Software Architectures

2 SWS Lecture 1 SWS Lab Classes

Hans-Werner Sehring
Miguel Garcia

Arbeitsbereich Softwaresysteme (STS)
TU Hamburg-Harburg

HW.Sehring@tuhh.de
Miguel.Garcia@tuhh.de
http://www.sts.tu-harburg.de/teaching/ss-05/SWArch/entry.html

Summer term 2005

© Software Systems Department. All rights reserved.

3. Design Patterns

1. Motivation and Fundamental Concepts
2. Revisiting Object-Oriented Analysis, Design, and Implementation
3. Design Patterns
    3.1 Design Pattern Rationale
    3.2 Selected Design Patterns
4. Pipes & Filter Architectures
5. Event-based Architectures
6. Layered Architectures & Persistence Management
7. Framework Architectures
8. Component Architectures

© Hans-Werner Sehring, STS. Alle Rechte vorbehalten.

Learning Objectives of Chapter 3

Students should be able to …

- understand how design patterns describe problem solutions in an abstract way,
- recognize some of the most prominent design patterns,
- apply those patterns patterns, and to combine them to achieve dense designs.

Required Reading:

- [GoF] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
  The definite reference.

Additional Reading:

- John Vlissides: *Pattern Hatching: Design Patterns Applied*, Addison-Wesley, 1998.
- Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Somerlad, Michael Stal: *Pattern-Oriented Software Architecture*, John Wiley & Sons, 1996.
- W. Brown, R. C. Malveau, H. W. McCormick, T. J. Mowbray: *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*, Wiley & Sons, 1998.
  Quite funny for those with a little experience in larger software projects.

3.1 Design Pattern Rationale

Design patterns:

- Term originally coined by Christopher Alexander for the architecture of buildings and towns.
- First efforts of sharing design knowledge for software development:
  - Donald Knuth: *The Art of Computer Programming*, vol. 1-3, 1973
  - “architecture handbook” (OOPSLA ´91)
  - James Coplien: *Advanced C++ Programming Styles and Idioms*, 1992
  - … and similar “best practices” books
- Patterns taken to software engineering by the “Gang of Four” [GoF].
  Still the most influential work.

Christopher Alexander:
“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.” (quoted in [GoF])

© Hans-Werner Sehring, STS. Alle Rechte vorbehalten.

Elements of Design Patterns

Design patterns consist of four essential parts:

- **Name:**
  - a name for a pattern adds to the design vocabulary
  - it allows design at a higher-level of abstraction
- **Problem:**
  - description of a problem and its context
  - sometimes enumeration of typical design flaws
- **Solution:**
  - elements that make up the design and their relationships
  - responsibilities and collaborations
- **Consequences:**
  - time and space trade-offs
  - eventually language and implementation concerns

3.2 Selected Design Patterns

1. Motivation and Fundamental Concepts
2. Revisiting Object-Oriented Analysis, Design, and Implementation
3. **Design Patterns**
   - 3.1 Design Pattern Rationale
   - 3.2 Selected Design Patterns
4. Pipes & Filter Architectures
5. Event-based Architectures
6. Layered Architectures & Persistence Management
7. Framework Architectures
8. Component Architectures
9. Architecture-centric Software Construction

© Hans-Werner Sehring, STS. Alle Rechte vorbehalten.

Classification of Design Pattern

Following [GoF], we distinguish three categories of design patterns:

- creational patterns,
- structural patterns, and
- behavioral patterns.

[GoF, p. 10]

handled in this chapter
handled in other chapters

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

Singleton Pattern (1)

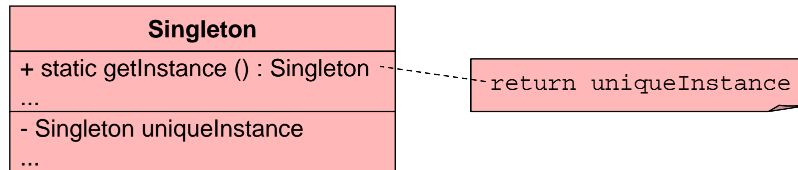
Category: creational.

Useful when

- There must be only one instance of a class.
  Examples: resource manager objects, locks.
  `java.awt.Toolkit.getDefaultToolkit ()`
- It has to be accessible from a well-known access point.
  Achieves less polluted namespaces, removes the need for a global variable.
  Examples: input and output streams, log manager.
  `System.out` (not exactly a Singleton)
- Client code should remain stable even when the instance's class is refined.
  Example: localized versions of numbers, dates, …
  `java.util.Calendar.getInstance ()`
  (gets a calendar using the default time zone and locale)

Singleton Pattern (2)

Structure:



Singleton classes …

- define a method that delivers the one instance, and
- are themselves responsible for creating that instance.

Singleton Pattern (3)

In Java (typically):

```
public class Singleton {

    static private Singleton uniqueInstance ;

    static public Singleton getInstance () {
        if (uniqueInstance == null)
            uniqueInstance = new Singleton (…) ;
        return uniqeInstance ;
    }// getInstance

    private Singleton (…) {
        …
    }// constructor

    …

}// class Singleton
```

not callable from outside

 © Hans-Werner Sehring, STS. Alle Rechte vorbehalten.

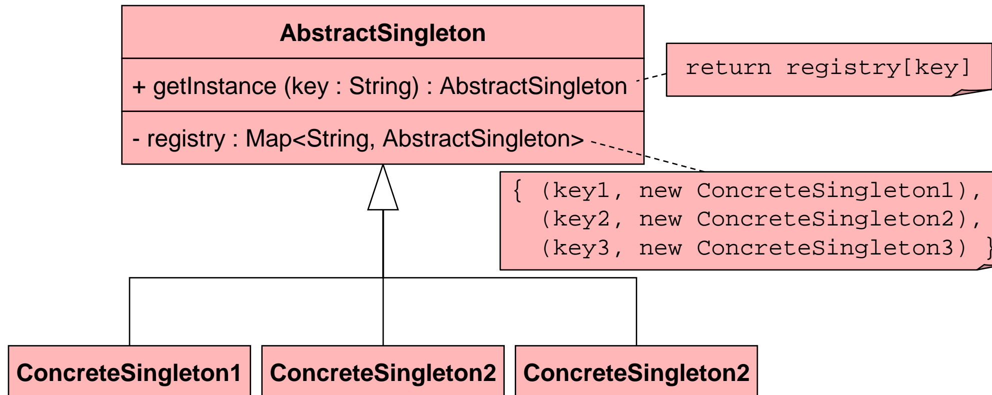
Singleton Pattern (4)

Sometimes there is a class managing singletons: registry of singletons.

Singletons are registered under keys (strings, numbers) and can be retrieved later.

Example: `java.awt.color.ColorSpace.getInstance (int colorspace)`

Especially in conjunction with the ability to deliver an instance of a subclass.



Singleton Pattern (5)

In Java:

```
public class Singleton {
    static private Singleton uniqueInstance ;
    static private Hashtable instances ;
    static {
        instances = new Hashtable (3) ;
        instances.put ("key1", new Singleton (x)) ;
        instances.put ("key2", new Singleton (y)) ;
        instances.put ("key3", new SpecialSingleton ()) ;
    }// static constructor
    static private Singleton getInstance () {
        if (uniqueInstance == null) {
            String key = System.getProperty ("SOME_SINGLETON") ;
            uniqueInstance = (Singleton)instances.get (key) ;
        }
        return uniqueInstance ;
    }// getInstance

    private Singleton (…) {
        …
    }// constructor
}// class Singleton
```

initialization

variable set in environment to either *key1*, *key2*, or *key3*

© Hans-Werner Sehring, STS. Alle Rechte vorbehalten.

Abstract Factory Pattern (1)

Category: creational.

To maximize code reuse it is desirable to abstract from concrete classes and instead only use the names of interfaces or abstract classes.

One point where concrete classes are needed is for object construction.

