TestObjectList{
    public static void main(String args[]) {
        ObjectList list = new ObjectList;

        list.sort("bubble", new CharacterSorterFactory());
        list.sort("binary", new CharacterSorterFactory();

        list.sort("bubble", new NumericSorterFactory());
        list.sort("binary", new NumericSorterFactory());
  }
}
```

  - Easy to extend with other families of algorithms
    - just implement the factory and product interfaces

Behavioral Design Patterns

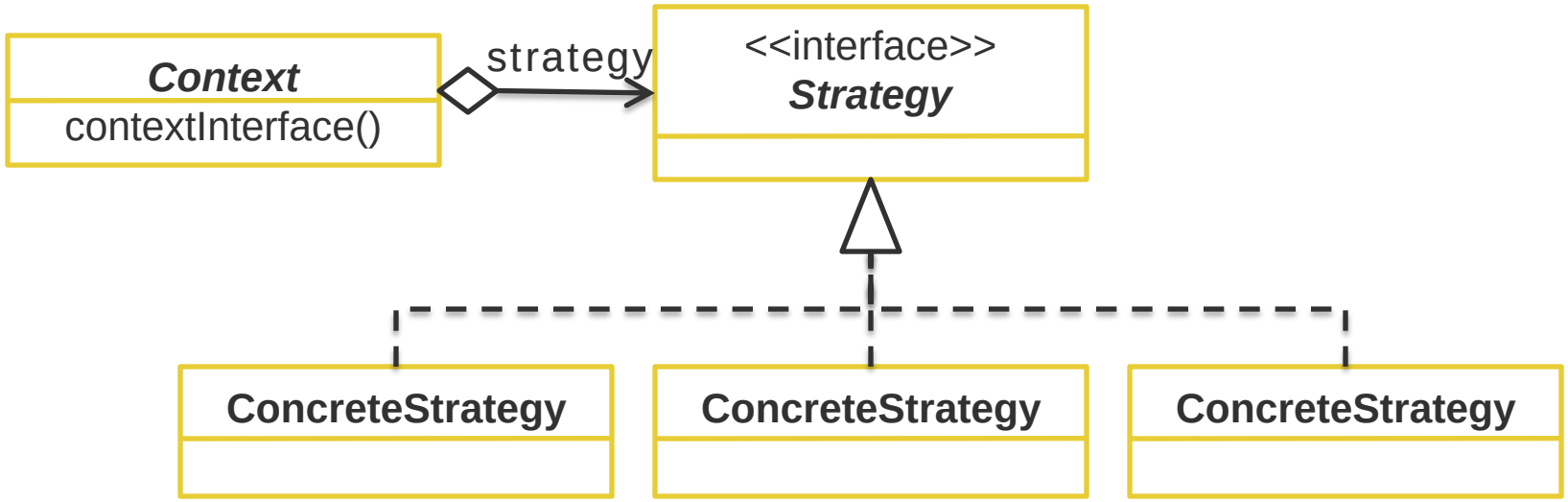
Strategy Pattern

- Context
  - Different objects with the same interface have different behaviors. E.g., objects which can sort, using Bubble Sort, Binary Tree Sort, Quicksort, or any other sorting algorithm.
- Problems
  - By inheritance
    - Duplicated code across subclasses
    - Difficult behavior change at runtime
  - By interface
    - No code reuse of same behavior
  - Design Principle: Favor composition over inheritance
- Solution
  - Encapsulate behavior

Behavioral Design Patterns

Strategy Pattern

- Definition
  The Strategy-Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it. [1]



Behavioral Design Patterns

Strategy Pattern

- Use it when
  - Many related classes only differ in behavior
  - Different, interchangeable variants of algorithms are needed

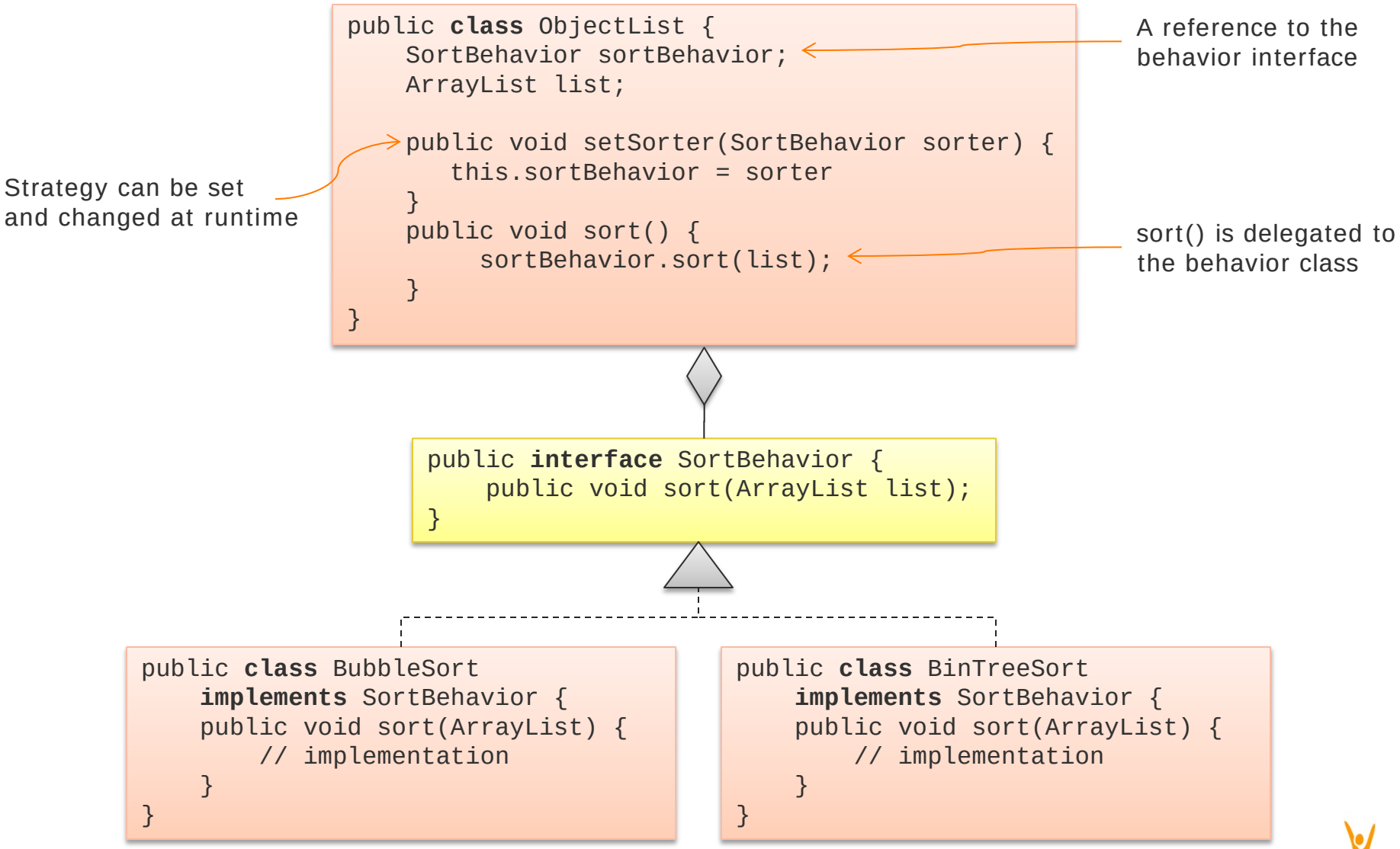
- Implement it by
  - Separating the aspects that vary from what stays the same
  - An interface for the strategy
  - Inheritance for the concrete strategy

Behavioral Design Patterns

- Let’s try to realize
  - ObjectList: containing a list of other objects
  - ObjectList.sort() can sort its list of objects
  - The sorting algorithm should be selectable at runtime
- Strategy
  - sort(list : ObjectList)
- Concrete Strategies:
  - BubbleSort, BinTreeSort, …
- Context
  - ObjectList

Behavioral Design Patterns

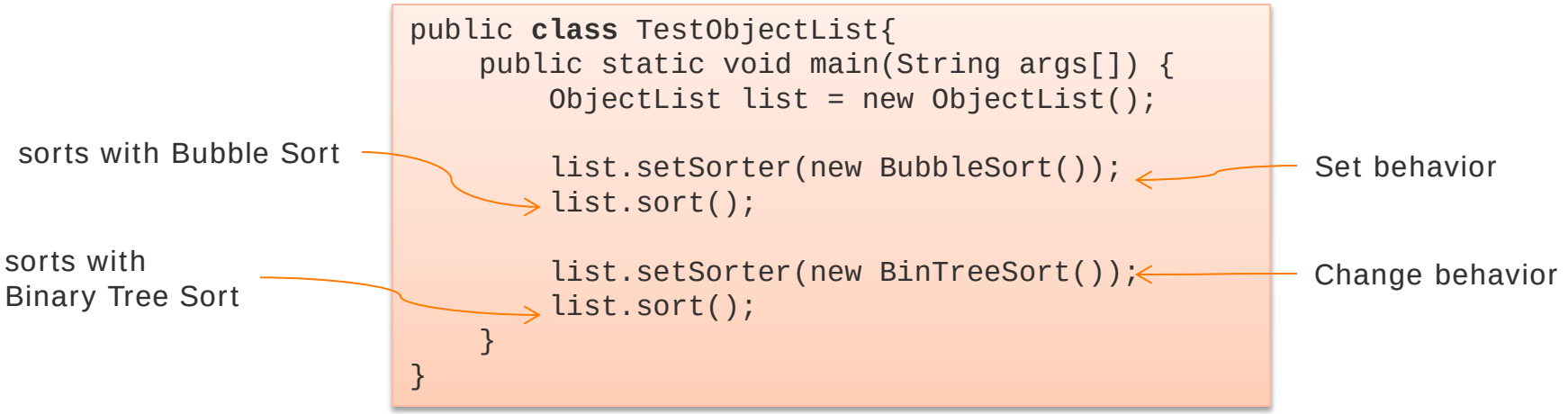
Strategy Pattern



Behavioral Design Patterns

Strategy Pattern

- Use this implementation:




