Design Patterns

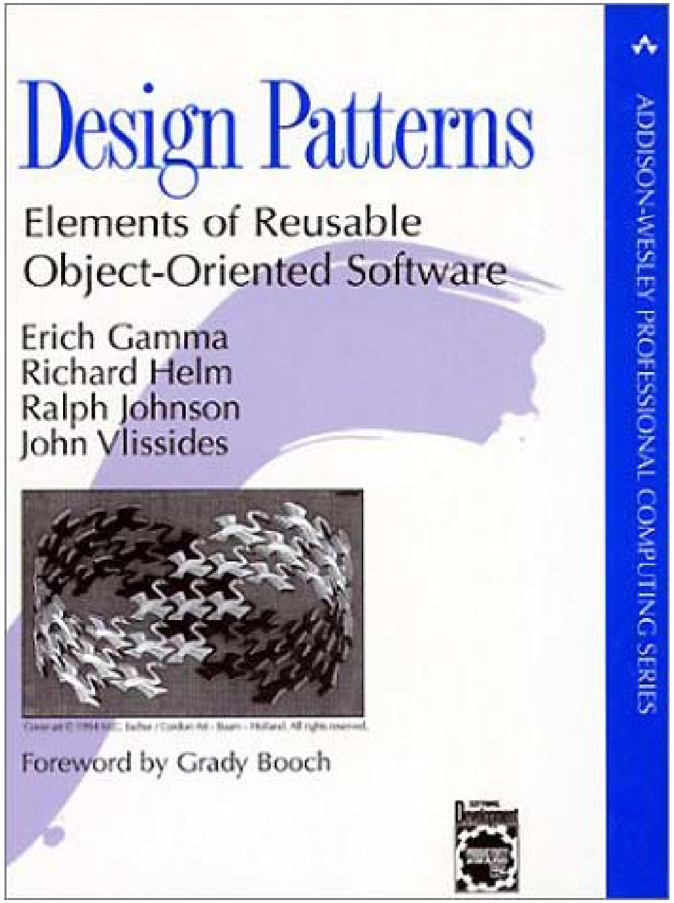
- Introduction
- What is a Design Pattern
- Benefits
- Expressing Design Pattern
- A Classification
- Selected Patterns

SWC SOFTWARE CONSTRUCTION

RWTH AACHEN UNIVERSITY

Design Patterns - History

- **E. Gamma at Zurich University**
  - Development of an Editor-Toolkit (ET)
  - Extension to the framework ET++
  - Description of design solutions for selected problems (Ph.D. thesis)
  - called "Design Patterns" according to C. Alexander et al.



- **Book "Design Pattern"**
  - gives a classification for design patterns
  - defines 23 design patterns

Design Patterns – Definition

- **A description of an object-oriented design technique which** names, abstracts **and** identifies aspects of a design structure **that are useful for creating an object-oriented design.**

- **A design pattern identifies** classes **and** instances**, their** roles**,** collaborations **and** responsibilities**. Each design pattern focuses on a particular object-oriented design problem or issue.**

- **It describes when it applies, whether it can be applied in the presence of other design constraints, and the consequences and trade-offs of its use."**

E. Gamma et al (1995): Design Patterns, Addison-Wesley, ISBN 0-201-63361-2

Motivation for Design Patterns

- **An architecture should be**
  - open and flexible
  - reusable, understandable, and changeable

- **Basic principles for applying Design Patterns:**
  - encapsulation → information hiding
  - loose coupling of components
  - strong coupling of components only on justification
  - right delegation of responsibility
  - perceiving responsibility and delegation
    - separation of concern

Classification (acc. Gamma et al.)

- **Scope**
  - *Class*:
    - static, defined through **inheritance** between classes
  - *Object*:
    - dynamically, defined through **associations** between objects
- **Purpose**
  - *Creational*:
    - Design Patterns for **creating** objects
  - *Structural*:
    - Design Patterns for building **complex objects**
  - *Behavioral*:
    - Design Patterns for accomplishing **complex tasks**

Design Patterns (acc. Gamma et al.)

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

Pattern Description – A structure

- **Name**
  - Characterizes the pattern
- **Problem**
  - Objective of the pattern
  - Synonyms of the pattern
  - Motivation and concrete scenarios or examples
  - Areas where to apply the pattern
- **Solution**
  - Basic structure of the pattern, described as packages, classes and interactions
  - Associated elements (other patterns, classes, …)
- **Consequences**
  - Advantages and disadvantages of the pattern usage
- **Implementation**
  - Techniques and pitfalls
  - Example implementation
  - at least two real life implementations
- **Related patterns**
  - Differences to other patterns

Pattern:
Factory Method

Factory-Method – Problem

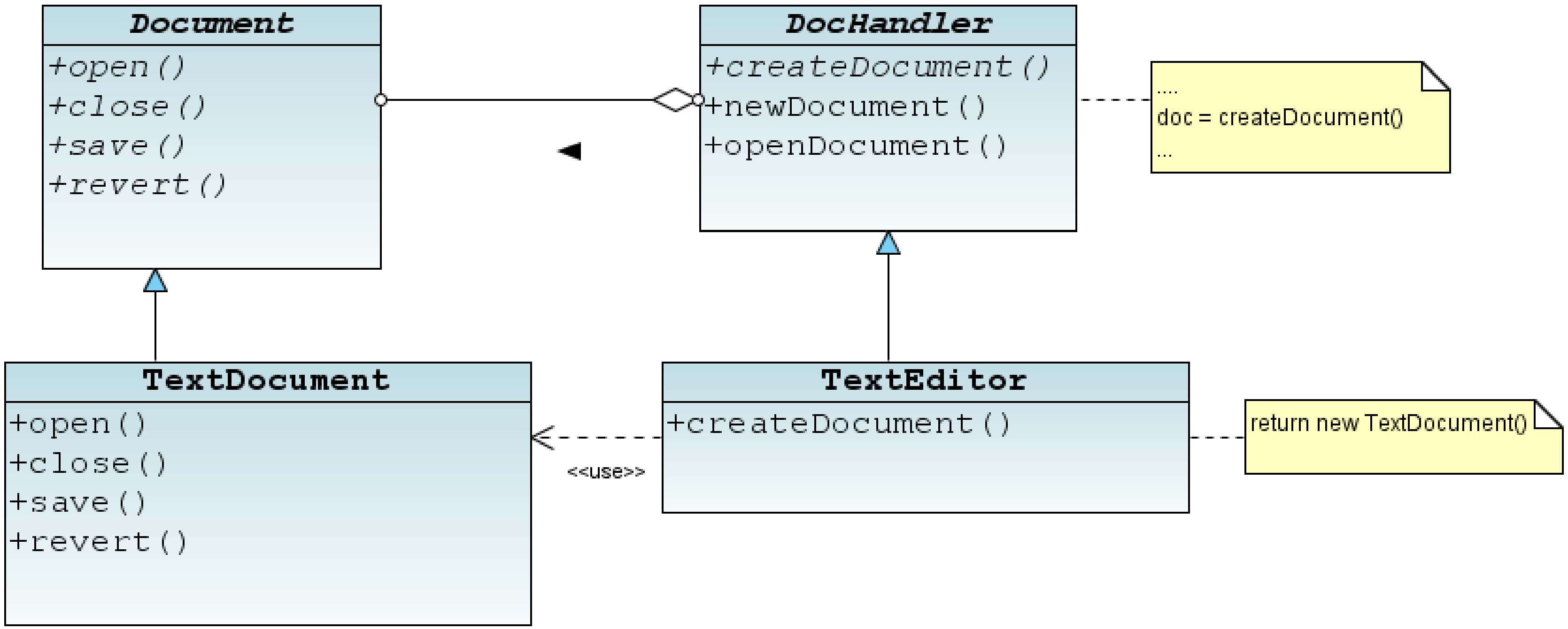
- **Problem**
  - Extendible applications (often frameworks) typically provide abstract classes (so called hot spots)
  - These abstract classes must allow to create concrete objects anyway
  - The classes of those concrete objects are only known when the framework is extended

- **Example**
  - An framework for graphical editors must be able to create objects of
    - rectangles, circles, lines
    - user defined shapes

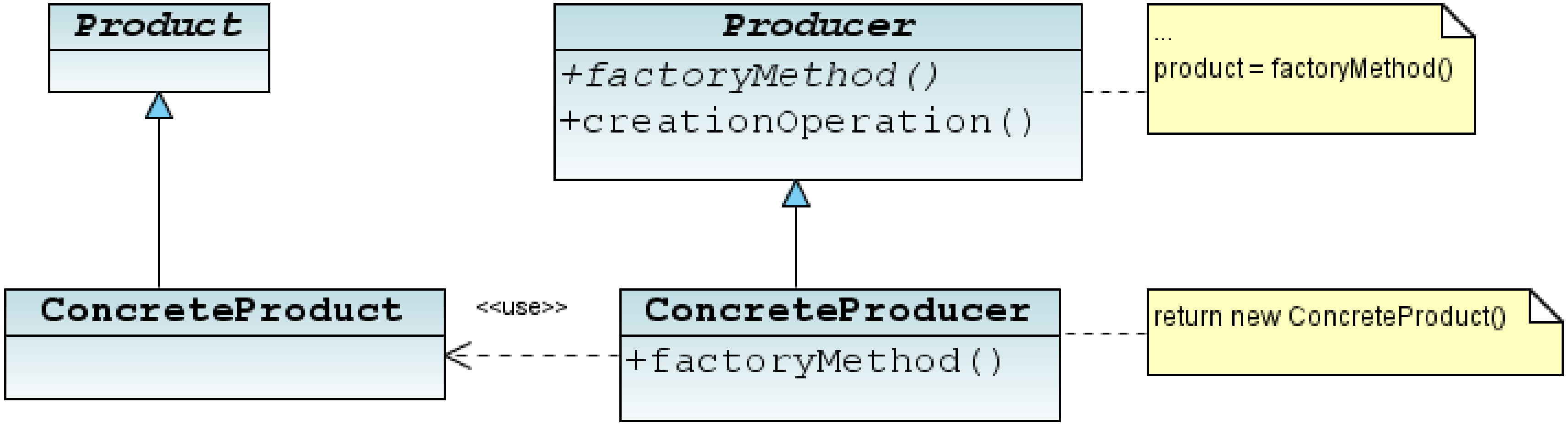
Factory-Method – Solution

- **Solution**
  - Abstract classes define an abstract creation method
  - This method must be redefined by concrete subclasses later
  - Therefore subclasses are responsible for the final creation

Factory-Method – Example



Factory-Method – Structure



Factory-Method – Discussion

- **Advantages**
  - the pattern contains no product specific classes
  - frameworks may be designed domain-independent
  - no coupling between product and producer
  - no coupling between concrete class and framework behavior
  - higher flexibility for creating objects

- **Disadvantages**
  - overhead for defining and implementing subclasses
    (especially if only one product is needed)

Factory-Method – Remarks

- **The Producer may implement a default creation behavior in the factory method**
- **Factory methods may be parameterized for creating different products**
  - this may save special factory methods
  - subclasses may add variety or change existing creation behavior
- **Alternatives**
  - using a generic creator class which takes the (concrete) product class as a parameter
  - from this generic class concrete creator classes can be derived

Pattern: Singleton

Singleton – Problem / Solution

- **Problem**
  - for a specific class exactly one instance shell be created
  - this instance must be globally accessible
  - e.g. printer spooler, file system, window manager

- **Solution 1**
  - Access this instance through a global variable
  - This doesn't prevent creation of multiple instances

- **Solution 2**
  - Define class variables / methods only
  - No inheritance possible (e.g. in Java)

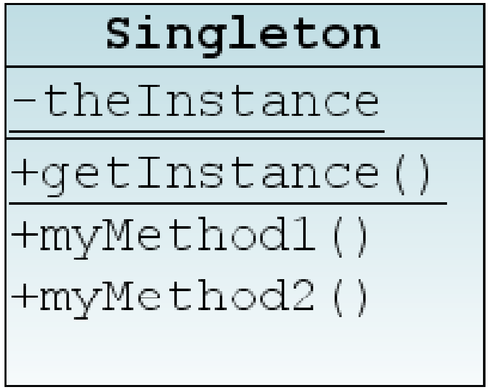
Singleton – Solution

Solution

- The class is responsible for the existence of only one instance
- The class is globally accessible
- The class gives access to the (single) instance

Realization structure

- The instance is hold by the private class variable `theInstance`
- The instance can be accessed by the public class method `getInstance()`
- call `Singleton.getInstance()`
- The constructor has be declared `protected`



Singleton – Implementation

```
class Singleton {

      static private Singleton theInstance = null;

      static public Singleton getInstance() {
            if (theInstance == null)
                  theInstance = new Singleton();
            return theInstance;
      }
      /* no public constructor allowed
         for usage in subclasses declared as
         “protected” */

      protected Singleton() { ... }

      ... // instance variables and methods here
}
```

Singleton – Remarks – 1

- **Sub-classing can be done but `getInstance()` has to be re-implemented**
- **Alternative**
  - applying Factory Method pattern

```
class NewSingleton extends Singleton {

	static public NewSingleton getInstance() {
		if (theInstance == null)
			theInstance = new NewSingleton();
		return theInstance;
	}

	protected NewSingleton() {
		super(); ...
	}
	... // instance variables and methods here
}
```

Singleton – Remarks – 2

- **Alternatives**
  - `getInstance()` may decide on global information which class should be instantiated
    - properties, environment variables, …

- **If a lot of singletons are needed a registry may be useful**
  - singletons my be accessed through a key
  - each singleton instance registers during construction

Singleton – Discussion

Advantages

- controlled access to one instance
- instance is created when needed
- redefinition of singletons is possible
- may be extended to exactly N instances (pooling)

Disadvantages

- reintroduction of (the concept of) global variables through the backdoor!