Example:

- Create elements of a graphical user interface (GUI).
- There are two variants of the GUI.
  - One is programmed in the Java AWT:
    ```
    Window    win = new java.awt.Frame () ;
    Component txt = new java.awt.TextField () ;
    …
    ```
  - A variant of the GUI is programmed using Java Swing:
    ```
    Window    win = new javax.swing.JFrame () ;
    Component txt = new javax.swing.JTextField () ;
    …
    ```


Abstract Factory Pattern (2)

To abstract from concrete classes at the point of instantiation, factories are used:

```
UIFactory uiFactory = new AWTUIFactory () ;
                      or = new SwingUIFactory ()
Window    win = uiFactory.createWindow () ;
Component txt = uiFactory.createTextField () ;
```

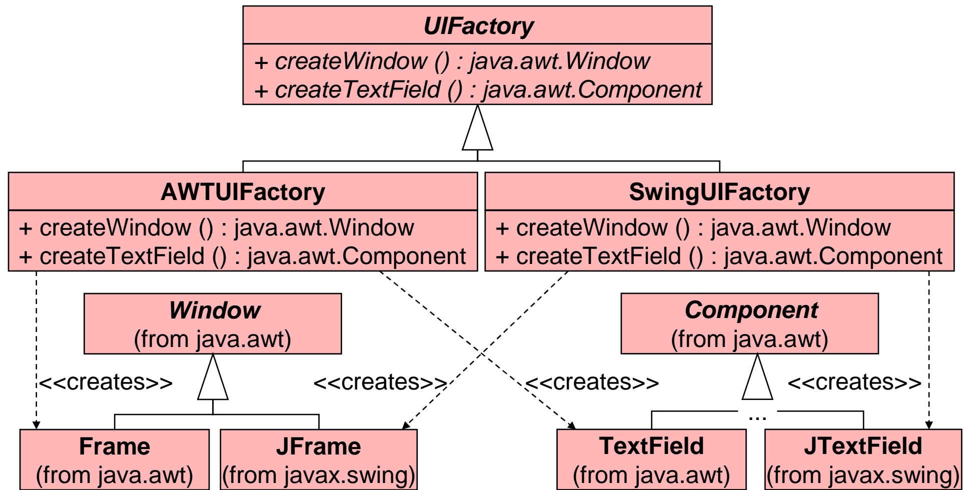
Advantages:

- An application can dynamically choose one implementation out of a family of possible implementations.
- Client code does not directly use implementation classes. Thus, these can be changed.
- Concrete classes can be added, e.g., new UI component types.

© Hans-Werner Sehring, STS. Alle Rechte vorbehalten.

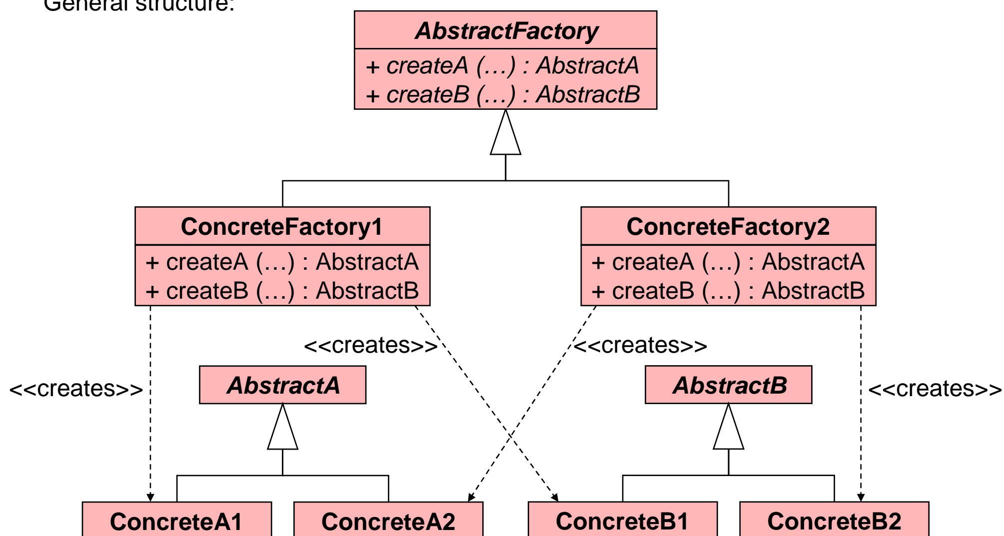
Abstract Factory Pattern (3)

To choose an implementation, a hierarchy of factory classes is established in correspondence to the application class hierarchy.



Abstract Factory Pattern (4)

General structure:



© Hans-Werner Sehring, STS. Alle Rechte vorbehalten.

Factory Method Pattern (1)

Category: creational.

Often creator objects are used to instantiate classes. One example just seen: factories.

If a creator's class does not know the classes to instantiate, the creation can be deferred to subclasses.

Example: (graphical) UML editor.

- The UML editor has a polymorphic “new node” menu to add elements to a diagram.
- The type of node added depends on the currently selected diagram.

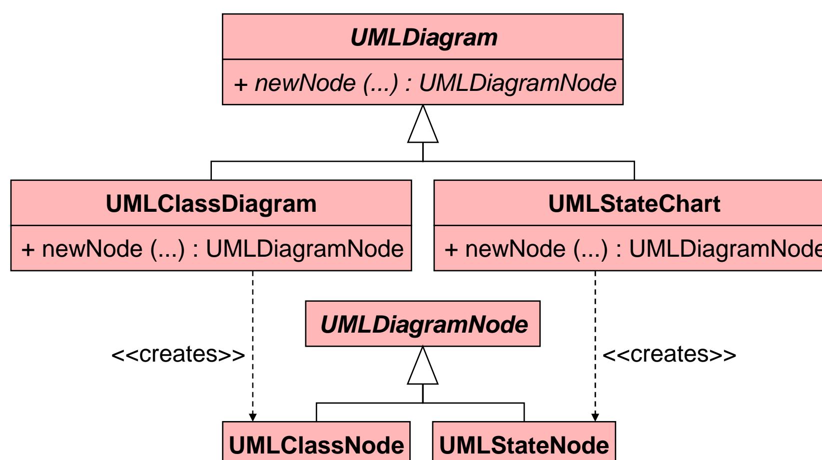
Naïve way:

```
UMLDiagramNode newNode ;
if (currentDiagram instanceof UMLClassDiagramm)
    newNode = new UMLClassNode (…) ;
else if (currentDiagram instanceof UMLStateChart)
    newNode = new UMLStateNode (…) ;
else
    …
```


Factory Method Pattern (2)

Better solution: diagrams are creators of nodes:

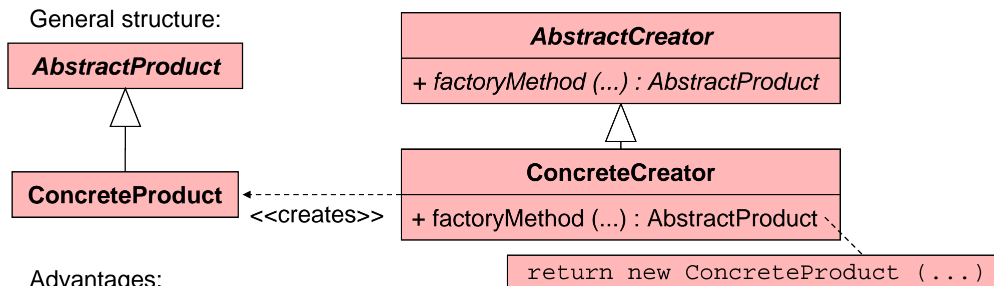
```
UMLDiagramNode newNode = currentDiagram.newNode (…) ;
```



 © Hans-Werner Sehring, STS. Alle Rechte vorbehalten.

Factory Method Pattern (3)

General structure:



Advantages:

- a class doesn't have to know the concrete classes for which to create objects (in the example: `UMLDiagram`).
- Because of this fact concrete classes can be added without changing the creator class (*`AbstractCreator`*).
- Knowledge on object construction is localized in places where it belongs (in the example: class diagrams know about classes, state charts about states, …).

Prototype Pattern (1)

Category: creational.

Sometimes objects have to be created and initialized without their class being known.