```
public class TestObjectList{
    public static void main(String args[]) {
        ObjectList list = new ObjectList();

        list.setSorter(new BubbleSort());
        list.sort();

        list.setSorter(new BinTreeSort());
        list.sort();
    }
}
```

Behavioral Design Patterns

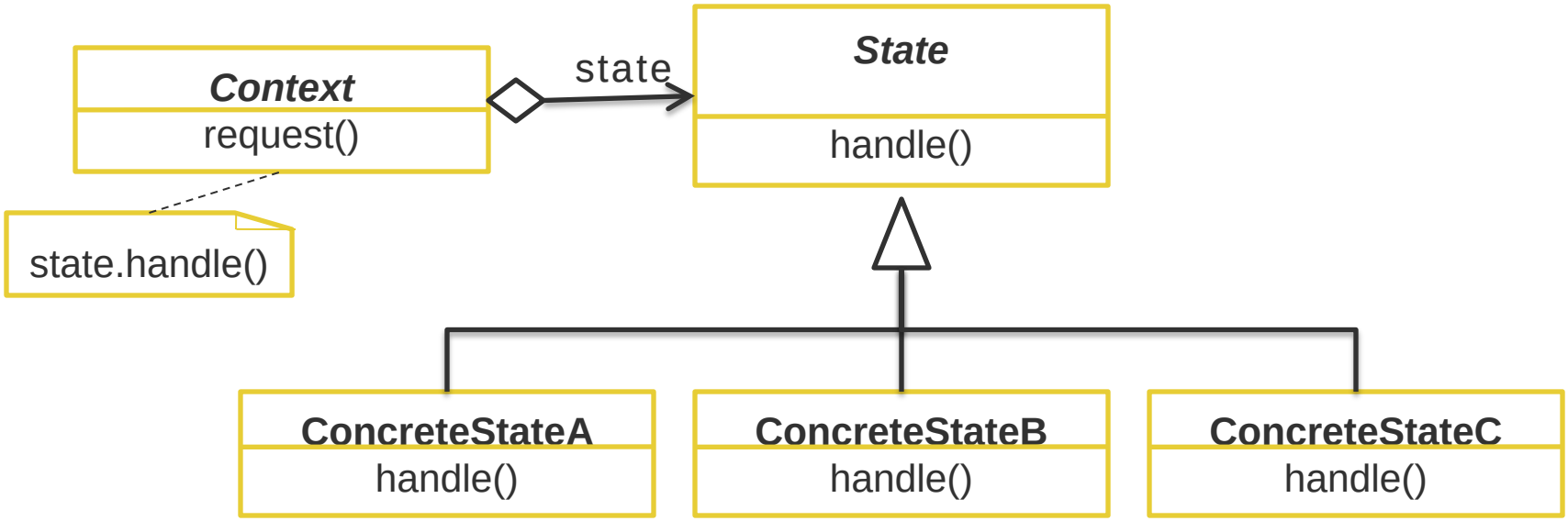
State Pattern

- Context
  - Encapsulates the states of an object as discrete objects, each extending a common superclass.
- Problems
  - An object's state is one of a predetermined set of values. When an object becomes aware of an external event, its state may change.
  - The behavior of a stateful object is determined by its state which may change at run-time.
- Solution
  - An object's behavior is determined by an internal state that changes in response to events.

Behavioral Design Patterns

State Pattern

- Definition
  The state pattern allow an object to alter its behavior when its internal state changes. The object will appear to change its class. [1]



Behavioral Design Patterns

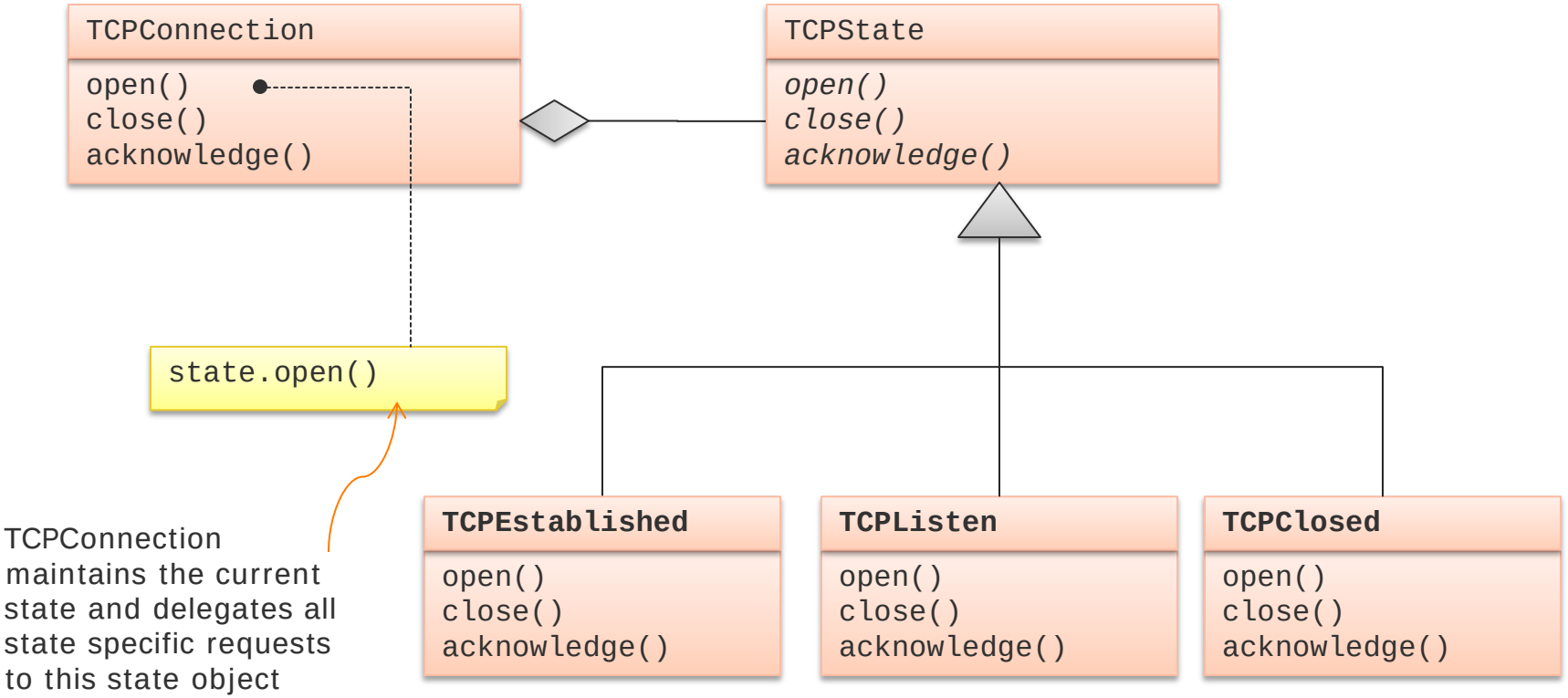