Exercise D

Design Patterns (acc. Gamma et al.)

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

Pattern:
Adapter

Adapter – Problem

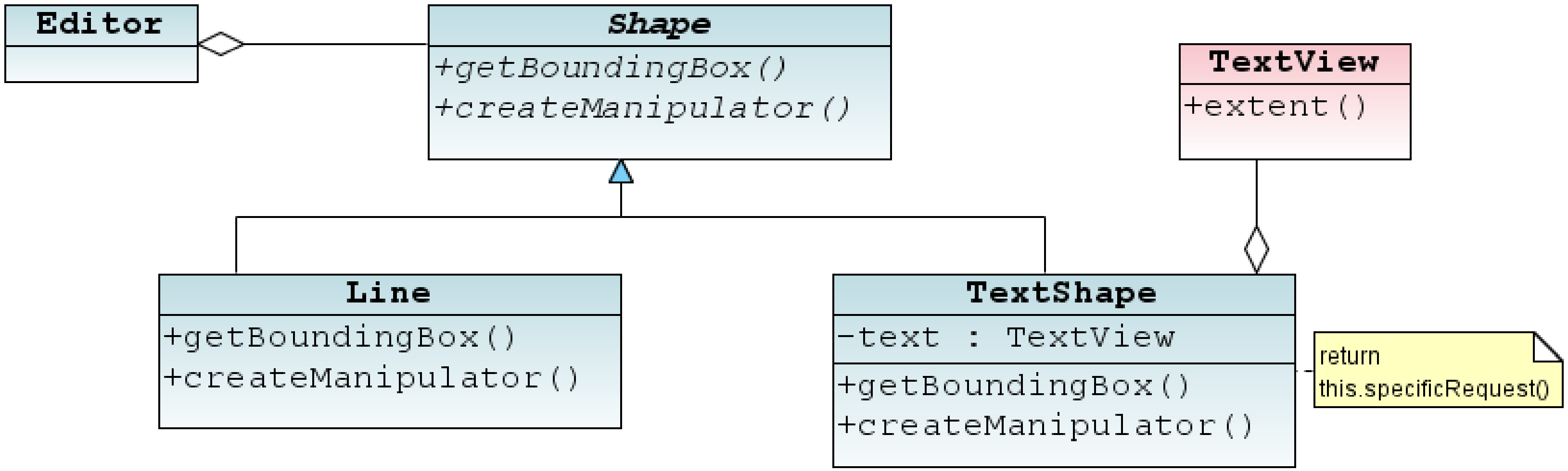
Problem

- One class ought to use another class – but the interfaces don't fit
- This mainly happens when combining or reusing different classes or class libraries

Possible solutions

- Change the source code (if available)
- Make a subclass of the class to be re-used and redefine interfaces

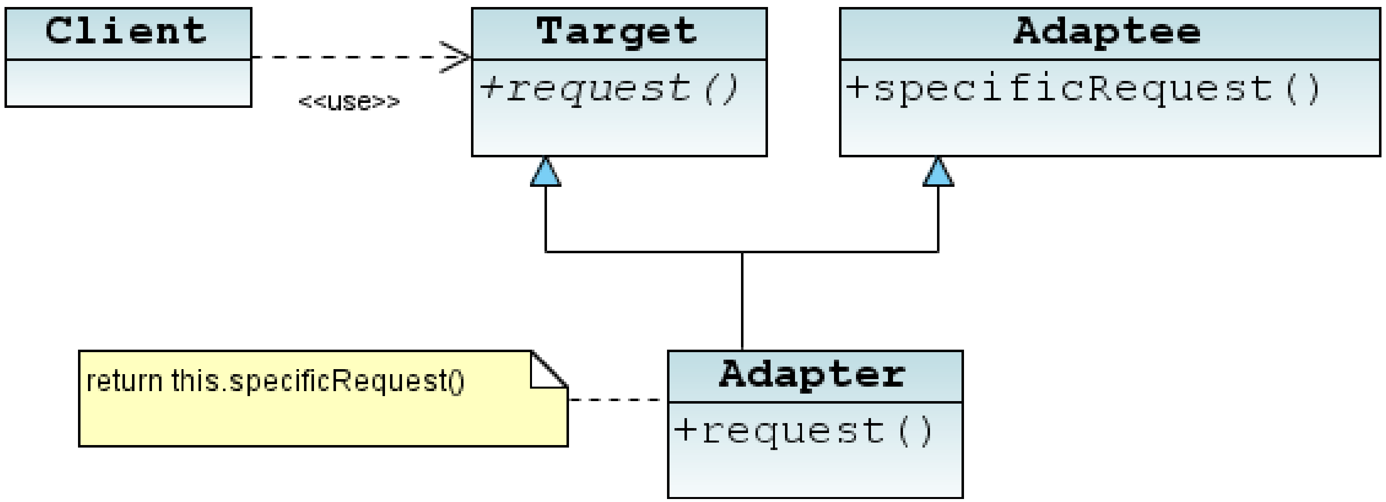
Adapter – Example



Adapter – Class Variant

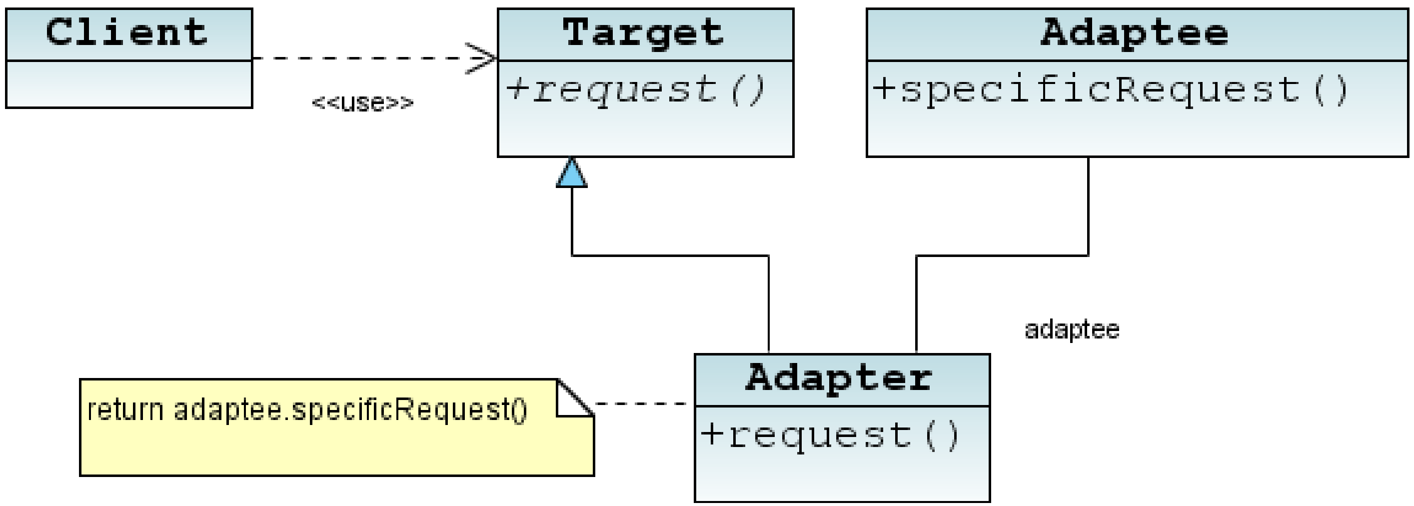
Solution

- Create a subclass of the target class and of the adaptee class (→ multiple inheritance must be provided)



Adapter – Object Variant

- **Solution**
  - Create a subclass of the target class which uses an object of the adaptee class and delegates the calls



Adapter – Discussion

Class Adapter

- The adaptee is adapted exactly to one target class
  - subclasses of the adaptee are not adapted
- Changes (adapts) the behavior of the adapted class by subclassing
- uses only one object at run-time

Object Adapter

- One adapter class adapts the adaptee class and all its subclasses
- Only slight changes of adaptee-behavior possible

Exercise E

Pattern: Composite

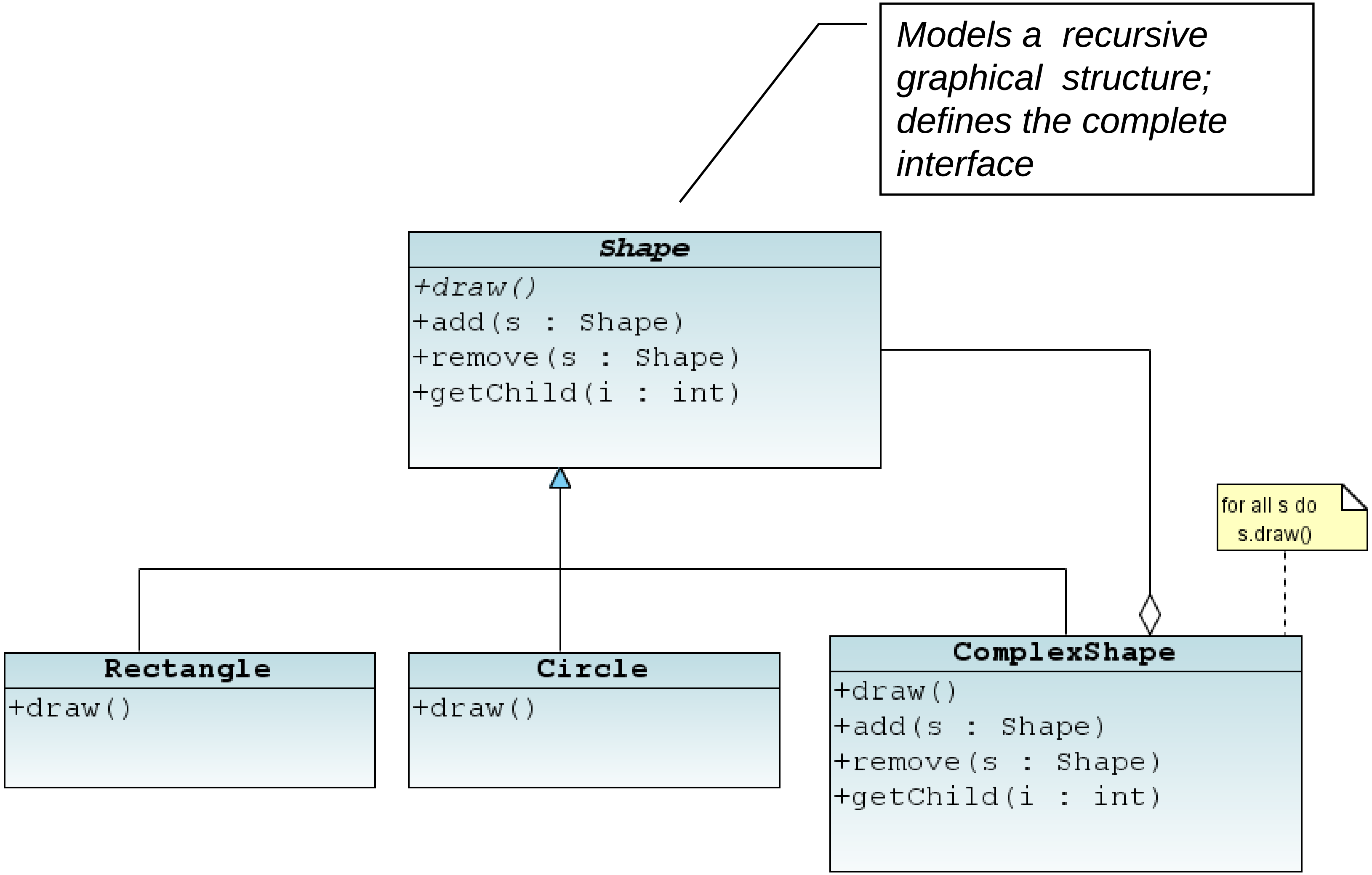
Design Patterns (acc. Gamma et al.)