Example:

- A graphical editor framework can is able to handle shapes of different kinds – lines, rectangles, ellipses, … (Frameworks are discussed later in this lecture.)
- The set of kinds of shapes is *dynamically* extendable by client code.
- Therefore, the graphical editor framework cannot contain code to instantiate shapes itself.

Solution:

- Client code creates a *prototype* of a shape it wants the framework to use.
- The framework can create new shapes by *cloning* the prototype.
- The clone is initialized with the same state as the original.

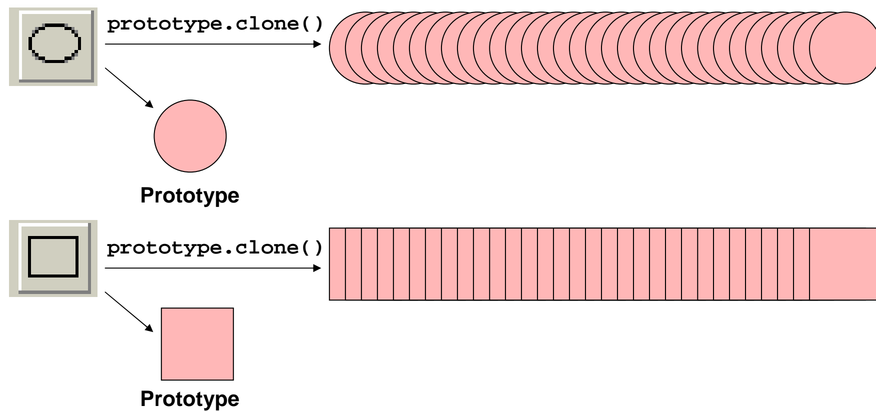
Advantages:

- Instances can be created without knowing their class.
- New instances are initialized without knowing their interface.
- This way, object creation is decoupled from the use of concrete classes.

© Hans-Werner Sehring, STS. Alle Rechte vorbehalten.

Prototype Pattern (2)

Each tool is initialized with an instance (a prototype) of the shape it is meant to create.
When creating a new shape, the tool clones the prototype.



Prototype Pattern (3)

Example:

- Framework:

```
class EditorFramework {
  private Hashtable shapesFromClient ;
  public EditorFramework () {
    shapesFromClient = new Hashtable () ;
  }// constructor
  public void register (String shapeName, Shape s) {
    shapesFromClient.put (shapeName, s) ;
  }// register
  public void addShape (String shapeName) {
    Object prototype = shapesFromClient.get (shapeName) ;
    Shape newShape = (Shape)prototype.clone () ;
    … (add newShape to drawing area etc.)
  }// addShape
}// class EditorFramework
```

- Client code:

```
…
EditorFramework editorFramework = new EditorFramework () ;
…
editorFramework.register ("my shape",
                          new SpecificShape (1, "xyz", 5.7));
…
editorFramework.addShape ("my shape") ;
```

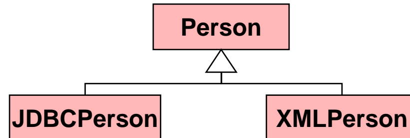
© Hans-Werner Sehring, STS. Alle Rechte vorbehalten.

Bridge Pattern (1)

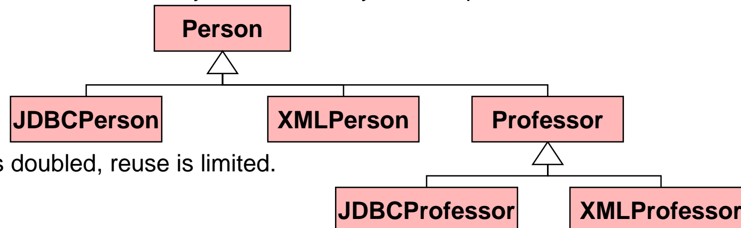
Category: structural.

When there are variants of an implementation, usually subclassing is used.
This makes the addition of further classes difficult, esp. with single inheritance.

Example: persistent person objects (`Person`) are either stored in a relational database (`JDBCPerson`) or in an XML file (`XMLPerson`).



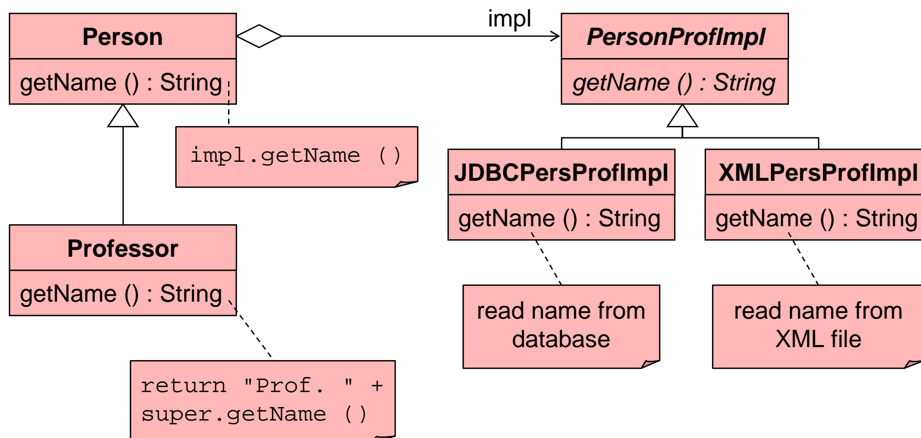
If you now want to refine `Person` to, say, `Professor`, you end up with:



Thus, the hierarchy is doubled, reuse is limited.

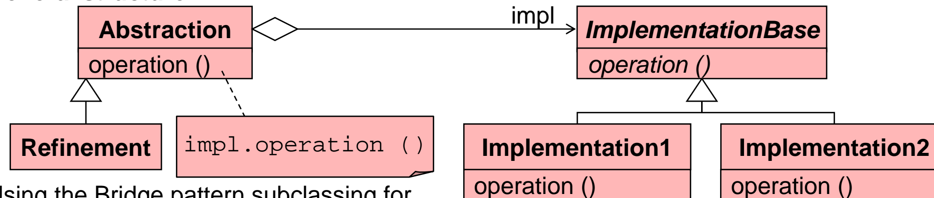
Bridge Pattern (2)

A better solution is to separate the conceptual and the implementation hierarchy, and to connect them by a bridge.



Bridge Pattern (3)

General structure:



Using the Bridge pattern subclassing for conceptual and technical reasons can be separated.
Especially, if subclasses solely rely on the interface of the superclass.

Advantages:

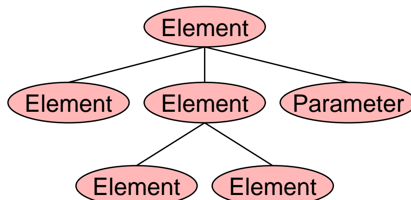
- Implementation can be chosen dynamically.
- Both conceptual abstractions and implementations can be extended by subclasses.
- Changes to an implementation have no impact on client code.
- The implementation classes are completely hidden from client code.
- Implementation objects can be shared.

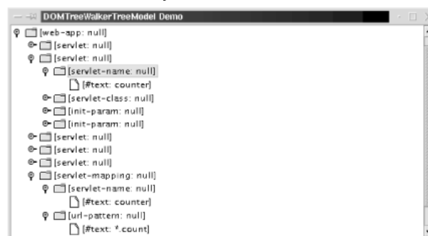
Adapter Pattern (1)

Category: structural.

Example: Visualization of XML documents.

- The Document Object Model (DOM) of an XML document is a tree.
- For trees exists a view in the standard Java class library: `javax.swing.JTree`.





But:

- The model of a JTree is a TreeModel (`javax.swing.tree.TreeModel`).
- How to use a DOM (`org.w3c.dom.Document`)?