State Pattern

- Use it when
  - An object's behavior depends on its state, and it must change its behavior at runtime depending on that state.
  - Operations have large, multipart conditional statements that depend on the object's state.

- Implement it by
  - Defining the transition stimuli of a state machine and extract it to an abstract base class
  - Defining each state by extending the abstract base class and implementing the states response to each transition
  - Defining a delegation wrapper class that holds an instance of the "current" state
  - Each state method may call back to the wrapper to change the current state object

Behavioral Design Patterns

State Pattern



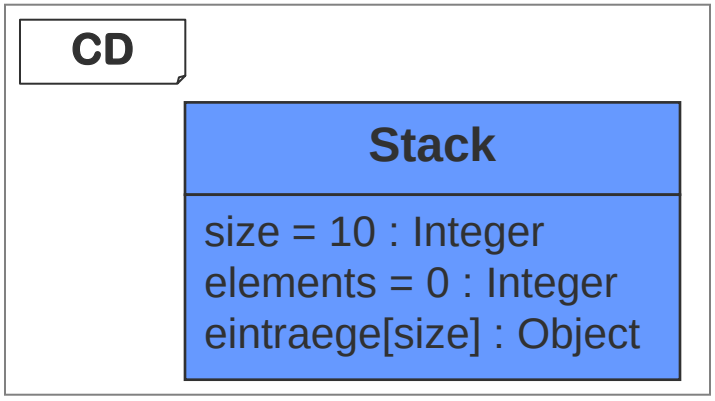
Appendix: State Patterns in Action

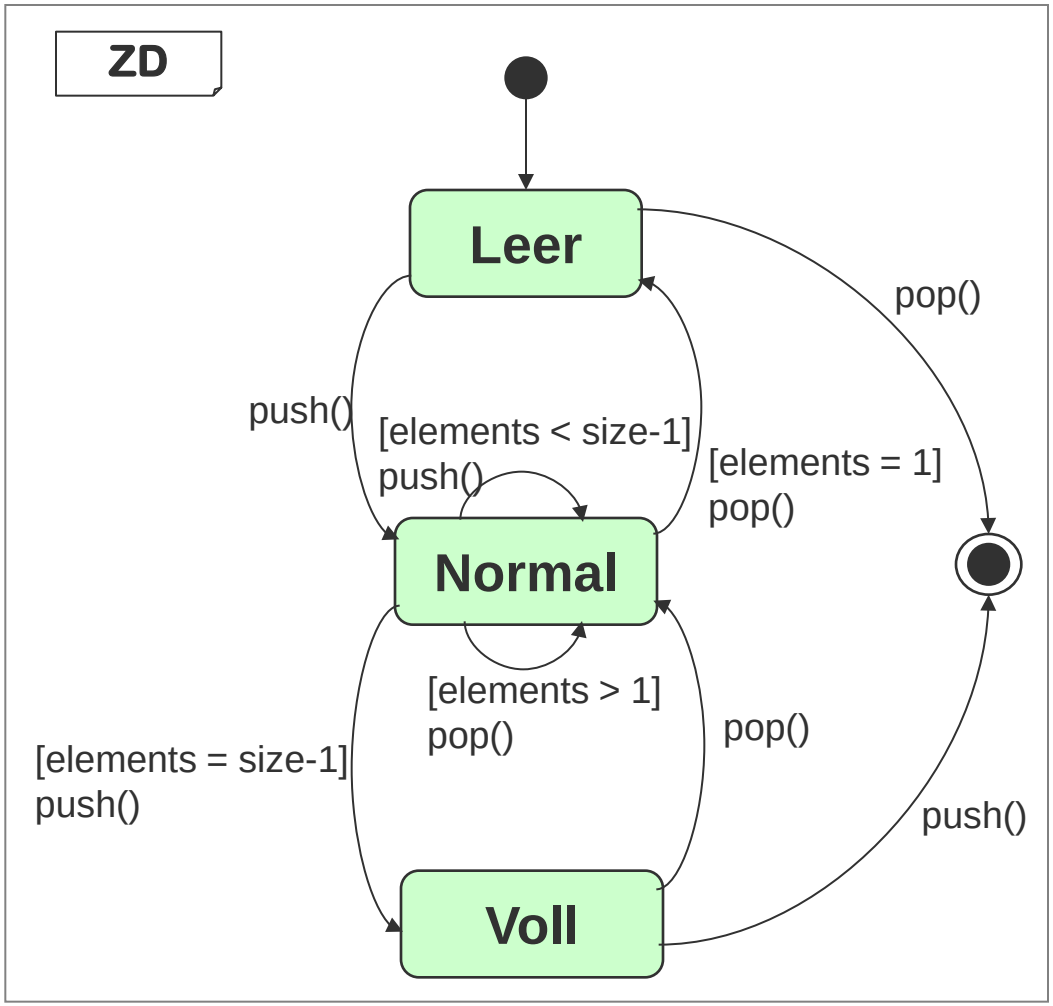
- Lightweight State Pattern vs. Heavyweight State Pattern (in german)

Behavioral Design Patterns

State Pattern Example: Stack

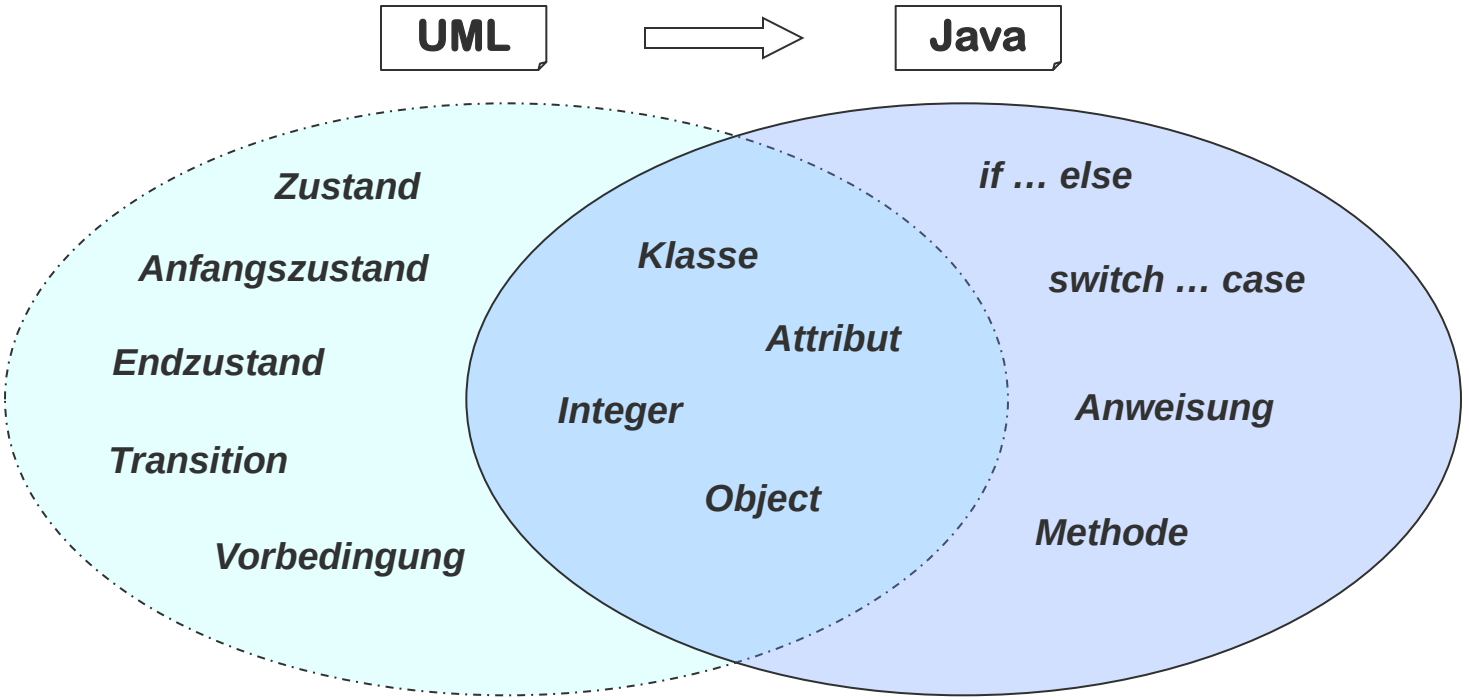
- LIFO Strategie
- Zwei Operationen
  - ***Push***: Gibt ein neues Element als oberstes Element auf den Stapel