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

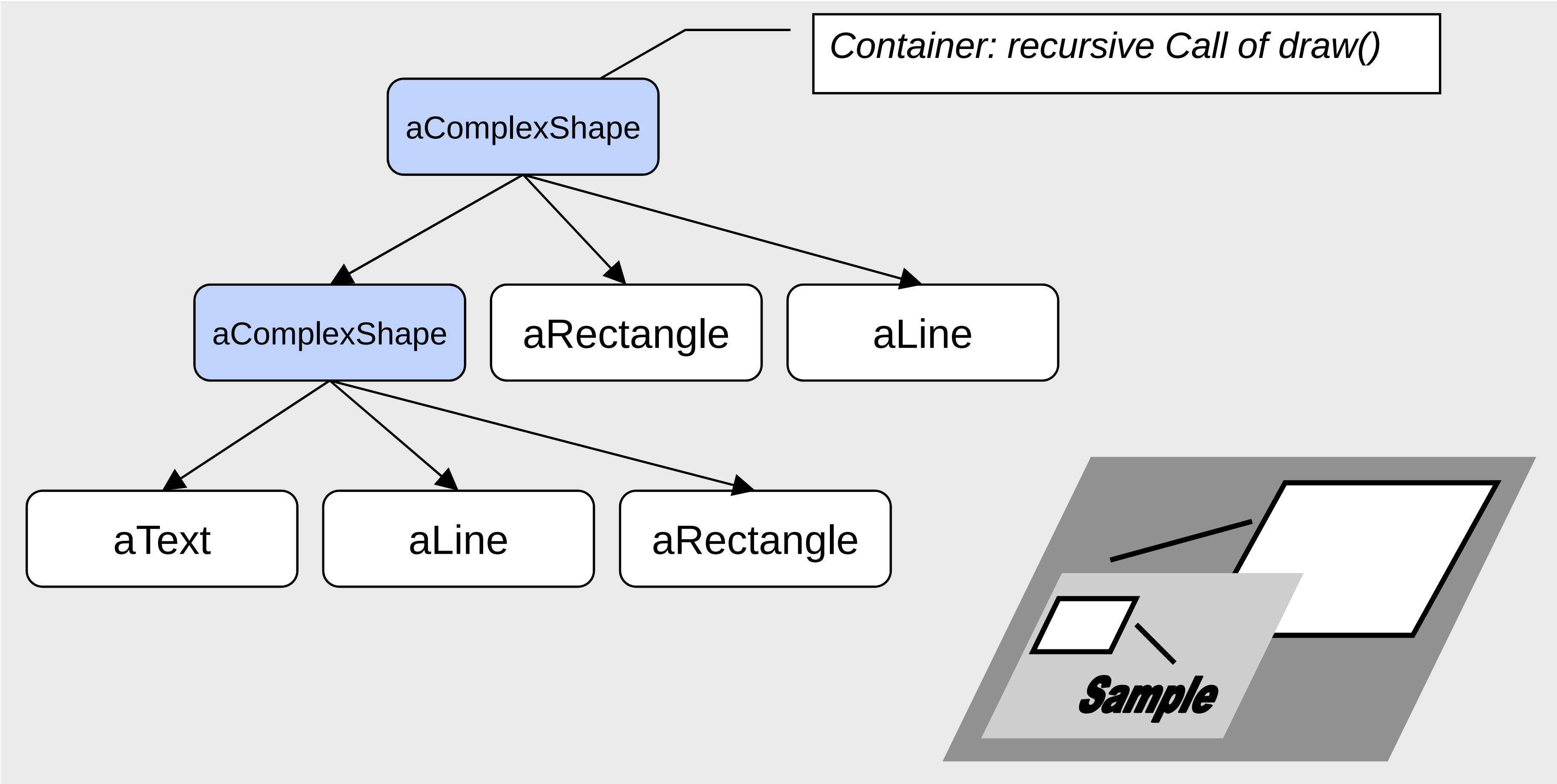
Composite – Problem / Solution

- **Problem**
  - Modeling a Part-Whole-Relationship
  - Parts (primitives) and composite objects (container) should provide the same interface
- **Solution**
  - Define a common superclass for primitives and containers
- **Examples**
  - all kinds of tree-structures:
    - Parse-Trees,
    - grouped graphical objects,
    - GUI-Elements

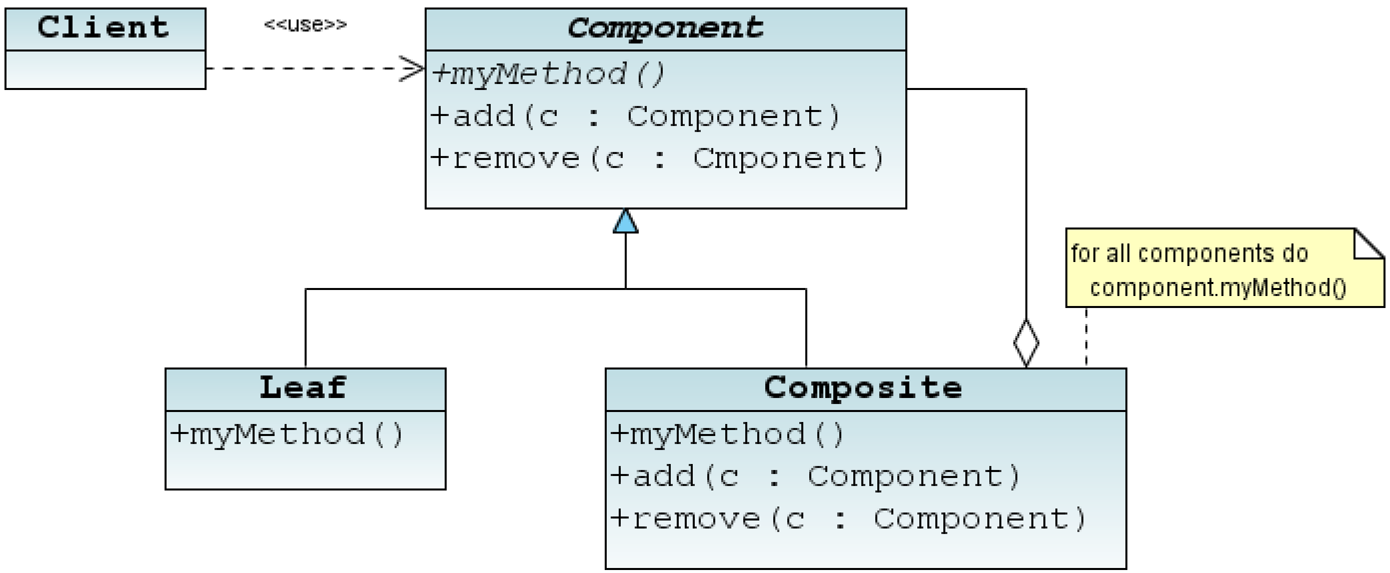
Composite – Example – 1



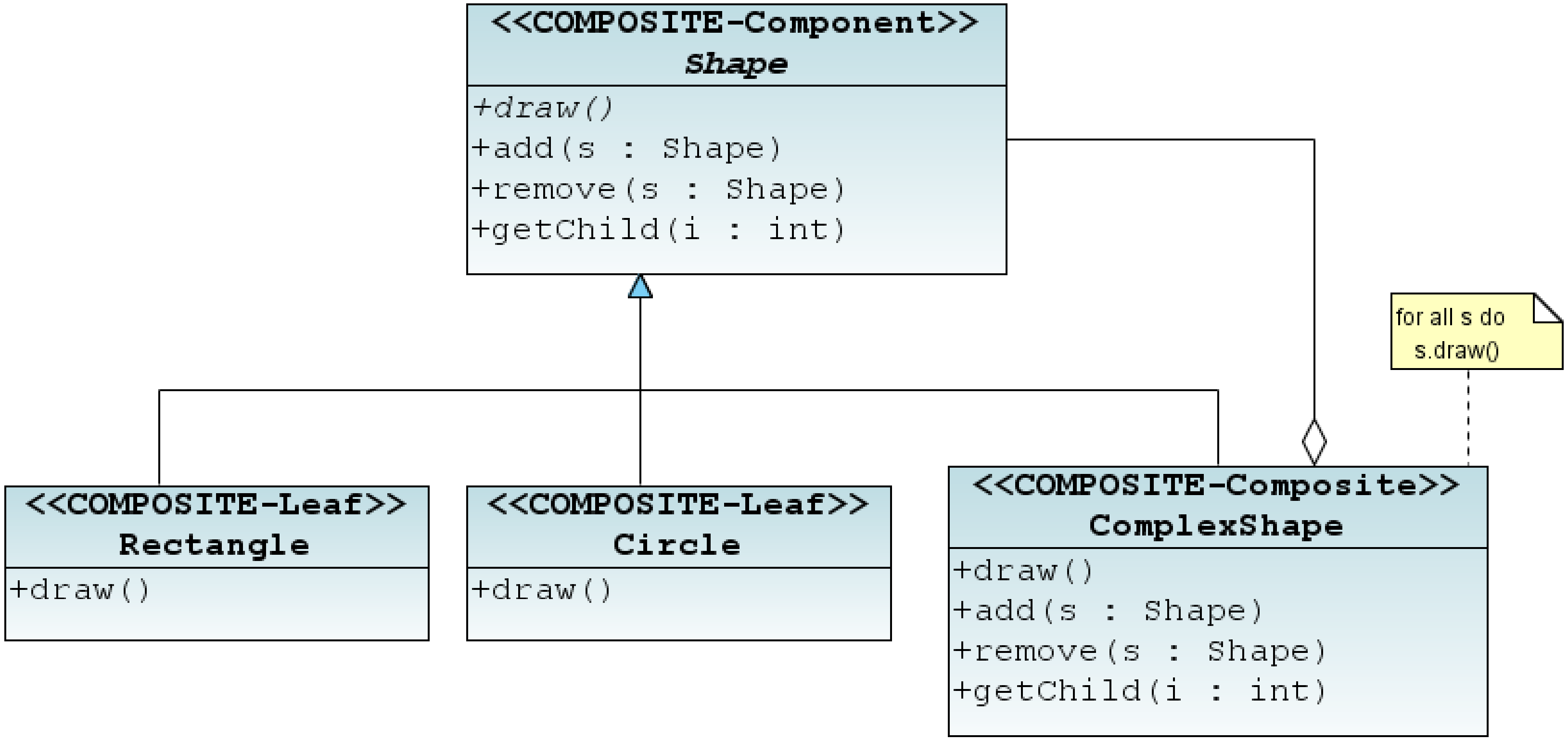
Composite – Example – 2



Composite – Structure



Composite – Example with roles



Composite – Remarks

- **Composite**
  - usually delegates method calls to all parts
  - may implement additional operations
  - the add() / remove-methods may only be used by composites

- **Composite-Operations**
  - may be defined in Component
    - uniform interface
    - implements a default behavior in Component (e.g. exception handling)
    - must not be used by primitives
  - may be defined in Composite
    - higher security

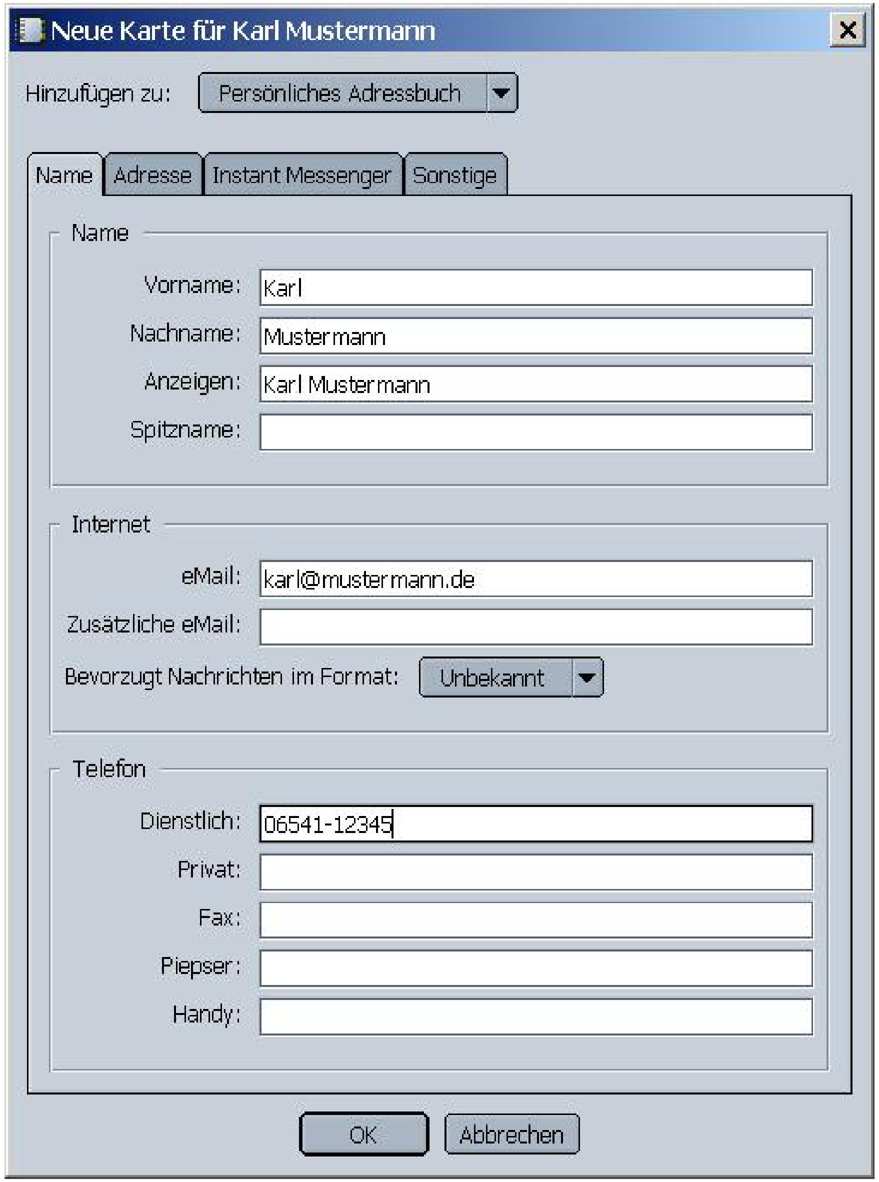
Composite – Discussion

- **Advantages**
  - same interface for primitives and composites
  - simple usage
  - easily extendable for new primitives
  - recursive composition of objects

- **Disadvantages**
  - Composite-operations in Component class
  - Restriction on primitives is a very unusual form of applying inheritance