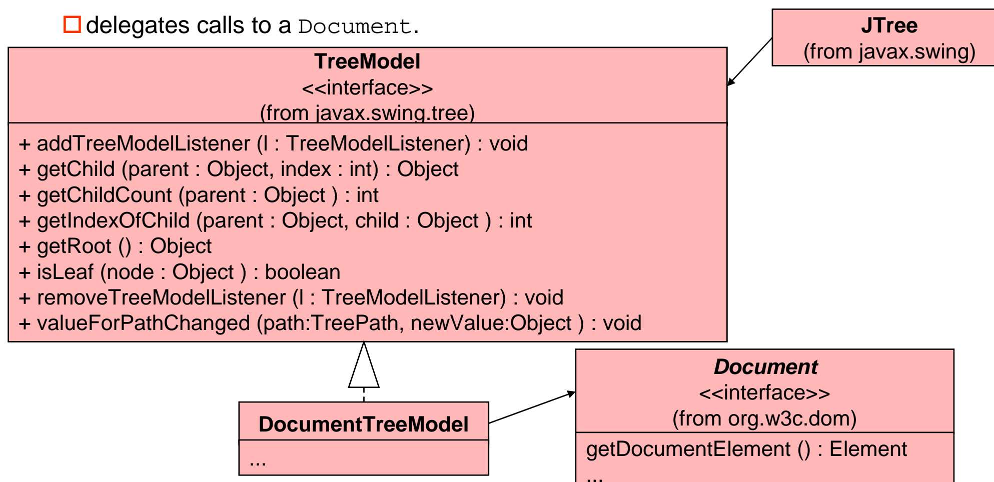
Solution: make a DOM look like a TreeModel.

 © Hans-Werner Sehring, STS. Alle Rechte vorbehalten.

Adapter Pattern (2)

Create a class `DocumentTreeModel` which

- implements `TreeModel` and
- delegates calls to a `Document`.



Adapter Pattern (3)

In Java:

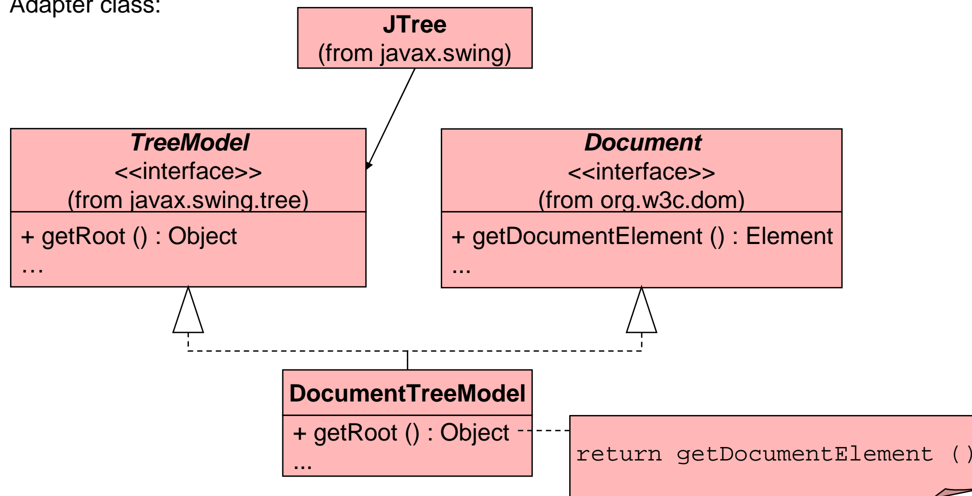
```
public class DocumentTreeModel implements TreeModel {

    private Document doc ; // the DOM to use

    public DocumentTreeModel (Document document) {
        this.doc = document ;
    }// constructor

    public Object getRoot () {
        return doc.getDocumentElement () ;
    }// getRoot

    public boolean isLeaf (Object node) {
        if (node instanceof Node)
            return ((Node)node).getChildNodes().getLength() == 0;
        else
            throw new RuntimeException ("Something is wrong?") ;
    }// isLeaf

    ...

}// class DocumentTreeModel
```

 © Hans-Werner Sehring, STS. Alle Rechte vorbehalten.

Adapter Pattern (4)

As in many cases, one can choose between delegation and subclassing.

Adapter class:



Adapter Pattern (5)

Applications of the Adapter pattern:

- Reuse a class with an interface that is not as needed
  (e.g., DOM in a JTree).
- Create a class that interoperates with different other classes that do not share a common interface
  (e.g., tree model for DOM, file system, …).
- Adapt multiple classes at once by adapting their parent interface, e.g., if number of subclasses would prohibitively high
  (e.g., adapt class Node from DOM instead of all Node subclasses separately).

 © Hans-Werner Sehring, STS. Alle Rechte vorbehalten.

Adapter Pattern (6)

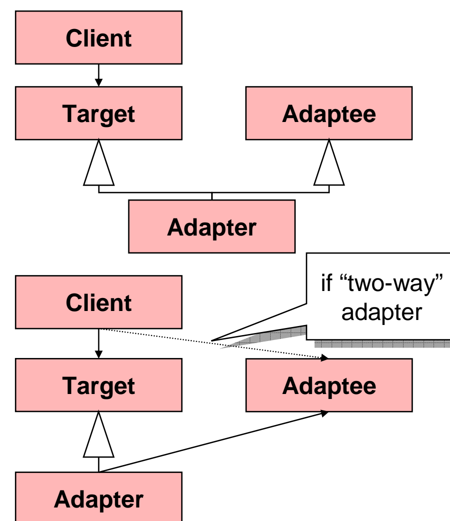
The general structure of Adapters is:

Class adapter:

- Can adapt only one class.
- Can override some of adaptee's behavior (since it is a subclass).
- Consists of only one object.

Object adapter:

- Can adapt more than one adaptee.
- Adaptee can change dynamically.
- Harder to override adaptee's behavior.



Decorator Pattern (1)

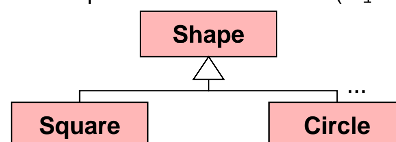
Category: structural.

Decorators dynamically attach new responsibilities to an object.

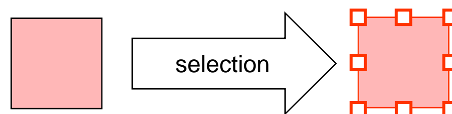
For extending functionality, decoration is a flexible alternative to subclassing.

Example: a graphics program with a drawing pane holding shape objects.

- Various kinds of shapes can be added to the drawing pane.
- For each kind of shape there one is a class (`Square`, `Circle`, …).



- A shape object is drawn with handles when selected.

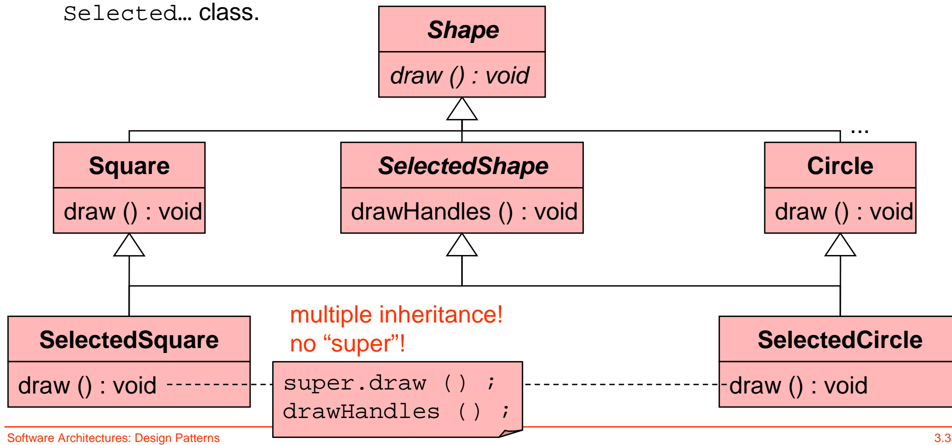


 © Hans-Werner Sehring, STS. Alle Rechte vorbehalten.