  - ***Pop***: Liefert oberstes Element zurück und entfernt es vom Stapel
- Speicherplatz ist begrenzt





Behavioral Design Patterns

State Pattern Example: Implementation Problem?



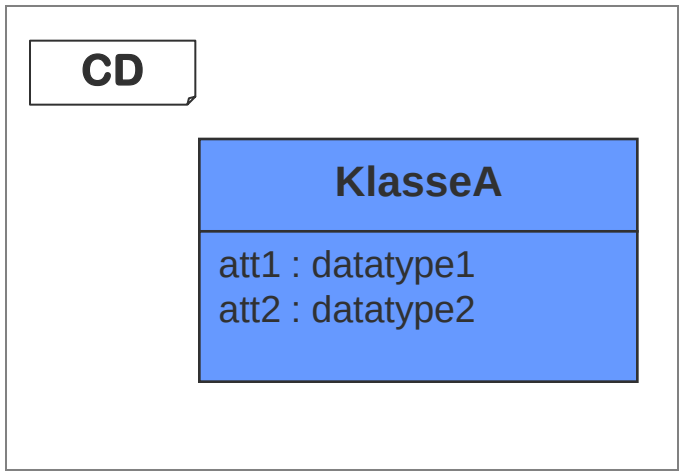
Behavioral Design Patterns

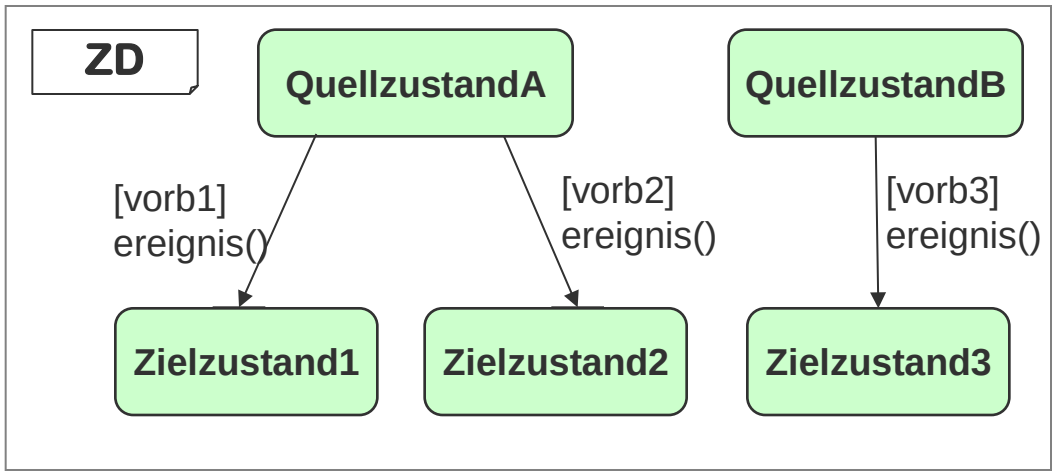
Lightweight: State as Attribute

- Zustände entsprechen Datenzuständen des Objekts
  - Diagrammzustand des Zustandsdiagramm muss aus dem Datenzustand des Objekts rekonstruierbar sein!
- Aufzählungsattribut als Speicher für den Zustand
  - Ermittlung des Diagrammzustands: Prüfen des *status-Attributs*
  - Für jeden *Zustand* wird eine *Konstante* (sogenannte *Zustandskonstante*) definiert
  - Ein *Ereignis* wird auf eine *Methode* abgebildet. In dieser Methode wird durch Fallunterscheidungen bestimmt in welchen Zustand sich das Objekt befindet
  - Zustandsübergang (Transition) ist die *Zuweisung* einer *anderen Zustandskonstanten* an das *status-Attributs*

Behavioral Design Patterns

Lightweight: State as Attribute





Java