Pattern: Strategy

Strategy – Motivation

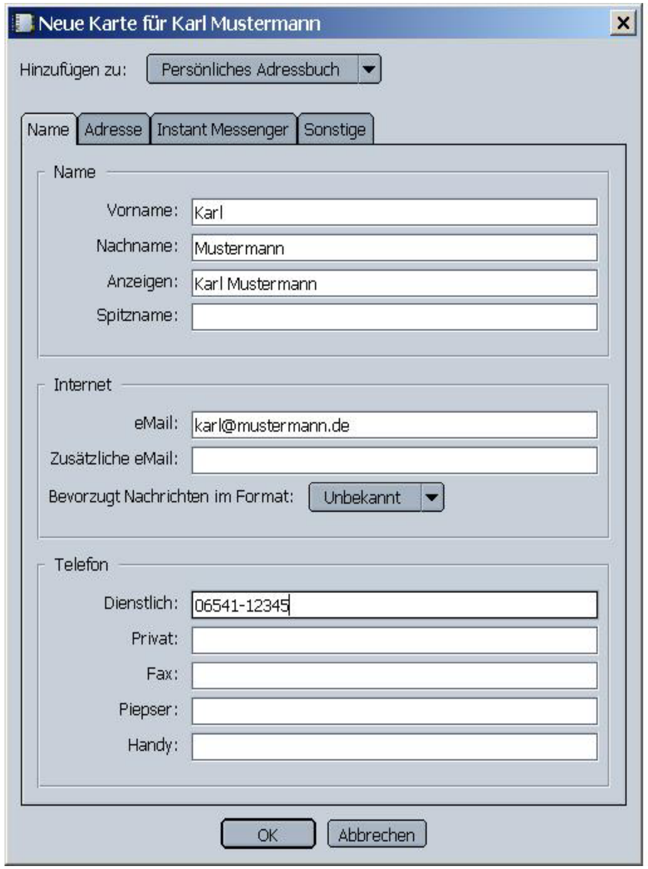


- An input form has to check the syntax of the input
  - e.g. integer number,
  - ZIP code,
  - e-mail,
  - phone number

Strategy – Solutions

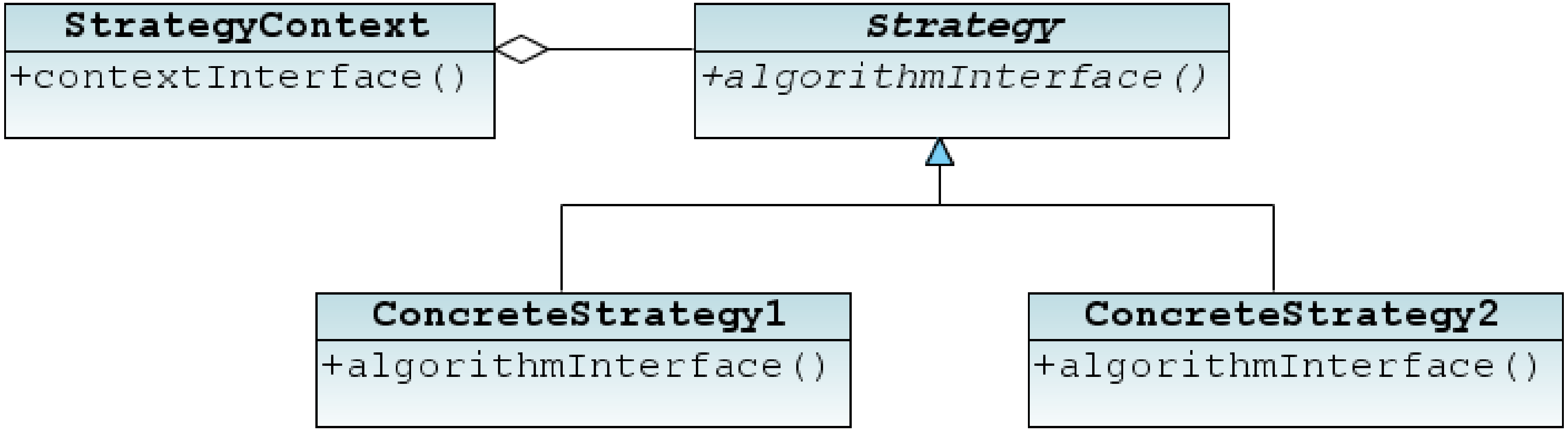
- **Solution 1: monolithic**
  - the whole input is submitted to a special method that checks syntax conformity of all fields
  - problem: bad changeability when changing the form or the syntax of fields, code duplication
- **Solution 2: factorizing**
  - methods for checking the input are implemented in special check methods used by the submit-method
  - problem: bad changeability when changing the form
- **Solution 3: specializing of `TextField`**
  - Special subclasses of TextField with check functionality are implemented (e.g. EmailTextfield) and used
  - problem: lots of very small classes

Strategy – Basic idea



- Introduction of special checking classes for checking the syntax
- Each TextField object becomes associated with a checking object and delegates checking
- The checking object may be exchanged during runtime, therefore checking syntax may change during runtime (e.g. key word, signature in a library search)

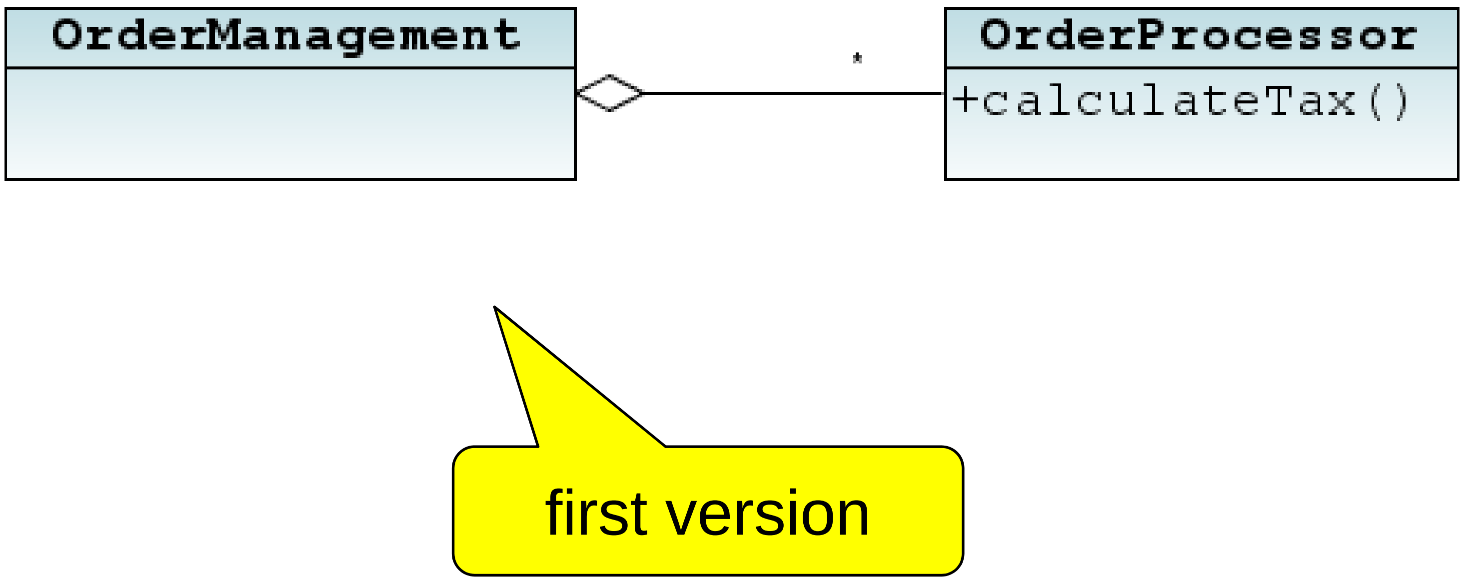
Strategy – Structure



- `StrategyContext.contextinterface()` delegates specific tasks to `Strategy.algorithmInterface()`
- only the object itself (`this`) or needed data are handed over → low coupling

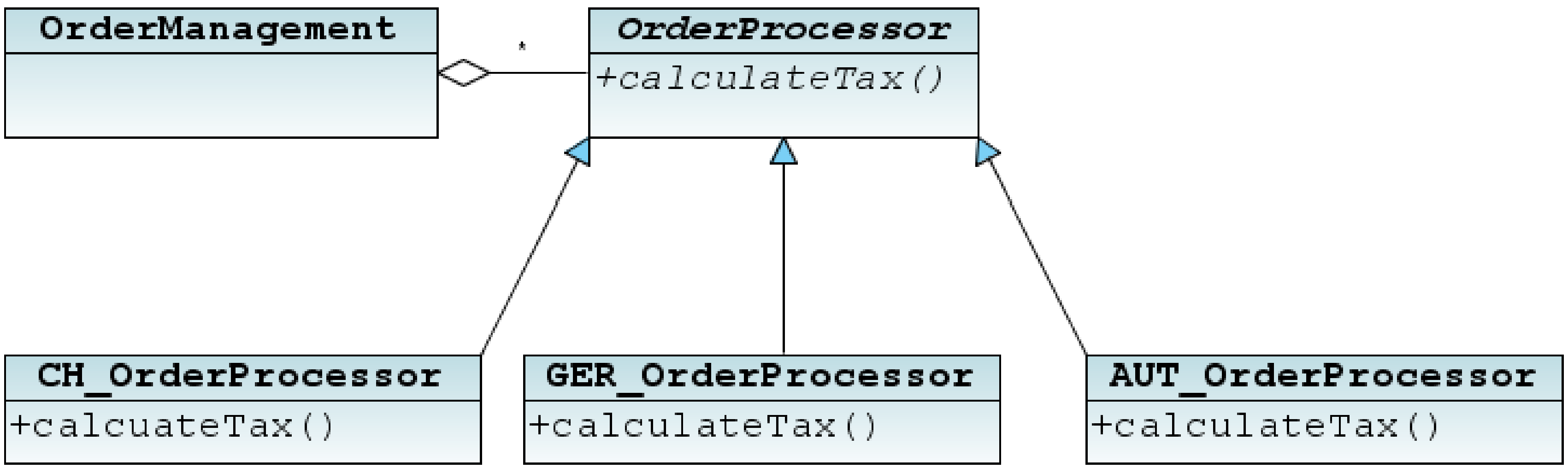
Strategy – Example - 1

- **A simple OrderManagement-System**
  - Processing of orders is delegated to `OrderProcessor` objects
  - `calculateTax()` returns the VAT of the order



Strategy – Example - 2

- **New requirement**
  - The system should process orders from abroad
    - e.g. Switzerland, Austria