Decorator Pattern (2)

Naïve solution:

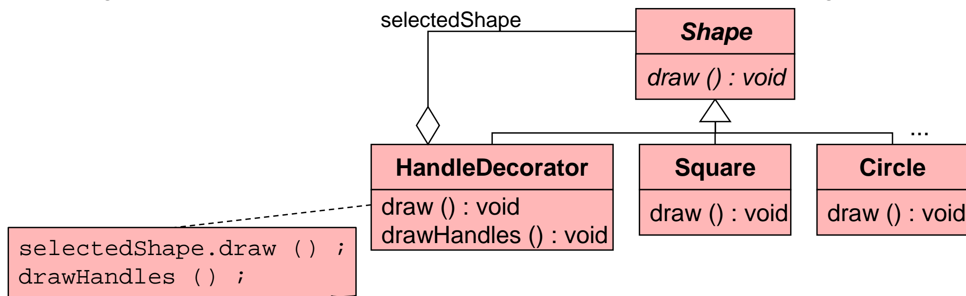
- Create subclasses `SelectedSquare`, `SelectedCircle`, … for each class `Square`, `Circle`, …
- When selected, exchange a shape instance with an instance of the corresponding `Selected`… class.



Decorator Pattern (3)

Instead of having a subclass for each class there is only one `HandleDecorator` for all kinds of shapes.

Making the `HandleDecorator` itself a Shape, it can be put on the drawing pane.

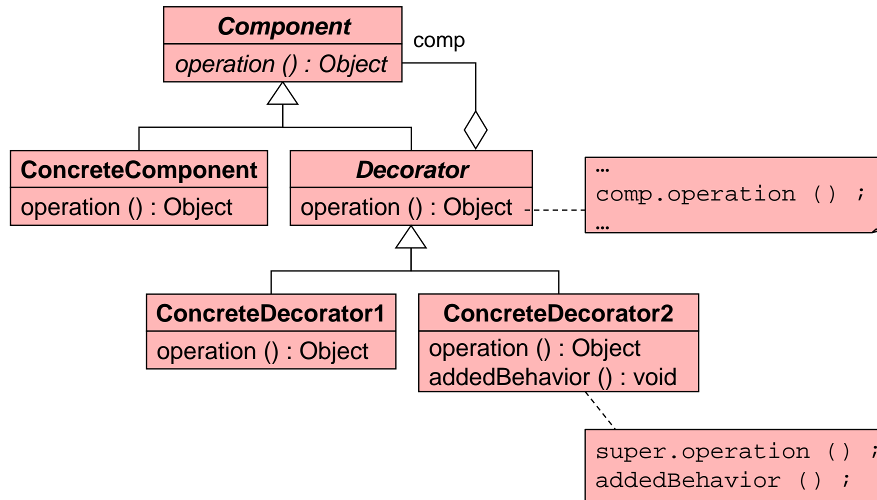


The `HandleDecorator`, upon redraw, lets the selected shape draw itself, and then adds the handles.
(Note: This way handles can decorate handles. To avoid this one more class is needed.)

 © Hans-Werner Sehring, STS. Alle Rechte vorbehalten.

Decorator Pattern (4)

General structure:



Decorator Pattern (5)

Advantages:

- Add behavior to an object dynamically, without affecting other objects.
- Change responsibilities.
- Offers an alternative to subclassing when the latter is not practical because of:
  - explosion of the number of subclasses,
  - multiple inheritance with languages which do not have this feature,
  - changing behavior, or because
  - class definitions are hidden or otherwise not available for subclassing (final, …).

 © Hans-Werner Sehring, STS. Alle Rechte vorbehalten.

Façade Pattern (1)

Category: structural.

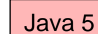
Often several objects are used in conjunction, e.g., if

- functionality is spread over several objects of a subsystem, or
- one operation needs to be carried out by several objects concurrently.

Example: While processing, an application GUI marks some menu items inactive.

- *All* menu items deactivated manually:

```
void deactivate (MenuElement m) {
  m.setActive (false) ;
  for (MenuElement me : m.getSubElements ())
    deactivate (me) ;
}// deactivate
```



- *Some* menu items deactivated manually:

```
menuItem1.setActive (false) ;
…
menuItemn.setActive (false) ;
```


Façade Pattern (2)

With naïve solution:

- Behavior is spread over application.
- Client code depends on implementation (here: Swing classes).

With a Façade (containing the code from the previous slide):

```
public class MenuItemActivationFacade {
  private JMenuItem menuItem1, …, menuItemn ;
  public MenuItemActivationFacade (MenuElement m1, …, MenuElement mn) {
    this.menuItem1 = m1 ;
    …
    this.menuItemn = mn ;
  }// constructor
  public void setActive (boolean active) {
    menuItem1.setActive (active) ;
    …
    menuItemn.setActive (active) ;
  }// setActive
}// class MenuItemActivationFacade
```

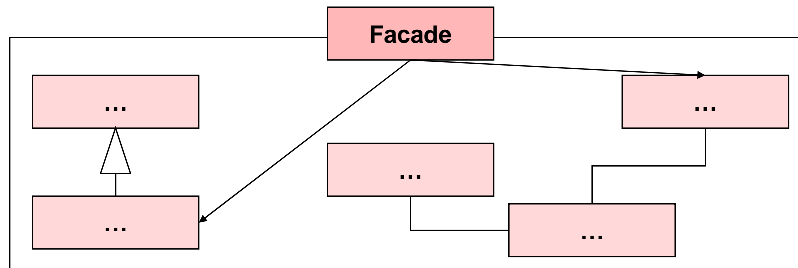
Client code then:

```
MenuItemActivationFacade miaf = new MenuItemActivationFacade (m1,…,mn) ;
miaf.setActive (false) ;
```

© Hans-Werner Sehring, STS. Alle Rechte vorbehalten.

Façade Pattern (3)

General structure:



Advantages:

- Provides a simple interface to a complex subsystem.
- Decouples clients from implementation classes.
- Defines a layering of subsystems with Façades as entry points.

Observer Pattern (1)

Category: behavioral.

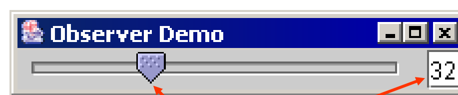
Good design breaks down a system into a set of cooperating classes.

For cooperation, objects have to know each other.

For reusability you want to decouple classes as much as possible.

Example: user interfaces.

- There are distinct visualizations of the same set of data (compare MVC).
- The views on the data have to be consistent.
- UI components should be reusable separately of each other.

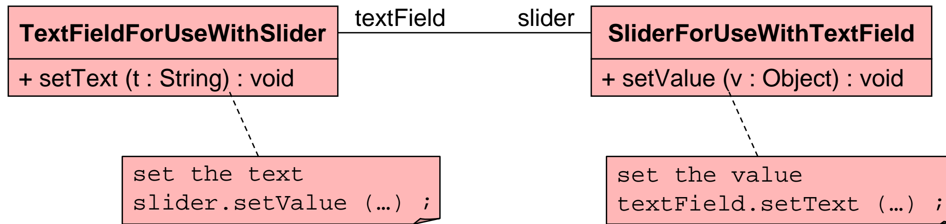


keep values consistent

© Hans-Werner Sehring, STS. Alle Rechte vorbehalten.

Observer Pattern (2)

Naïve object-oriented solution:

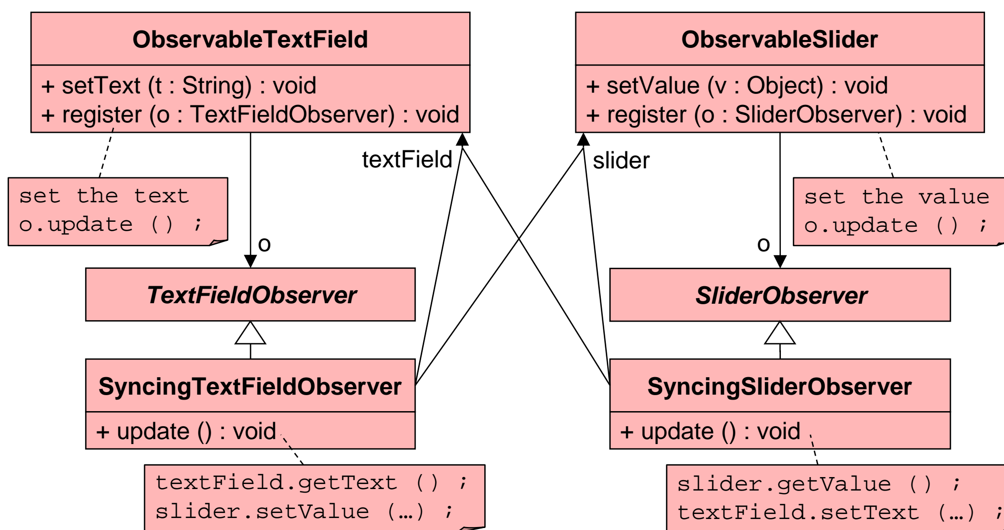


This way, the classes for textfields and sliders depend on each other.