```
public class KlasseA {

   Datatype1 att1;
   Datatype2 att2;



}
```

Java

```
private int status;
final static int QuellzustandA = 1;
final static int QuellzustandB = 2;
final static int Zielzustand1 = 3;
final static int Zielzustand2 = 4;
final static int Zielzustand3 = 5;
```

Java

```
public void ereignis() {
  switch( status ) {
    case QuellzustandA:
      if ( vorb1 ) {
        status = Zielzustand1;
      } else if ( vorb2 ) {
        status = Zielzustand2;
      }
      break;
      case QuellzustandB:
       …
}}
```

Behavioral Design Patterns

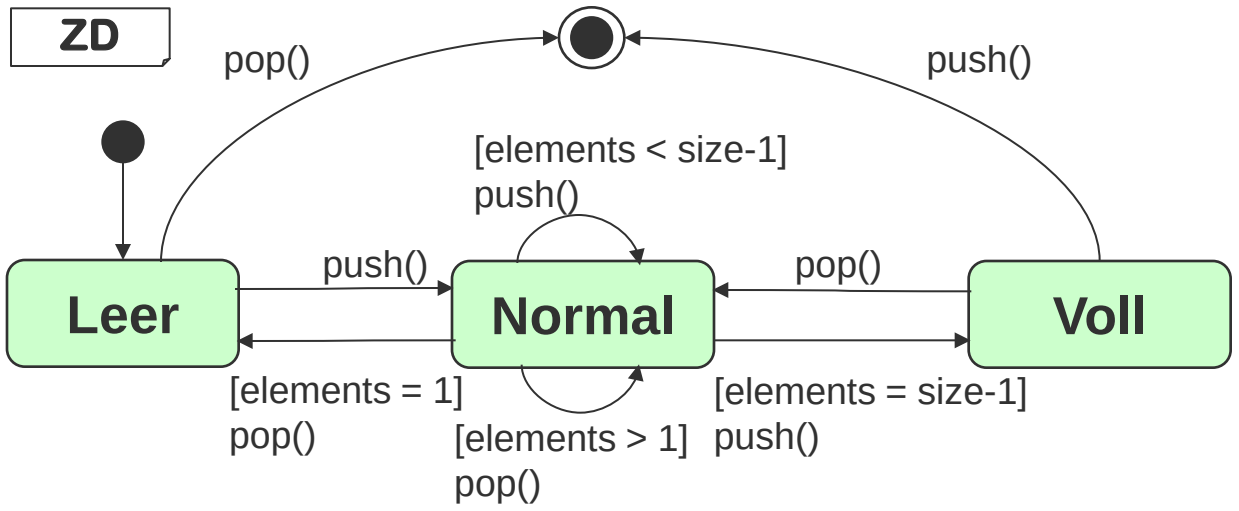
Lightweight: Stack Example

CD

Stack

size = 10 : Integer
elements = 0 : Integer
eintraege[size] : Object

ZD



Java

```
public class Stack {

  int size=10;
  int elements = 0;
  Object eintrage[] =
    new Object[size];

}
```

Java

```
private int status;
final static int anfang = 0;
final static int leer = 1;
final static int normal = 2;
final static int voll = 3;
final static int end = 4;
```

Java