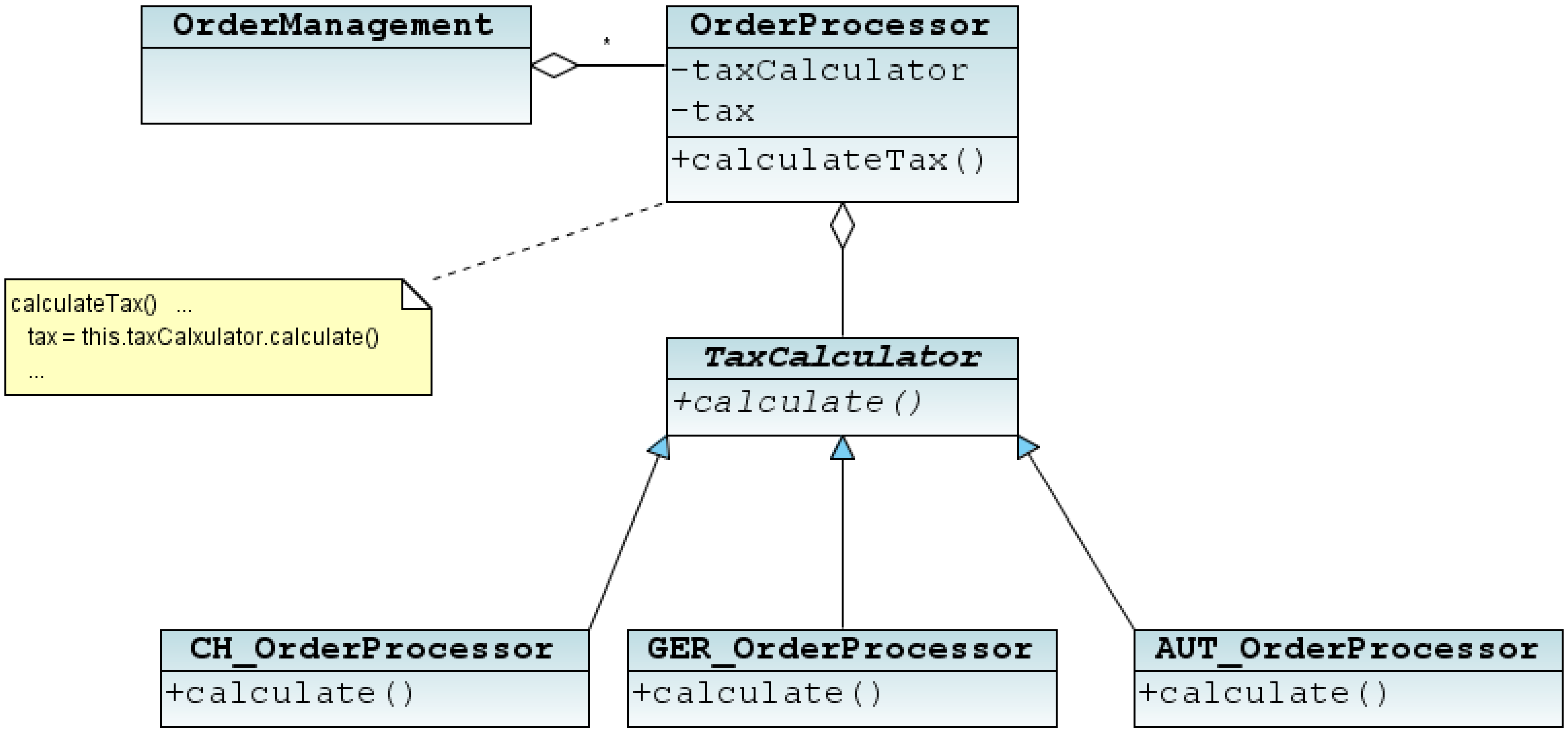


Alternative 1

Strategy – Example - 3

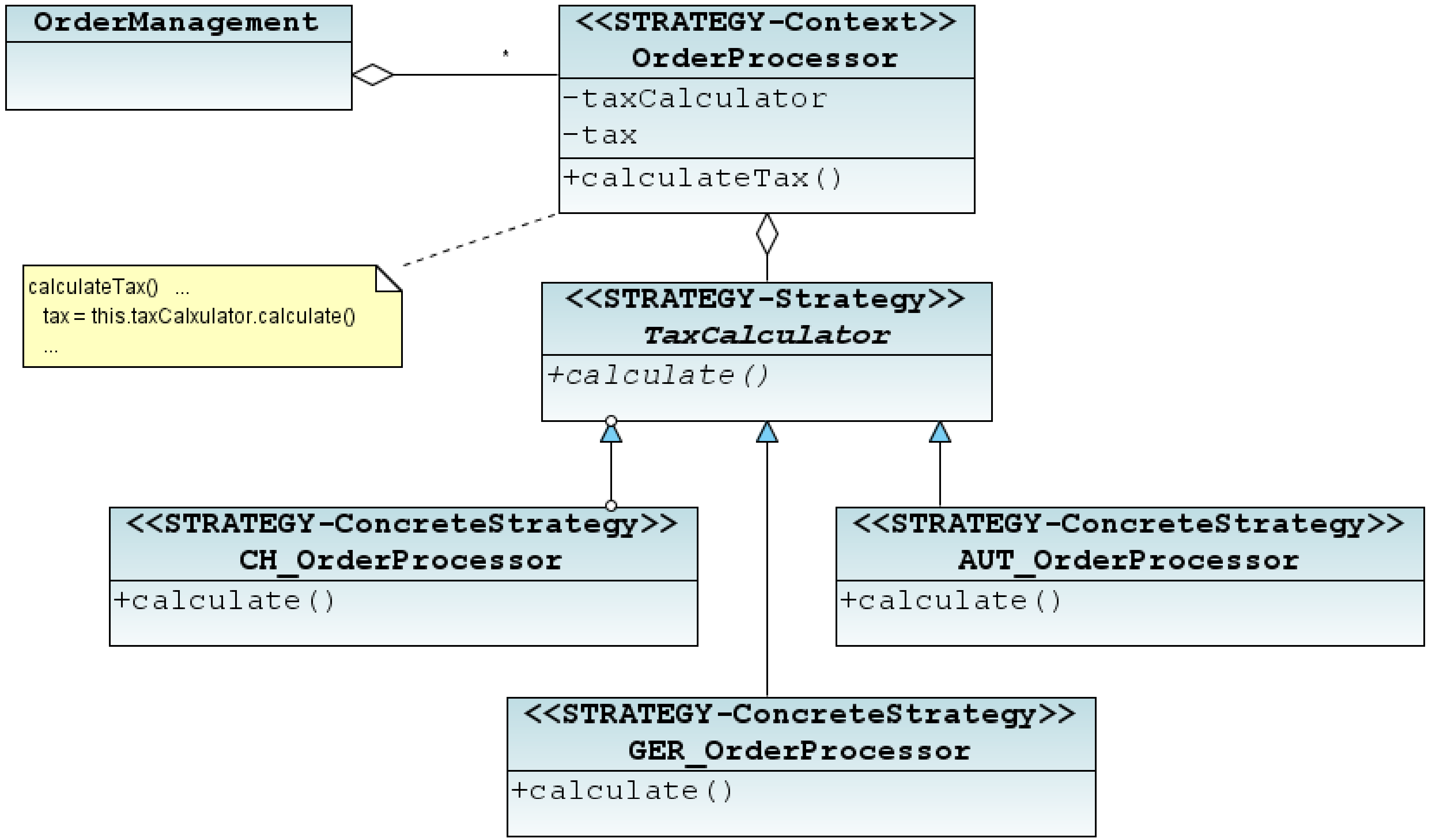
Applying the Strategy Pattern

- means applying the separation of concerns principle



Strategy – Example - 4

- **Mapping classes to pattern roles**



Strategy – Usage

- **Classes only differ in their behavior**
  - classes may be configured with different behavior
- **Different variants of an algorithm are needed**
  - implementation in different classes
- **The algorithm should be hidden**
  - encapsulation of the algorithm in a class
- **The behavior of a class may change at runtime (from outside the class)**

Strategy – Discussion

Advantages

- Strategy hierarchies may implement related algorithms
- Different implementations of the same behavior (e.g. time or space oriented)
- Classes may be configured by strategies → users have to know about the strategies

Disadvantages

- Communications overhead between context and strategy
- More objects and classes

Exercise F

Design Patterns (acc. Gamma et al.)

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

Pattern:
Template Method

Muster: Template Method

Problem

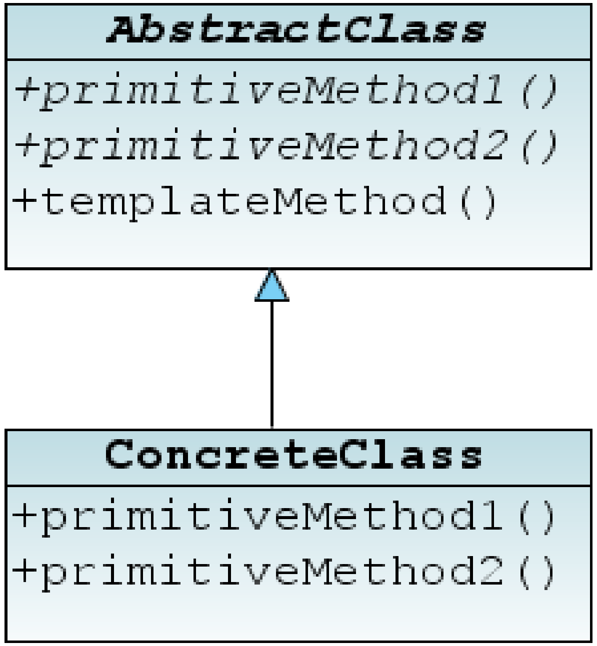
- Implementation of an algorithm with variants
- New variants should be added easily
- Minimal code duplication

Solution

- The common parts of the algorithm → template method
- Variant parts implemented by new methods → called by template method
- All variants have to follow the same schema

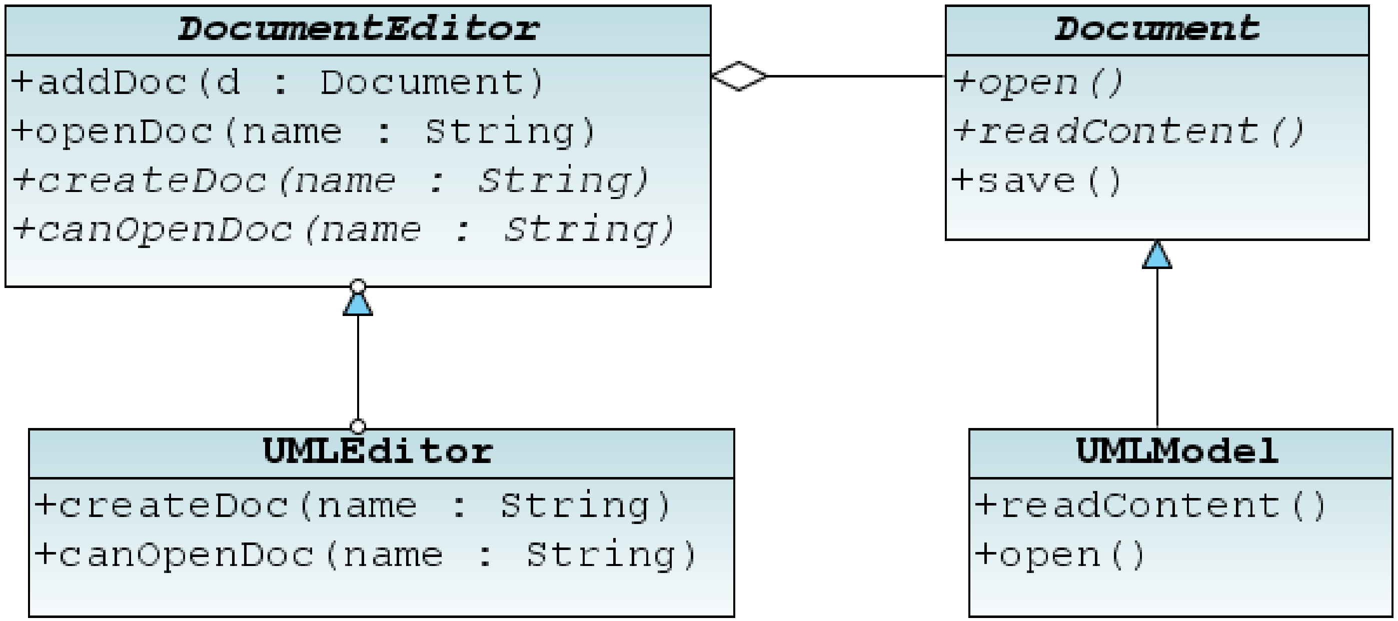
Template Method – Structure

- Implementation of algorithm frame in the concrete method `templateMethod()`
- Implementation of variable parts in abstract methods `primitiveOperationX()` called by `templateMethod()`
- A concrete subclass implements all primitive methods
- The primitive methods are typically only used inside → protected
- → "hook"-mechanism



Template Method – Example – 1

- An application may have several accociated documents.
- **openDoc** opens a application-specific document



Template Method – Example – 2

- **Implementation of `openDoc` in class `DocumentEditor`**

```
openDoc(String name) {
      if (!canOpenDoc(name))
            return; // problems opening the document
      Document doc = this.createDoc(name);
      if (doc != null) {
            this.addDoc(doc);
            doc.open();
            cont = doc.readContent();
      }
}
```

Template Method – Remarks

- **A main pattern for framework design**
  - inversion of control: „Don't call us, we'll call you" → superclass calls methods of subclasses
- **Primitive operations need not to be abstract but may implement default behavior**
- **The number of primitive operations should be as minimal as possible**
  - Narrow Inheritance Interface Principle
- **The algorithm should be generally parameterized**

Pattern:
Iterator

Design Patterns (acc. Gamma et al.)

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

Iterator – Problem / Solution

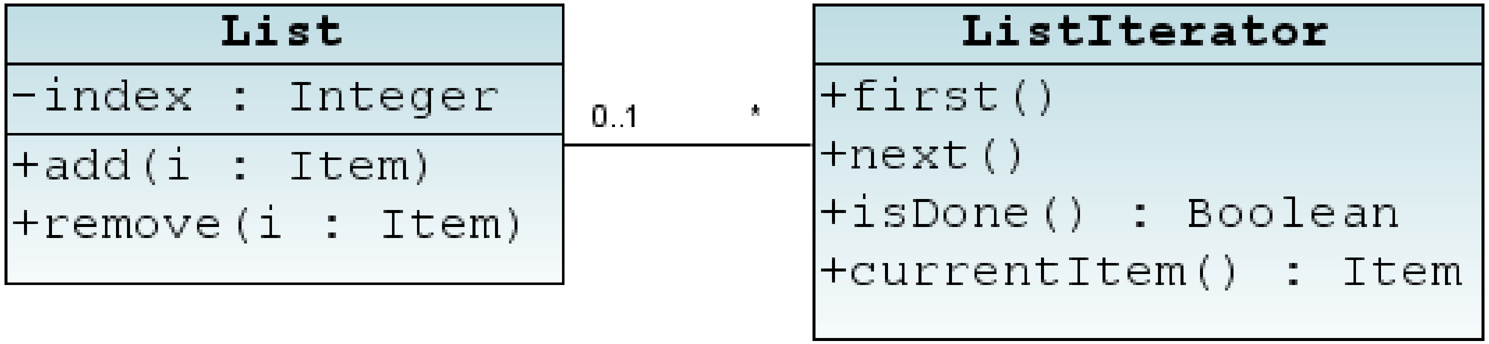
Problem

- An aggregate object such as a list should allow a way to traverse its elements without exposing its internal structure
- It should allow different traversal methods
- It should allow multiple traversals to be in progress concurrently
- But, we really do not want to add all these methods to the interface for the aggregate