For other UI setups, different classes would be needed.

Observer Pattern (3)

With Observer:



© Hans-Werner Sehring, STS. Alle Rechte vorbehalten.

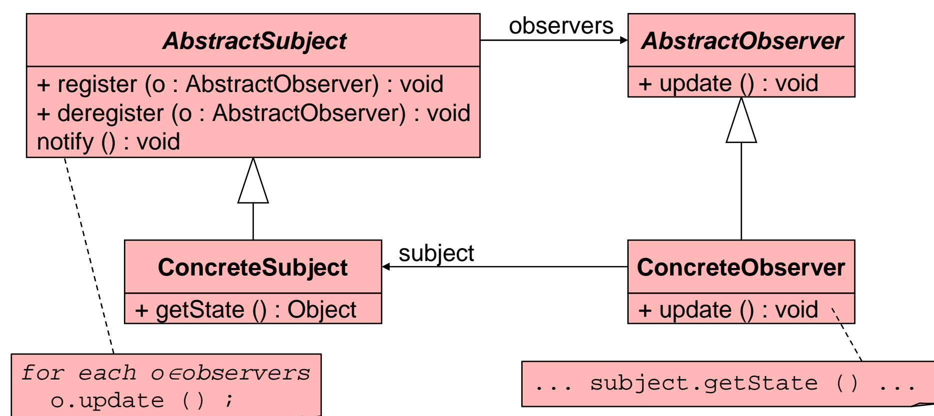
Observer Pattern (4)

In Java (both Observer classes implemented in one):

```
class SliderTextFieldListener
  implements ActionListener, ChangeListener
{
  private JSlider g ;
  private JTextField t ;
  public SliderTextFieldListener (JSlider g, JTextField t) {
    this.g = g ;
    this.t = t ;
    g.addChangeListener (this) ;
    t.addActionListener (this) ;
  }// constructor
  public void stateChanged (ChangeEvent e) {
    t.setText (Integer.toString (g.getValue ())) ;
  }// stateChanged
  public void actionPerformed (ActionEvent event) {
    int val ;
    try { val = Integer.parseInt (t.getText ()) ; }
    catch (NumberFormatException exc) { val = 0 ; }
    g.setValue (val) ;
  }// actionPerformed
}// class SliderTextFieldListener
```


Observer Pattern (5)

General structure:

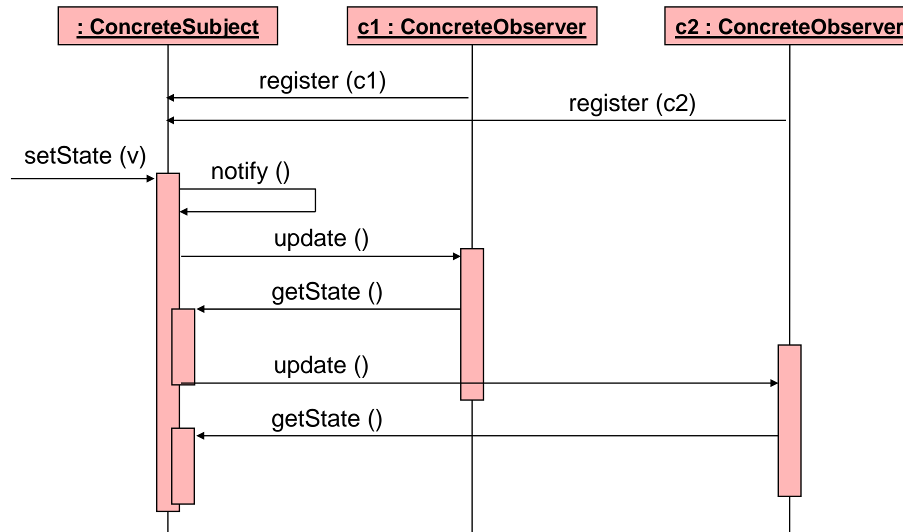


For further decoupling, e.g., event objects in Java.

 © Hans-Werner Sehring, STS. Alle Rechte vorbehalten.

Observer Pattern (6)

Interactions:



Observer Pattern (7)

Advantages:

- Encapsulates behavioral aspects.
- Changes of behavior can be done regardless of the subjects.
- Establishes lose coupling of objects (no direct method calls).

Disadvantage:

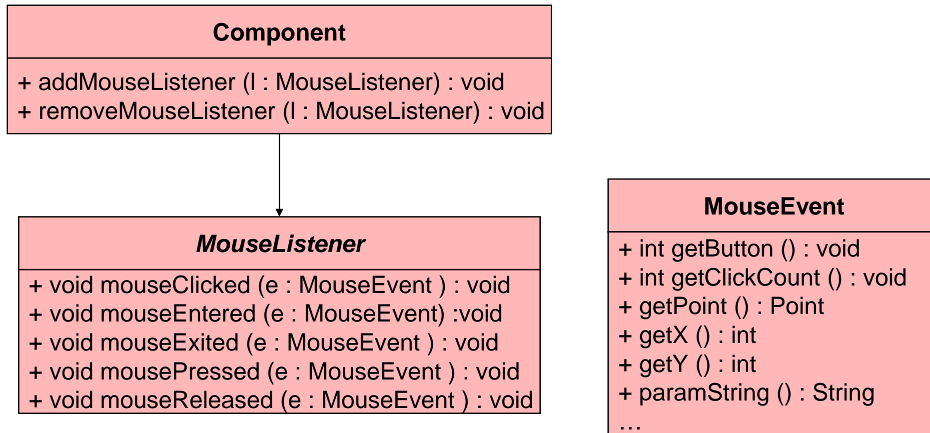
- Subjects have to expose inner state to observer (ex.: `getState()` is public).
- State of the subject while notifying observer may differ from the state when observer analyzes it.
- Better with event objects: pass event objects which *encapsulate* the state needed for usual applications *at the time of notification*.

 © Hans-Werner Sehring, STS. Alle Rechte vorbehalten.

Observer Pattern (8)

In Java: Event-Listener-Model (Java 2).

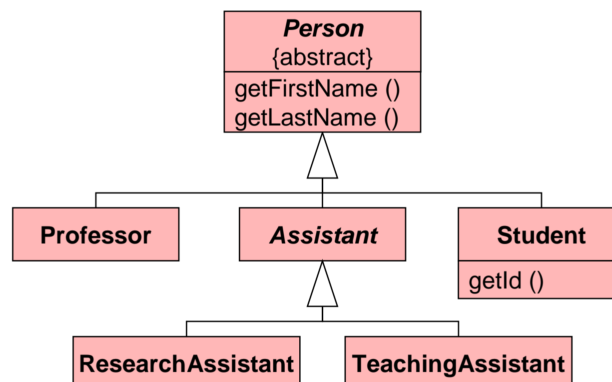
- Provides (part of) the information in event objects passed to the `udpate()` method.
- Example:



Visitor Pattern (1)

Category: behavioral.

Example: let the following classes be given.



The task is to handle collections of persons, e.g., printing out the names of persons.

© Hans-Werner Sehring, STS. Alle Rechte vorbehalten.