```
public void push() {
   switch( status ) {
     case leer:
       status = normal;
     break;
     case normal:
       if (elements < size-1) {
         status = normal;
       } else if (elements = size-1){
         status = voll;
       }
     break;
     case voll:
       status = end;
}
```

Behavioral Design Patterns

Heavyweight: State as Class

- Lightweight: Einzelne Methoden abhängig vom Zustand des Objekts, d.h. Verhalten einer Methode abhängig vom Wert eines Attributes
- **Problem**: Bei großen Zustandsräume werden Methoden unübersichtlich und sind nur schlecht erweiterbar (z.B. Hinzufügen eines neuen Zustands)

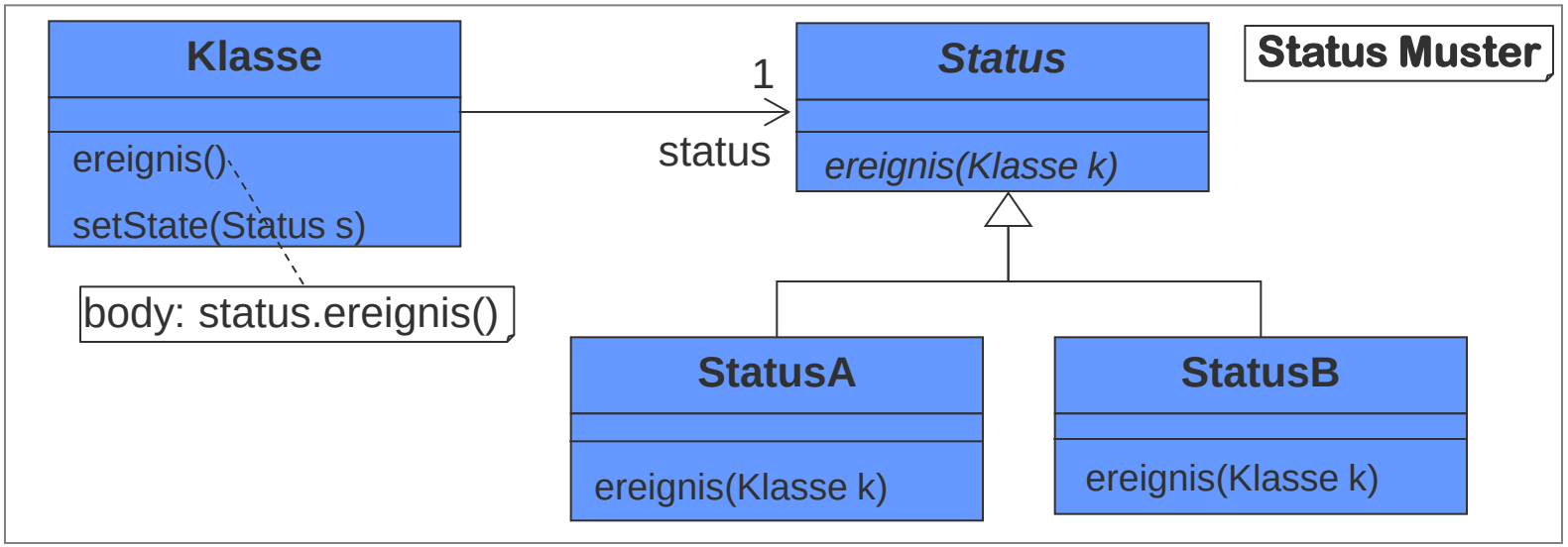
- Das *Status Entwurfmuster* (*State Pattern*) [1] eignet sich besonders gut für die Implementierung von Klassen, deren dynamischen Zustandswechsel mit Hilfe von Zustandsdiagrammen spezifiziert wurden

Behavioral Design Patterns

Heavyweight: State as Class

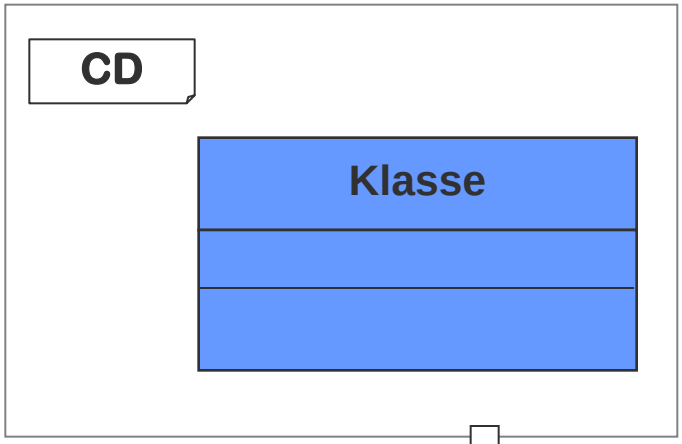