Solution

- Separation of data structure and iterator

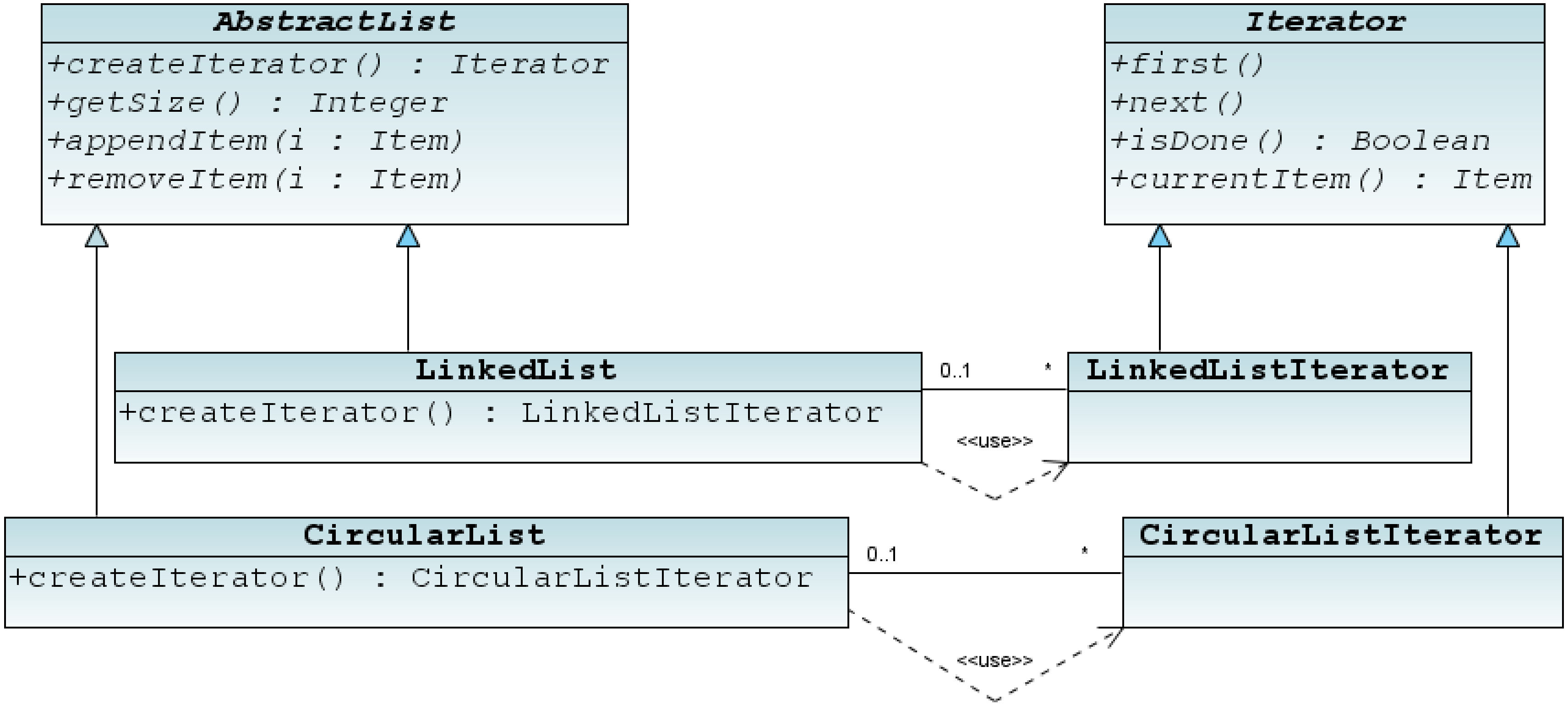
List aggregate with iterator




```
...
List myList = new List();
   ...
   ListIterator iterator = new ListIterator(myList);
   iterator.first();
   while (!iterator.isDone()) {
      Item item = iterator.currentItem();
      // Code here to process item.
      iterator.next();
   }
...
```

Iterator – Polymorphous Iteration

- Different implementations of a data structure have the **same interface**

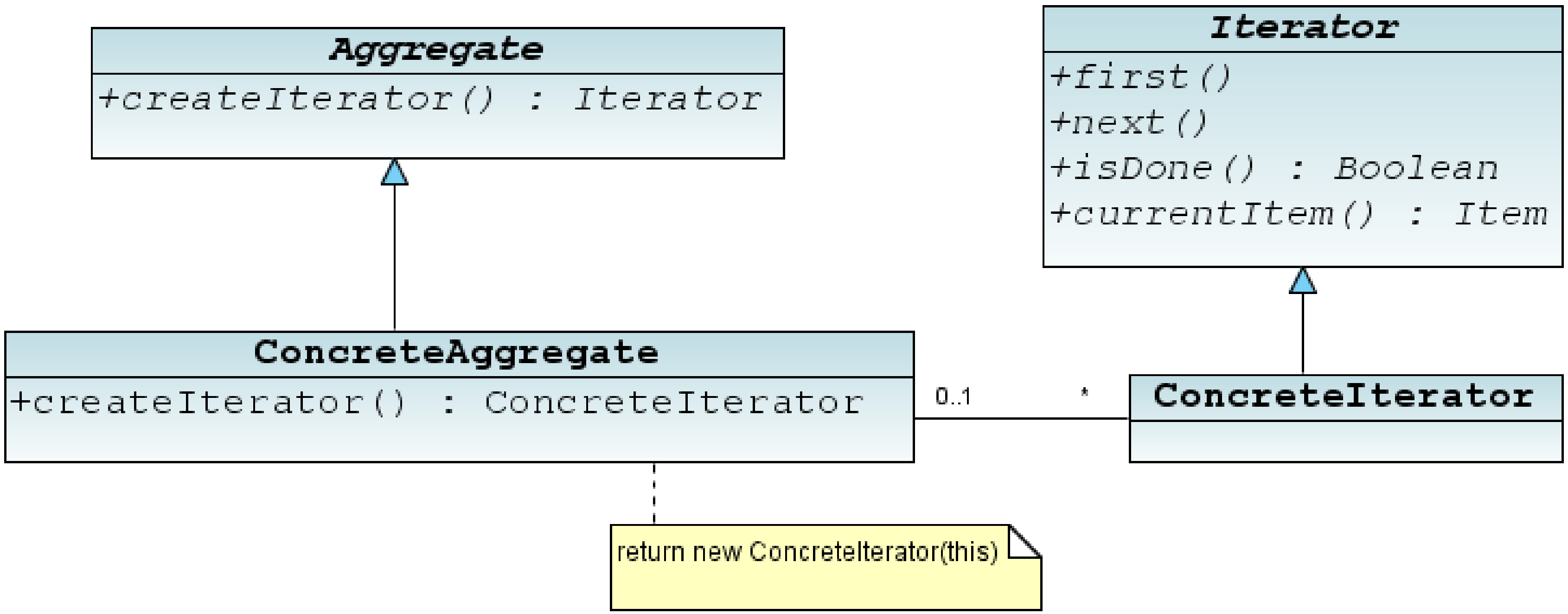


Example code

```
LinkedList list = new LinkedList();
CircularList circularList = new CircularList();

Iterator listIterator = list.createIterator();
Iterator circularIterator = circularList.createIterator();

handleList(listIterator);
handleList(circularIterator );

...
public void handleList(Iterator iterator) {
   iterator.First();
   while (!iterator.IsDone()) {
      Object item = iterator.CurrentItem();
      // Code here to process item.
      iterator.Next();
   }
}
```

Iterator – Structure

- A data structure (`Aggregate`) provides a **Factory Method** for iteration.
- `ConcretIterator` is **associated** with its data structure



Iterator – Example

```
List list = new List();
Iterator iter = list.createIterator();
iter.first();

while (!iter.isDone()) {
      Item item = iter.currentItem();
      // do something with item, z.B.
      if (item.getColor() == GREEN) {
            item.setColor(RED);
      }
      iter.next();
}

// typical C++ notation (acc. Gamma et al., 1995)
for(iter.first(); !iter.isDone(); iter.next()) { ... }
```

Iterator - Roles

- **Iterator**
  - Defines an interface for accessing and traversing elements
- **ConcreteIterator**
  - Implements the `Iterator` interface
  - Keeps track of the current position in the traversal
- **Aggregate**
  - Defines an interface for creating an `Iterator` object (a factory method!)
- **ConcreteAggregate**
  - Implements the `Aggregate` interface to return an instance of `ConcreteIterator`

Iterator – Remarks

- **Strongly typed languages have to consider the return type of `currentItem()`**
  - Can be solved with generic type parameters which initialize data structure and iterator
    - e.g. List<Order>, ListIterator<Order>
  - Can be solved alternatively with common superclasses of elements (at least Object)
- **Iterator may implement different iteration strategies on data structures**
  - e.g. inorder,
  - preorder,
  - postorder

Iterator – Example JDK

```
public interface Enumeration {
     boolean hasMoreElements();
     Object nextElement();
}
```

- **Data structures return an Enumeration object, e.g. Vector, Hashtable**

```
Vector vec;
Enumeration enum = vec.elements();
  while(enum.hasMoreElements()) {
    // Casting as nextElement returns Object
    Item element = (Item) nextElement();
    // ... do something with element
}
```

Iterator – Discussion

- **Advantages**
  - Separation of concerns between data structure and iterator
  - Several iterators on the same data structure
  - Polymorphous iteration abstracts of the concrete implementation of a data structure

- **Disadvantages**
  - Iterators access the internal state of the data structure → violation of encapsulation, high coupling
  - More objects and classes in the system

Exercise G

Design Patterns (acc. Gamma et al.)