Visitor Pattern (2)

Imperative approach, naïve:

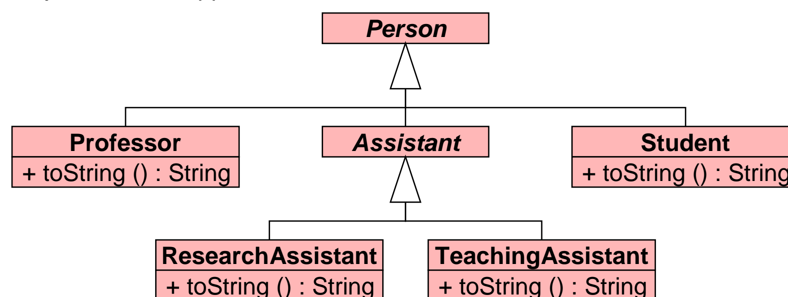
```
Set<Person> personSet = ... a set of Persons
Iterator<Person> i = personSet.iterator () ;
while (i.hasNext ()) {
  Person p = i.next () ;
  String name = p.getFirstName () + " " + p.getLastName () ;
  if (p instanceof Professor)
    System.out.println("Prof. " + name) ;
  else if (p instanceof Assistant)
    System.out.println (name) ;
  else if (p instanceof Student) {
    Student s = (Student)p ;
    System.out.println (name + "," + s.getId ()) ;
  }
  else
    ; // what to do???
}// while
```

Problems:

- hard-coded subtypes ⇒ hard to add new Person types
- hard-coded behavior ⇒ what to do if there are different output formats?

Visitor Pattern (3)

More object-oriented approach: use the `toString()` method of `java.lang.Object`.



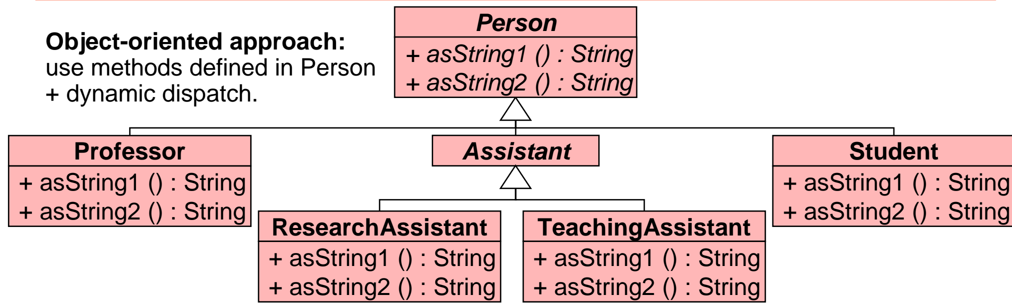

```
Set personSet = ... a set of Persons
Iterator i = personSet.iterator () ;
while (i.hasNext ())
  System.out.println (i.next ().toString ()) ;
```

Problem: hard-coded behavior ⇒ what if there are different output formats?

 © Hans-Werner Sehring, STS. Alle Rechte vorbehalten.

Visitor Pattern (4)

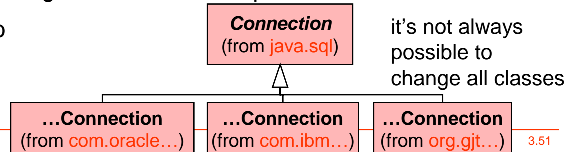
Object-oriented approach:
use methods defined in Person
+ dynamic dispatch.




```
Set<Person> personSet = … a set of Persons
Iterator<Person> i = personSet.iterator () ;
while (i.hasNext ())
  System.out.println (i.next ().asString1 ()) ;
```

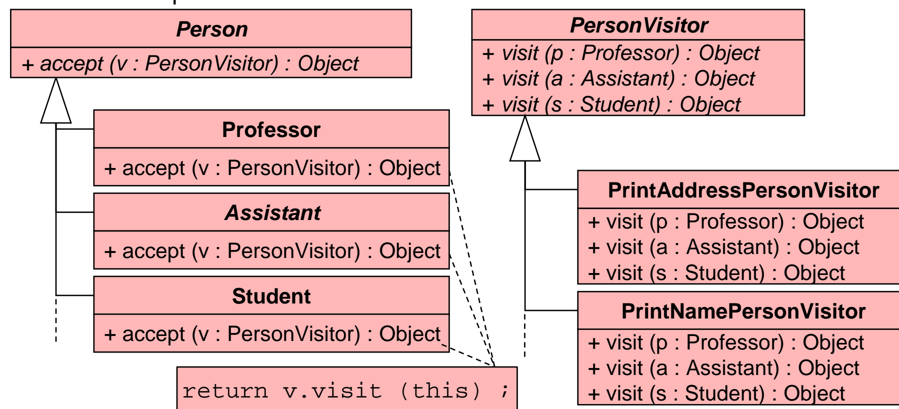
Problem: fixed set of behaviors ⇒ adding new behaviors requires to

- change class Person and thus to
- recompile *all* classes



Visitor Pattern (5)

Solution: encapsulate behaviors in classes of their own ⇒ Visitors.



Double dispatch: behavior depends on type of object and on Visitor.

Behavior given by subclass of `PersonVisitor` ⇒ extendable by introducing new ones.

 © Hans-Werner Sehring, STS. Alle Rechte vorbehalten.

Visitor Pattern (6)

Visitor definition:

```
class PrintNamePersonVisitor implements PersonVisitor {
  private String printName (Person p) {
    return p.getFirstName () + " " + p.getLastName () ;
  }// printName
  public Object visit (Professor p) {
    System.out.println ("Prof. " + printName (p)) ;
    return null ;
  }// visit
  public Object visit (Assistant a) {
    System.out.println (printName (a)) ;
    return null ;
  }// visit
  public Object visit (Student s) {
    System.out.println (printName (s) + ", " + s.getId ()) ;
    return null ;
  }// visit
}// class PrintNamePersonVisitor
```


Visitor Pattern (7)

Usage of a visitor (client code):

- Print names:

```
Set<Person> personSet = … a set of Persons
Iterator<Person> i = personSet.iterator () ;
PersonVisitor pnpv = new PrintNamePersonVisitor () ;
while (i.hasNext ())
  i.next ().accept (pnpv) ;
```

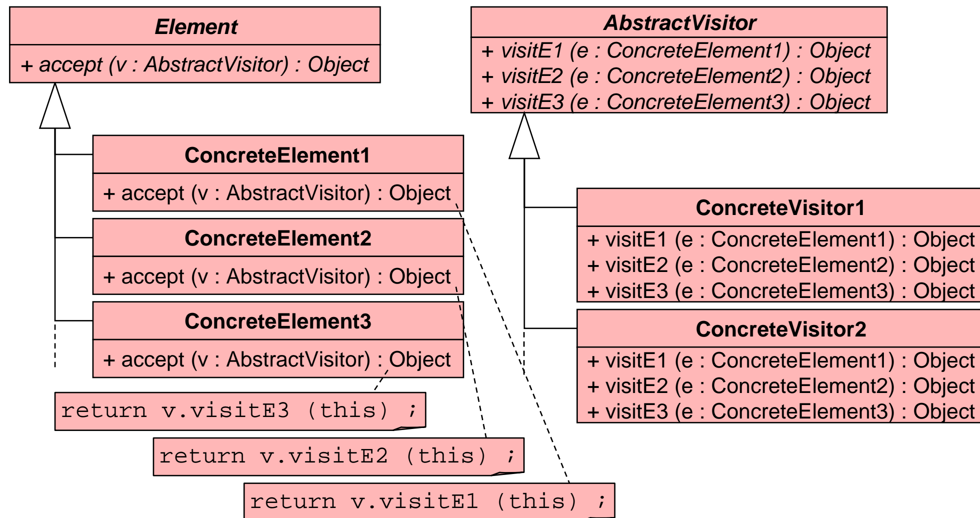
- Print addresses:

```
Set<Person> personSet = … a set of Persons
Iterator<Person> i = personSet.iterator () ;
PersonVisitor papv = new PrintAddressPersonVisitor () ;
while (i.hasNext ())
  i.next ().accept (papv) ;
```

 © Hans-Werner Sehring, STS. Alle Rechte vorbehalten.