- Statusabhängiges Objekt **delegiert** zustandsspezifischen Operationen an ein Zustandsobjekt
- Zustandsobjekt ist indirekte Instanz der Klasse *Status*
- Für jeden Zustand wird eine konkrete Klasse von *Status* abgeleitet

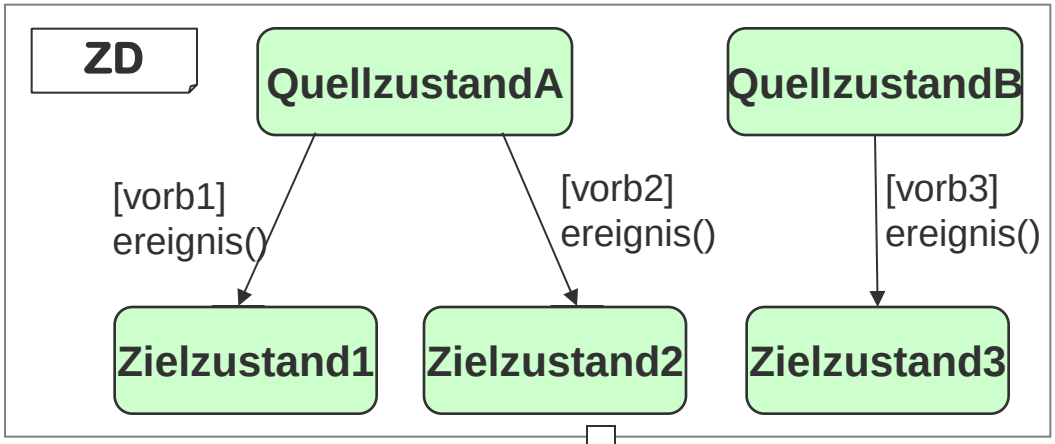
- **Vorteile**
  - Komplexe und schwer leserliche Bedingungsanweisungen können vermieden werden
  - Neue Zustände und neues Verhalten können auf einfache Weise hinzugefügt werden (erhöht die Wartbarkeit)
  - Zustandsobjekte können wieder verwendet werden

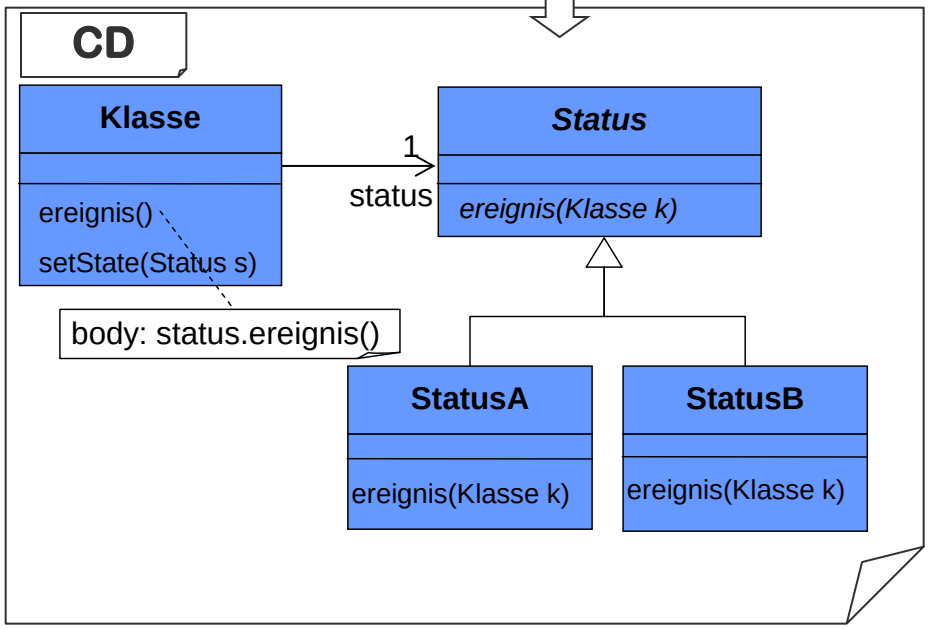


Behavioral Design Patterns

Heavyweight: State as Class





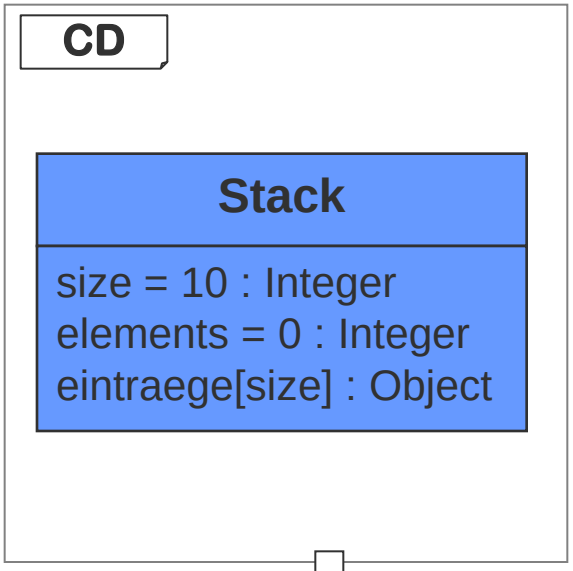


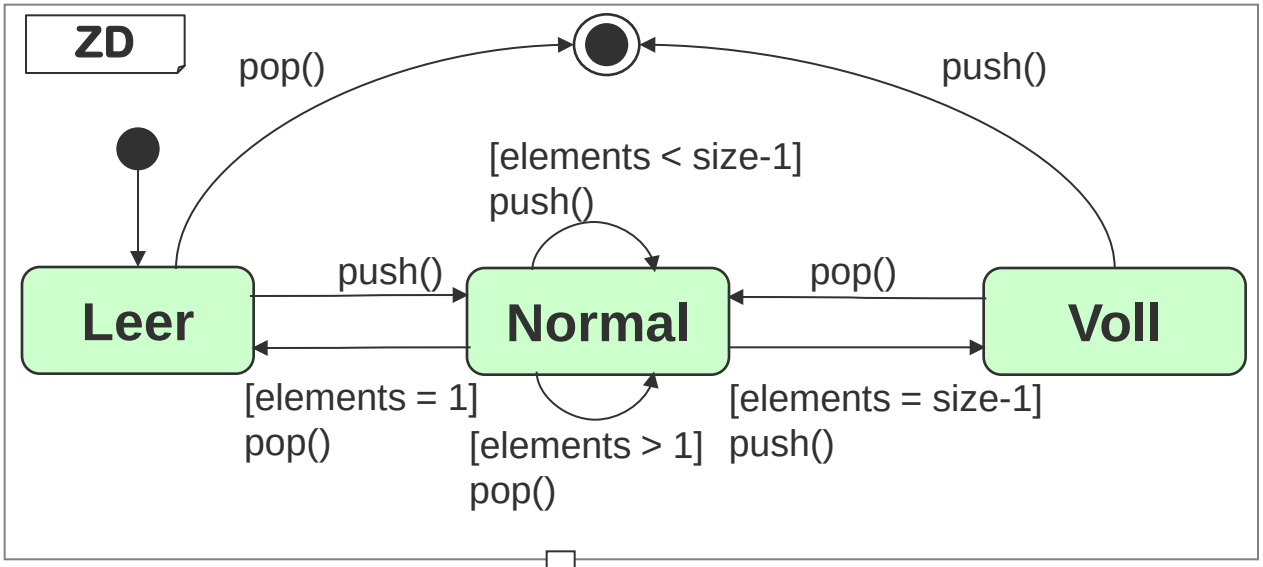
Java

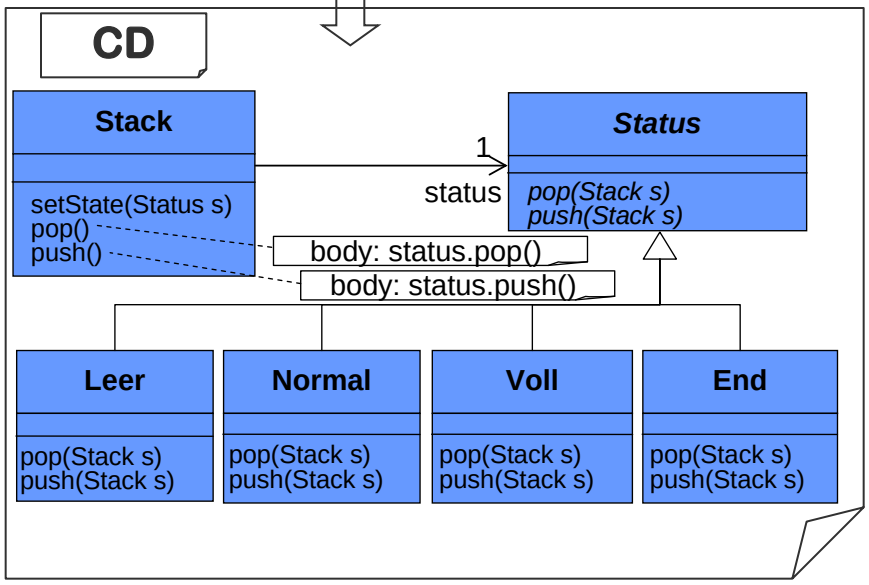
```
public class Klasse{
  public void ereignis(){
    status.ereignis(this);
  }
  Zielzustand1 zielzustand1 = …
  Zielzustand2 zielzustand2 = …
  Zielzustand3 zielzustand3 = …
}