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

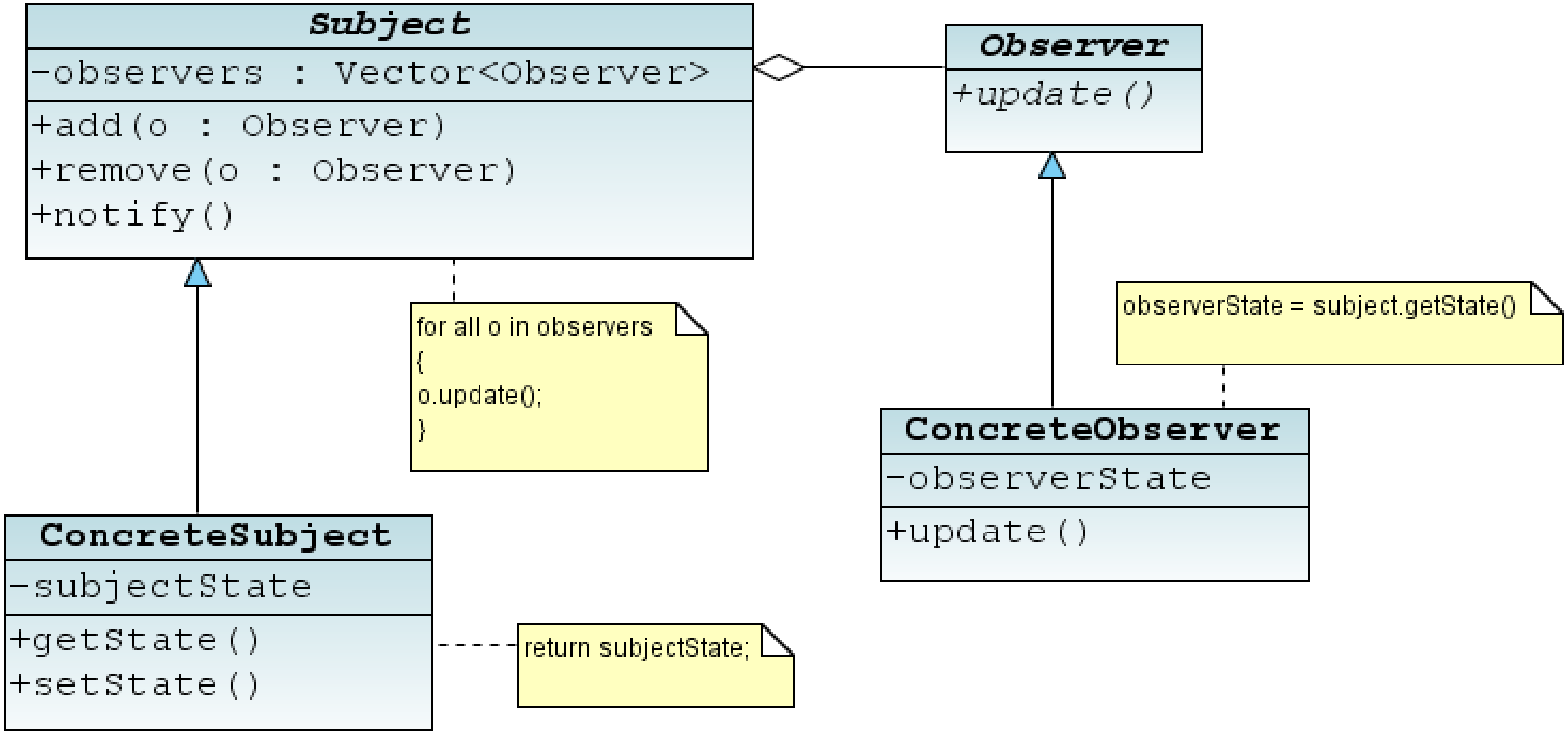
Pattern:
Observer

Observer – Problem / Solution

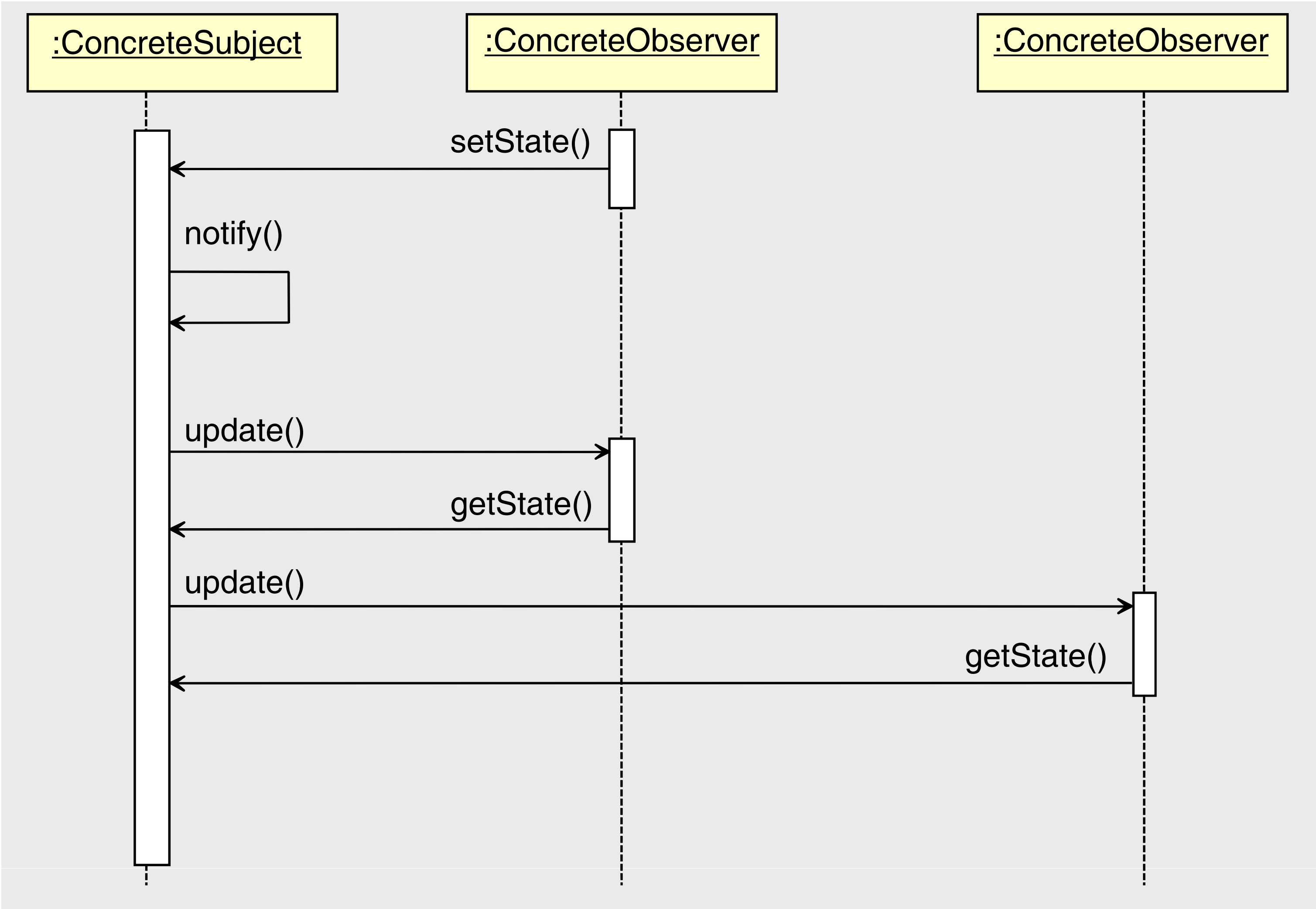
- **Definition of a 1:n-relationship between an object (`Subject`) and several depending (displaying) objects (`Observer`)**
- **If the `Subject` changes its state the depending objects shell be updated automatically**
  - The `Subject` may defer the update until a sequence of operations is done

→ Separation of representation and business logic

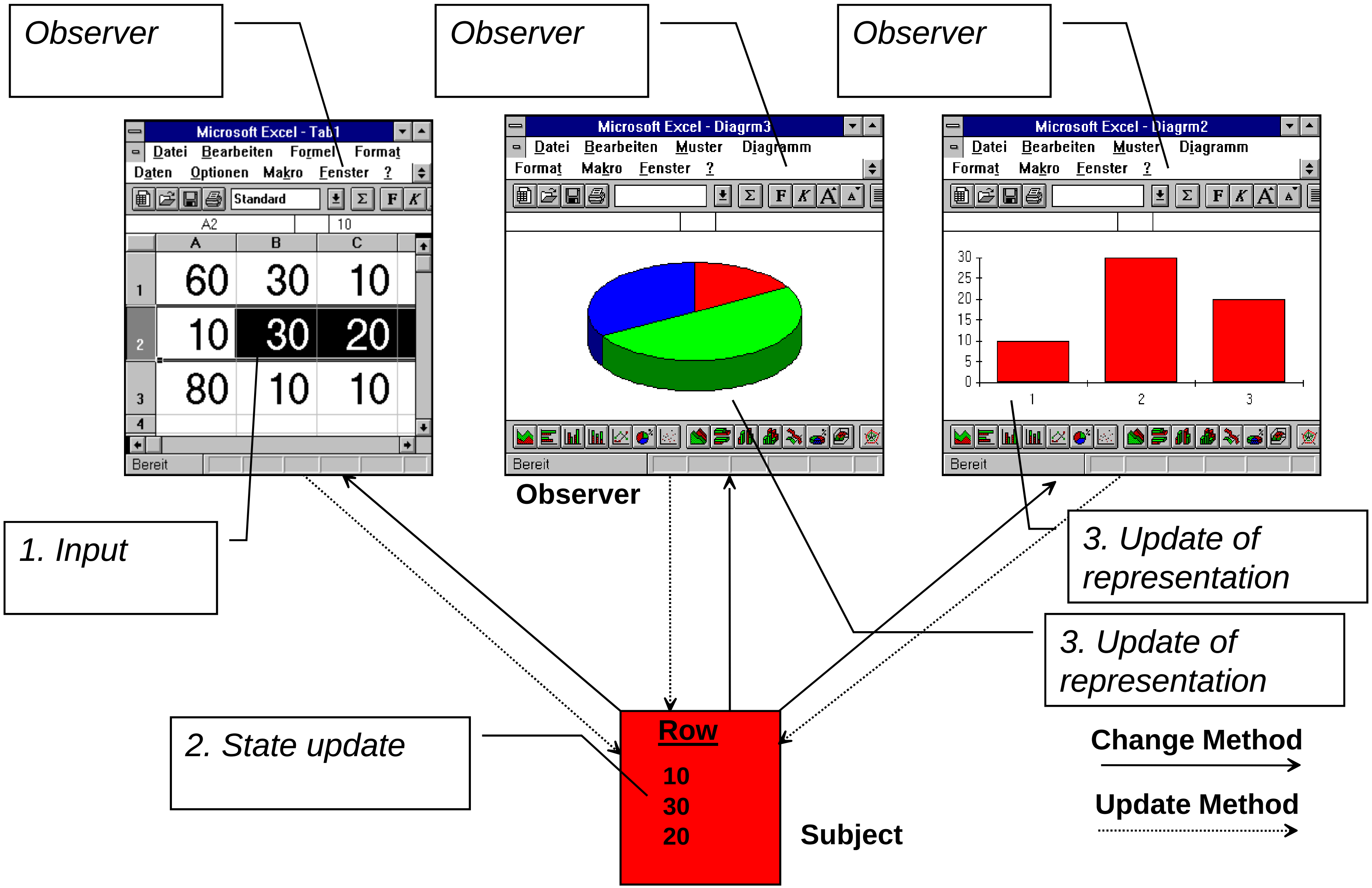
Observer – Structure



Observer – Sequence diagram



Observer – Example



Observer – Discussion

Advantages

- Minimal coupling between subject and observer
- New observers have to be registered without changing the subject

→ Implementation of „**Model-View-Controller**“

Disadvantages

- Bad design of update dependencies may lead to unnecessary / senseless updates

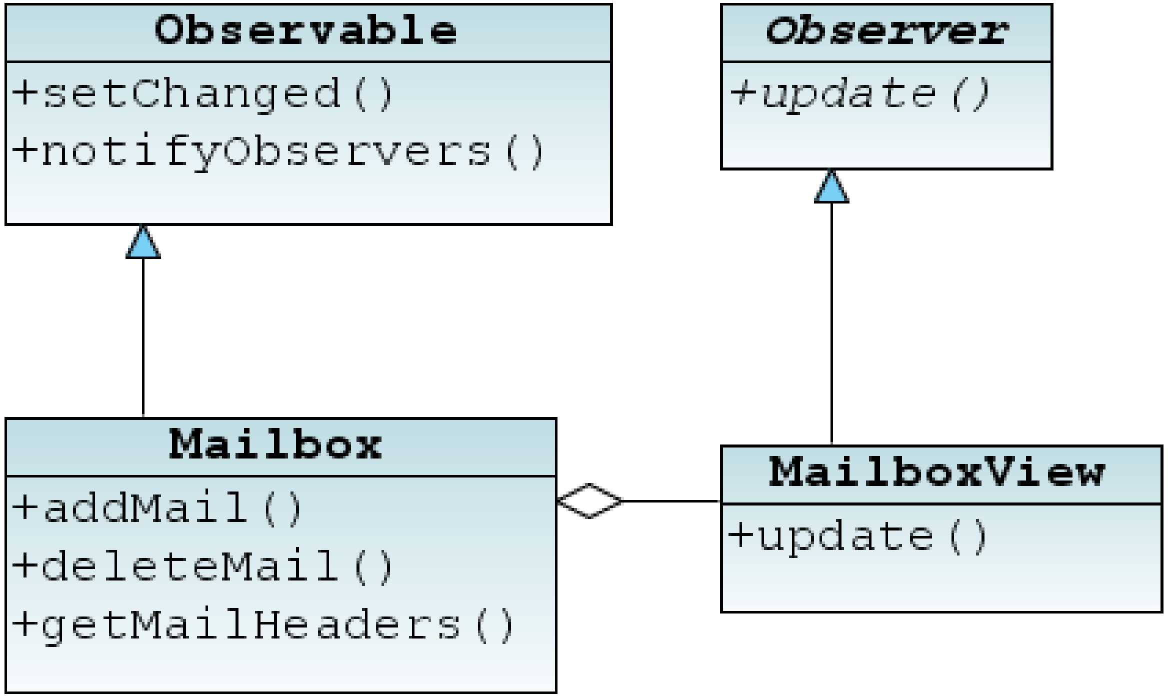
Observer – Class Observable

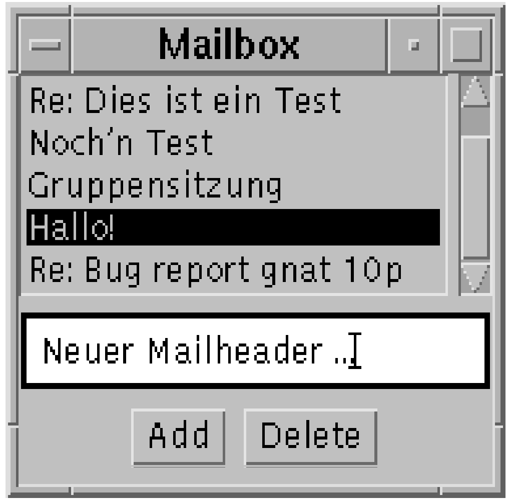
- **java.util provides a class `Observable` as superclass for a subject.**
- **Observer classes have to implement the interface `Observer` (update method).**

Method Summary	
`void`	**`addObserver`**`(Observer o)` Adds an observer to the set of observers for this object, provided that it is not the same as some observer already in the set.
`protected void`	**`clearChanged`**`()` Indicates that this object has no longer changed, or that it has already notified all of its observers of its most recent change, so that the `hasChanged` method will now return `false`.
`int`	**`countObservers`**`()` Returns the number of observers of this `Observable` object.
`void`	**`deleteObserver`**`(Observer o)` Deletes an observer from the set of observers of this object.
`void`	**`deleteObservers`**`()` Clears the observer list so that this object no longer has any observers.
`boolean`	**`hasChanged`**`()` Tests if this object has changed.
`void`	**`notifyObservers`**`()` If this object has changed, as indicated by the `hasChanged` method, then notify all of its observers and then call the `clearChanged` method to indicate that this object has no longer changed.
`void`	**`notifyObservers`**`(Object arg)` If this object has changed, as indicated by the `hasChanged` method, then notify all of its observers and then call the `clearChanged` method to indicate that this object has no longer changed.
`protected void`	**`setChanged`**`()` Marks this `Observable` object as having been changed; the `hasChanged` method will now return `true`.