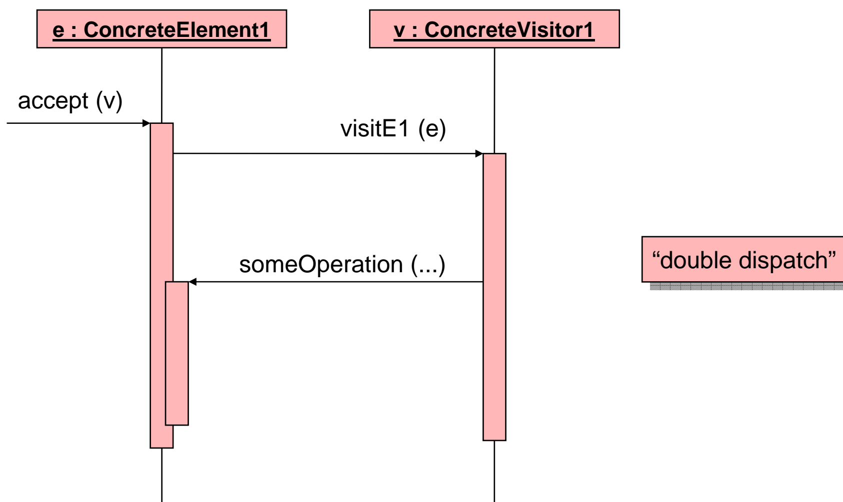
Visitor Pattern (8)

General structure:



Visitor Pattern (9)

Interactions:



"double dispatch"

© Hans-Werner Sehring, STS. Alle Rechte vorbehalten.

Visitor Pattern (10)

Advantages:

- Visitors make it easy to add new operations.
  - This is possible even for existing code not available in source format.
  - Just the `accept()` method has to be included.
- Visitors encapsulate related operations and separate unrelated ones.
  - They avoid interfaces becoming to “broad”.
  - They allow different implementations of a functionality.
- Visitors can add a state for visiting processes.
  - Therefore, no parameter or similar has to be added to the elements.
  - This allows algorithms to walk structures, compute cumulative values, etc.

Visitor Pattern (11)

Disadvantages:

- With Visitors the addition of new classes becomes harder. When adding a new subclass it is required to …
  - add a visit method in the Visitor interface and
  - adjust all Visitor implementations accordingly.
  - Still, with visitors …
    - you get compile time errors for missing operations (in contrast to cascades of conditional statements), and
    - you can supply a standard method for all Visitors (if base visitor type is given by an abstract class).
- A visitor breaks up encapsulation. Often, methods have to be exposed (made public) to be accessible by a visitor, which should not be called otherwise.

© Hans-Werner Sehring, STS. Alle Rechte vorbehalten.

Visitor Pattern (12)

With languages like Java up to version 1.4 (without class parameters, type parameters, and overloading) there is a problem determining the type of objects returned by `accept()` / `visit()`.

```
String name = (String)person.accept (new PersonVisitor () {
  public Object visit (Professor p) ) {
    return "Prof. " + printName (p) ;
  }// visit
  public Object visit (Assistant a) {
    return printName (a) ;
  }// visit
  public Object visit (Student s) {
    return printName (s) + "," + s.getId () ;
  }// visit
}) ;
```

type cast needed
⇒ another visitor??

Visitor Pattern (13)

With class or type parameters, e.g. in Java 5, this can be avoided:

- Visitor definition:

```
abstract class interface PersonVisitor<T> {
  private String printName (Person p) {
    return p.getFirstName () + " " + p.getLastName () ;
  }// printName
  abstract public T visit (Professor p) ;
  abstract public T visit (Assistant a) ;
  abstract public T visit (Student s) ;
}// class PersonVisitor<T>
```

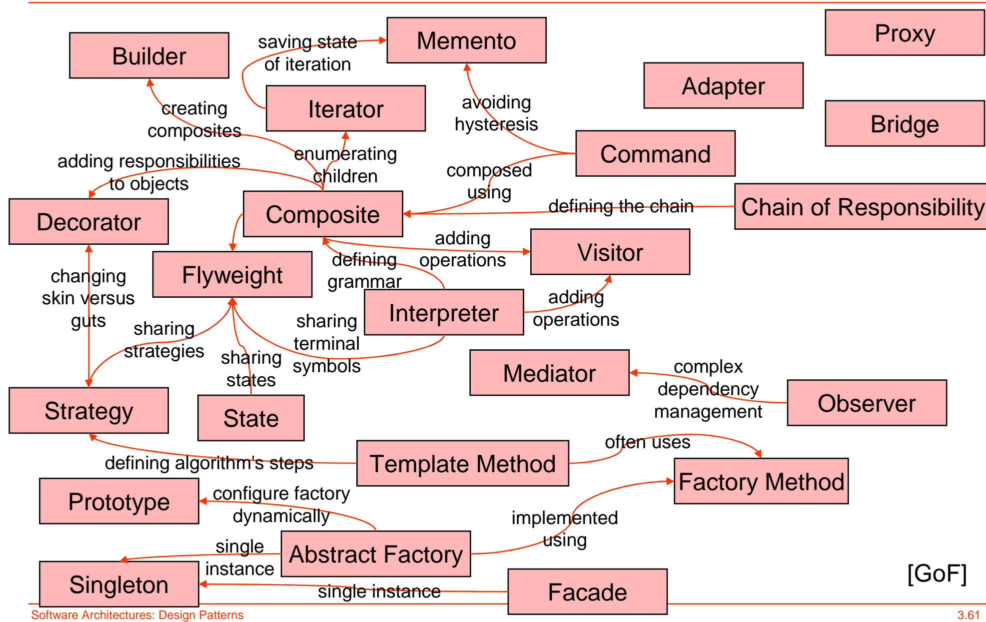
PersonVisitor (T)
+ visit (p : Professor) : T
+ visit (a : Assistant) : T
+ visit (s : Student) : T

- Client code:

```
String name = person.accept (new PersonVisitor<String> () {
  public String visit (Professor p) ) {
    return "Prof. " + printName (p) ;
  }// visit
  public String visit (Assistant a) {
    return printName (a) ;
  }// visit
  public String visit (Student s) {
    return printName (s) + "," + s.getId () ;
  }// visit
}) ;
```

© Hans-Werner Sehring, STS. Alle Rechte vorbehalten.

Design Pattern Relationships (1)



Design Pattern Relationships (2)

Abstract Factory using Prototypes:

```
class UIFactory {

  private Object winProto, txtProto ;

  public UIFactory (Window winProto, Component txtProto) {
    this.winProto = winProto ;
    this.txtProto = txtProto ;
  }// constructor

  public Window createWindow () {
    return (Window)winProto.clone () ;
  }// createWindow

  public Component createTextField () {
    return (Component)txtProto.clone () ;
  }// createTextField

}// class UIFactory
```

© Hans-Werner Sehring, STS. Alle Rechte vorbehalten.

Design Pattern Relationships (3)

Combinations with Singleton:

- Abstract Factory as Singleton:
```
Window    win = UIFactory.getInstance ().createWindow () ;
Component txt = UIFactory.getInstance ().createTextField () ;
```
- Façade as Singleton:
```
MenuItemActivationFacade.getInstance ().setActive (false) ;
```


Design Pattern Relationships (4)

Pattern combinations not discussed in [GoF]:

- Bridge to Flyweights:
```
PersonImpl personImpl = new JDBCPersonImpl (4711) ;
Person     peter              = new Person (personImpl) ;
Professor peterTheProfessor  = new Professor (personImpl) ;
…
```
- Decoration to observe objects not designed for Observer pattern
```
class ObservableC extends C {

  private Set observers ;
  …
  public void setX (T x) {
    super.setX (x) ;
    notify () ;
  }// setX
  private void notify () {
    Iterator i = observers.iterator () ;
    while (i.hasNext ())
      ((Observer)i.next ()).update () ;
  }// notify

  …

}// class ObservableC
```

© Hans-Werner Sehring, STS. Alle Rechte vorbehalten.

Applicability of Patterns

Design Patterns are usually thought as a programming guideline.

The basic principles can also be found on more abstract levels of design:

- The *Mediator Architecture* [Wiederhold] consists of
  - Mediators (see Mediator pattern) and
  - Wrappers (see Adapter pattern).
- *Wrapping* of legacy applications (e.g., by stubs defined in CORBA IDL) corresponds to the Adapter pattern.
- Proxies are also found as components
  - Network proxies
  - Proxy web servers
- …