public class QuellzustandA{
  public void ereignis(Klasse k){
    if (vorb1) {
      k.setState(k.zielzustand1);
    } else if (vorb2) {
      k.setState(k.zielzustand2);
    }
}
```

Behavioral Design Patterns

Heavyweight: Stack Example







Java

```
public class Stack{
  public void push(){
    status.push(this);
  }
  public void pop(){
    status.pop(this);
  }
  Anfang anfang = …
  Leer leer = …
  Normal normal = …
  Voll voll = …
  End end = …
}
```

```
public class Normal{
  public void push(Stack s){
    if (elements < size-1) {
      s.setState(s.normal);
    } else if { (elements = size-1)
      s.setState(s.voll);}
  }
  public void pop(Stack s){
    if (elements > 1) {
      s.setState(s.normal);
    } else if { (elements = 1)
      s.setState(s.leer);}
  }
}
```

Literature

Links

- Patterns Overview (german)

  http://wwwswt.informatik.uni-rostock.de/deutsch/Infothek/Entwurfsmuster/patterns/
- Huston Design Patterns
  Overview and Examples in Java and C++

  http://www.vincehuston.org/dp/
- J2EE Patterns
  Some patterns used in J2EE Core

  http://java.sun.com/blueprints/corej2eepatterns/Patterns/
- Web Presentation Patterns (MSDN)

  http://msdn2.microsoft.com/en-us/library/ms998516.aspx

Literature

Books

(1) Design patterns - elements of reusable object-oriented software; E. Gamma, R. Helm, R. Johnson, J. Vlissides (Group of Four); Addison-Wesley, 1995.

(2) Head First Design Patterns; E. Freeman et al.; O'Reilly, 2004.

(3) Java BluePrints: Model-View-Controller; Sun Microsystems; http://java.sun.com/blueprints/patterns/MVC-detailed.html, 2002.

(4) Design Patterns; Vince Huston; http://home.earthlink.net/~huston2/dp/patterns.html; 2003 – 2007.

(5) Patterns in Java; M. Grand; Wiley Publishing; 2002.

(6) Remoting Patterns – Foundations of Enterprise, Internet and Realtime Distributed Object Middleware; M. Völter, M. Kircher, U. Zdun; Wiley; 2005.