Observer – Example 1

- **Simple Mailbox with GUI**





Observer – Example 2

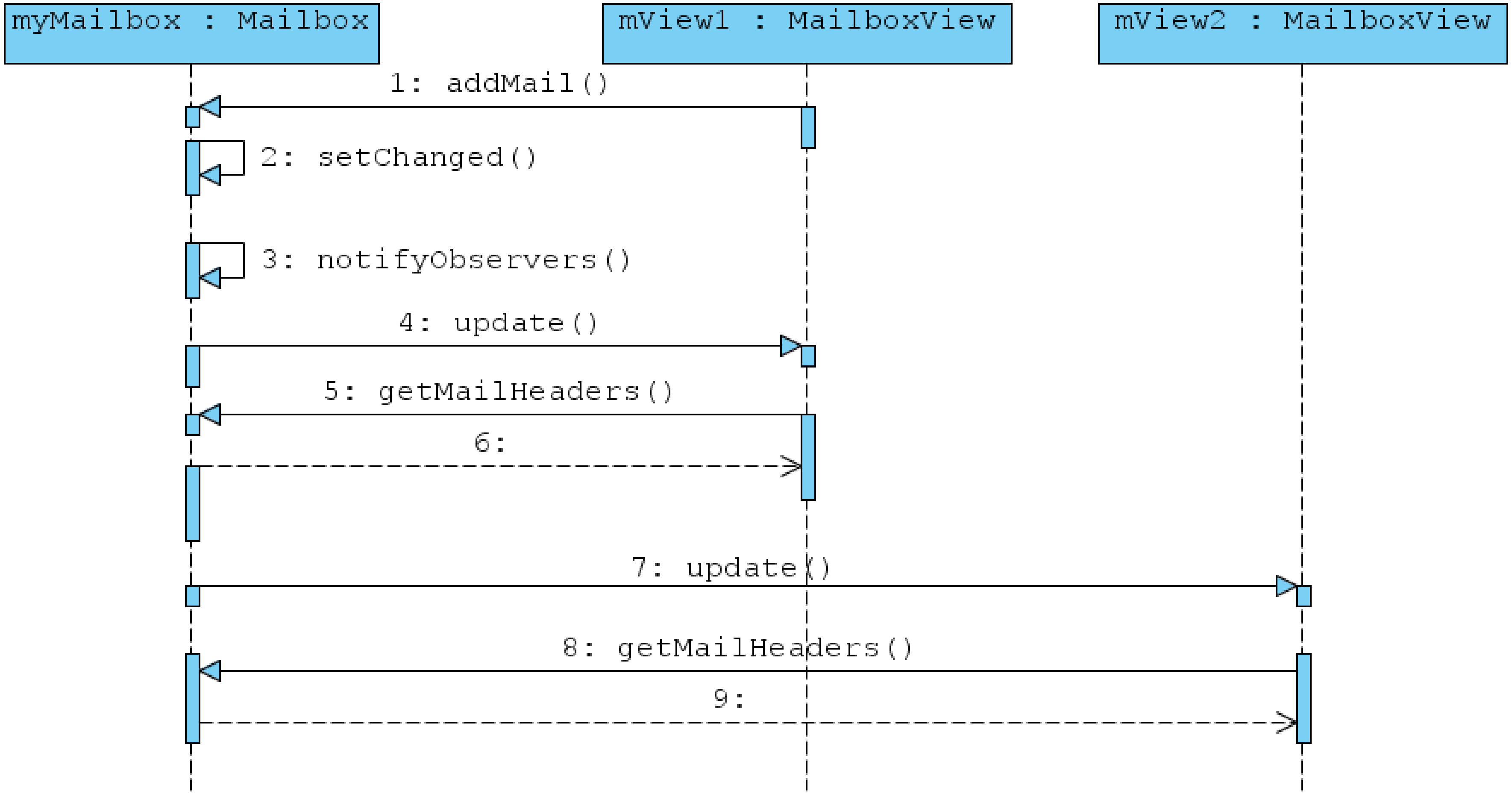
```
class Mailbox extends Observable {
private Vector mails;

public Mailbox() { ... }

public void addMail(String header) {
     if( (header != null ) && !mails.contains(header) )
{
          mails.addElement(header);
          this.setChanged(); // state changed
          this.notifyObservers(); // notification
     }
}

public void deleteMail(String header) { ... }

public Enumeration getMailHeaders() { ... }
}
```

Observer – Example 3

```
class MailboxView implements Observer {

private Mailbox mailbox;

public MailboxView(Mailbox box) {
     mailbox  = box;
     mailbox.addObserver(this); // enlist observer
     ...
}

public void update(Observable o, Object arg) {
     this.fillList(((Mailbox) o).getMailHeaders());
}

}
```

Observer – Example 4

Sequence diagram



Exercise H

Pattern: Decorator

Pattern: Decorator

Purpose

- Attach additional features to objects dynamically
- Remove those features if they are not needed anymore
- We want to freely combine features
  - E.g. an image can have a background, a border, a hyper-link and a note

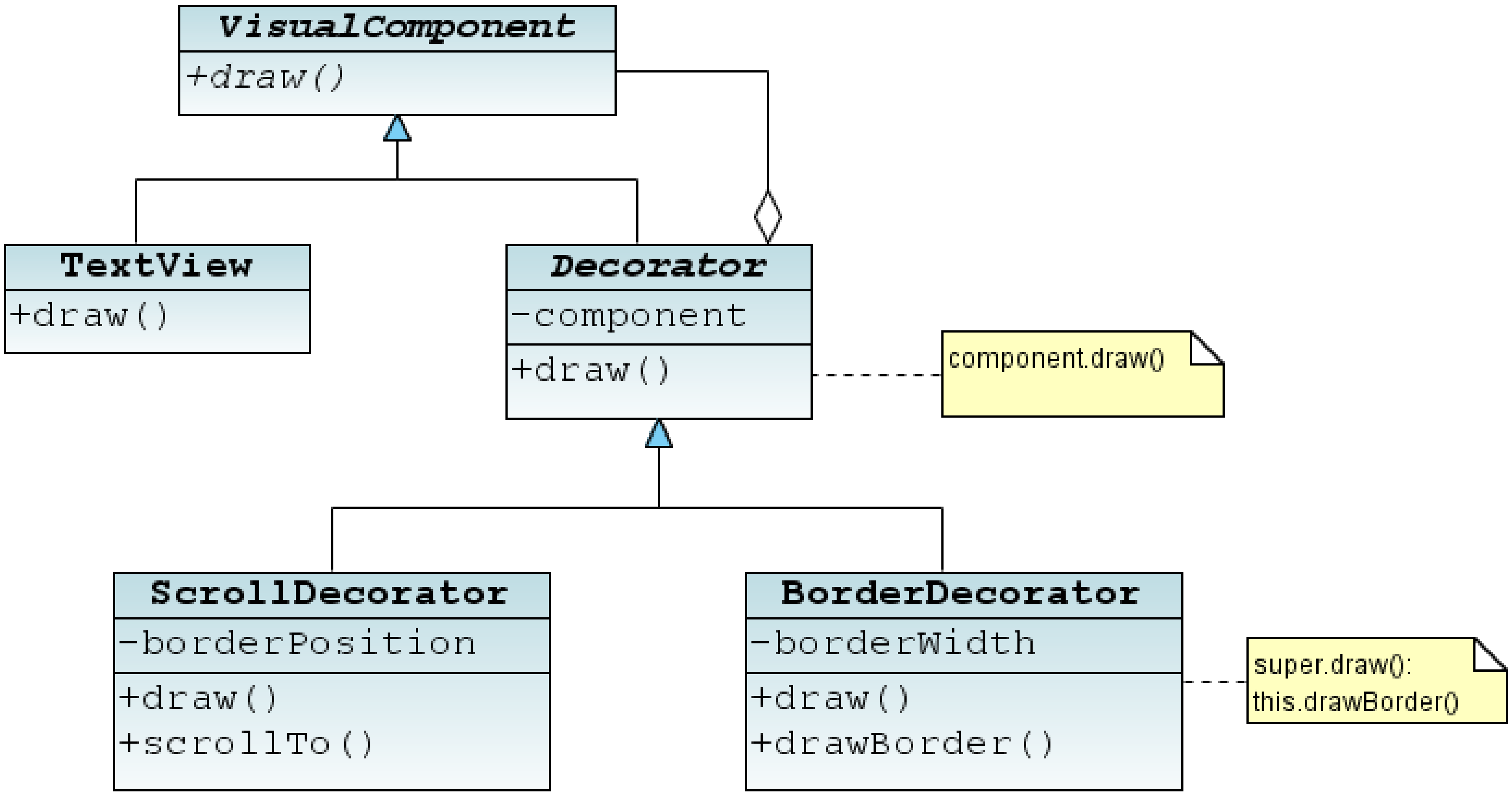
Application

- If inheritance is not practicable because of the large number of combinations of independent feature enhancements
- We cannot afford a class per combination

Solution

- Create a new class (Decorator) defining „intermediate objects"
- The intermediate object delegates operations to the "original" object, after having performed additional features

Decorator : Example



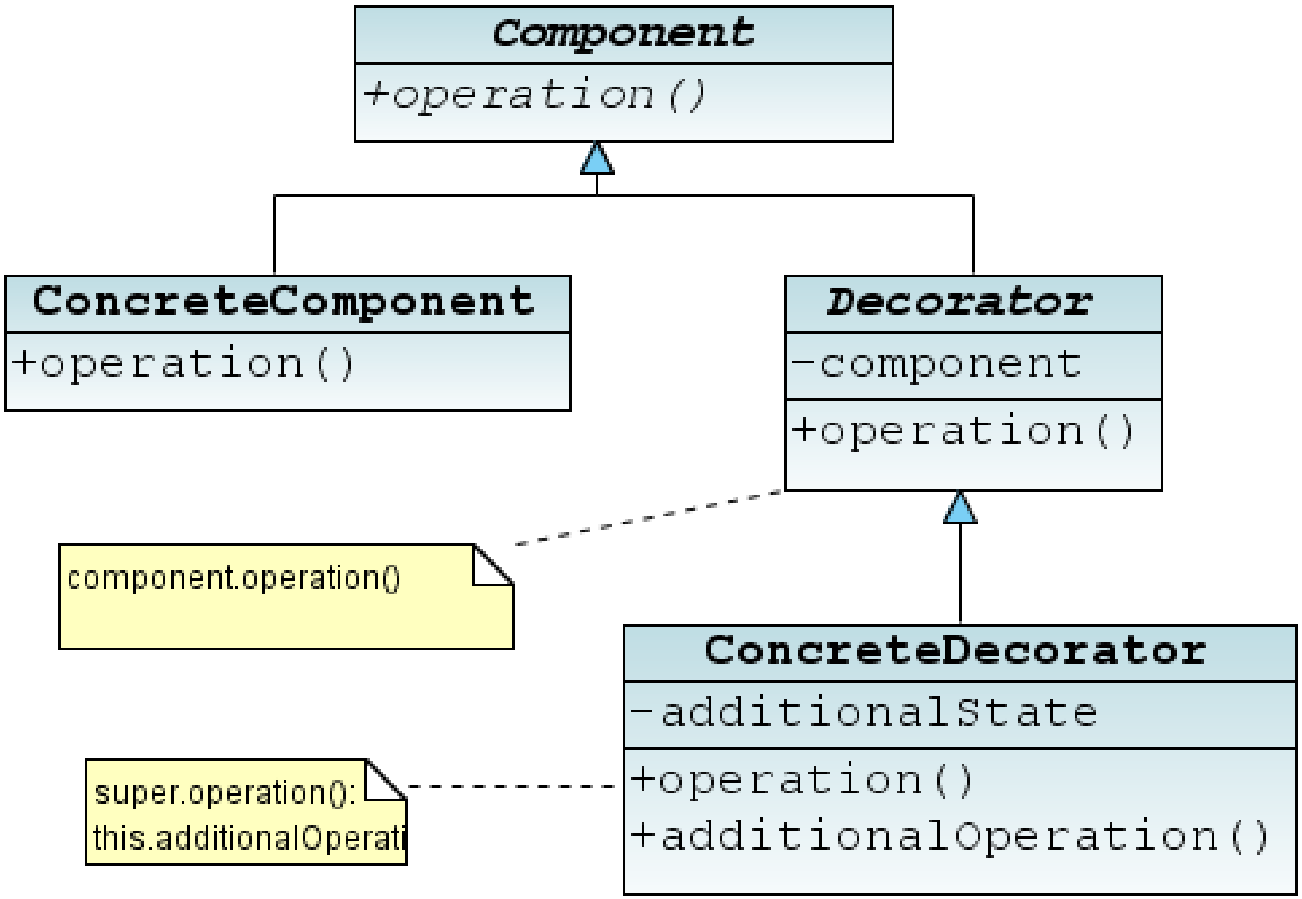
Decorator: Beispiel-Code

```
public abstract class VisualComponent {
        public abstract void draw();
```

```
public abstract class Decorator extends VisualComponent
{
  private VisualComponent comp ;
  public Decorator (VisualComponent comp) {
    this.comp = comp ;
  }
  public void draw() {
    comp.draw ();
  }
```

```
public class BorderDecorator extends Decorator {
  public BorderDecorator (VisualComponent comp) {
    super(comp);
  }
  public void draw() {
    super.draw();
    this.drawBorder();
  }
   private void drawBorder() {
   ...
   }
}
```

Decorator : Structure



Decorator : Discussion

- Increased flexibility compared to inheritance.
- Top-level classes of a hierarchy don't need to implement all features
- Decorators lead to an increased number of objects
- Decorator and component classes must have a common component superclass.

Design Patterns (acc. Gamma et al.)

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

Design Patterns – Characteristics

- A design pattern represents well-known and documented design experience which can be reused
- Design patterns introduce an abstraction on the level of micro-architecture (that's on top of singular classes)
- Design patterns may be combined
- Design patterns back up the development and documentation of complex and heterogeneous software designs
- Design patters facilitate changeability and reusability
- Design patterns form a vocabulary of design thus facilitating design
- Design patterns may be used for reengineering also

Summary

- Design patterns encapsulate design best practices

- The usage of design patterns leads to more flexible architecture

- Design patterns allow talking about design solutions

- Understanding design patterns is „easy“

- Applying design patterns needs design experience!