13) Entwurfsmuster (Design Patterns) - Eine Einführung

Prof. Dr. Uwe Aßmann
Lehrstuhl
Softwaretechnologie
Fakultät für Informatik
TU Dresden
11-0.2, 5/20/11

1) Patterns for Variability
2) Patterns for Extensibility
3) Patterns for Glue
4) Patterns in AWT
Sehr Empfohlen


  Compact introduction into patterns.
Sonstige Literatur

- [The GOF (Gang of Four) Book] E. Gamma, R. Johnson, R. Helm, J. Vlissides. Design Patterns. Addison-Wesley.
- Papiere:
History: How to Write Beautiful Software

- Beginning of the 70s: the window and desktop metaphors (conceptual patterns) are discovered by the Smalltalk group in Xerox Parc, Palo Alto
- 1978/79: Goldberg and Reenskaug develop the MVC pattern for user Smalltalk interfaces at Xerox Parc
  - During porting Smalltalk-78 for the Eureka Software Factory project
- 1979: Alexander's Timeless Way of Building
  - Introduces the notion of a pattern and a pattern language
- 1987: W. Cunningham, K. Beck OOPSLA paper “Using Pattern Languages for Object-Oriented Programs” discovered Alexander's work for software engineers by applying 5 patterns in Smalltalk
  - Working with ET++, one of the first window frameworks of C++
  - At the same time, Vlissides works on InterViews (part of Athena)
- 1991: Pattern workshop at OOPSLA 91, organized by B. Anderson
- 1993: E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Abstraction and Reuse of Object-Oriented Design. ECOOP 97, LNCS 707, Springer
- 1995: First PLOP conference (Pattern Languages Of Programming)
- 1995: GOF book
The Most Popular Definition

- A Design Pattern is
  - A description of a standard solution for
  - A standard design problem
  - In a certain context

- Goal: Reuse of design information
  - A pattern must not be “new”!
  - A pattern writer must have a “aggressive disregard for originality”

- In this sense, patterns are well-known in every engineering discipline
  - Mechanical engineering
  - Electrical engineering
  - Architecture
How do I display and edit a data structure on the screen?
- Reaction on user inputs?
- Maintaining several views
- Adding and removing new views

Solution: Model-View-Controller pattern (MVC), a set of classes to control a data structure behind a user interface
- Developed by Goldberg/Reenskaug in Smalltalk 1978
MVC is a set of classes to control a data structure behind a user interface.
MVC is a complex design pattern and combines the simpler ones Observer, Composite, Strategy.

- Relation between Model and Views is grasped by Observer (asynchronous communication)
- Relation between Controller and Views by Strategy (variation of updating the screen)
- Relation within Views by Composite (tree-formed views)
Design Pattern Model/View/Controller (MVC)

- UML has a notation for patterns (collaboration classes)
  - With role identifiers
Pattern 1: Observer

- Views may register at the model
  - They become observers of the model
  - They are notified if the model changes.
  - Then, every view updates itself by accessing the data of the model.

- Views are independent of each other
  - The model does not know how views visualize it
  - Observer decouples views strongly
Structure Observer

**Subject**
- register(Observer)
- unregister(Observer)
- notify()

**ConcreteSubject**
- getState()
- setState()
- SubjectState

**Observer**
- update()

**ConcreteObserver**
- update()
- ObserverState

ObserverState = Subject.getState()

for all b in observers {
    b.update()
}

return SubjectState
Sequence Diagram Observer

- Update() does not transfer data, only an event (anonymous communication possible)
- Observer pulls data out itself
- Due to pull of data, subject does not care nor know, which observers are involved: subject independent of observer
Views may be nested (Composite)

- Composite represents trees
- For a client class, Compositum unifies the access to root, inner nodes, and leaves
- In MVC, views can be organized as Composite
Composite has an recursive n-aggregation to the superclass.

```java
Component

commonOperation()
add(Component)
remove(Component)
getType(int)

Leaf

commonOperation()

Composite

commonOperation()
add(Component)
remove(Component)
getType(int)

Client

for all g in childObjects
g.commonOperation()

Pseudo implementations

* childObjects
```
Pattern 3: Strategy

The relation between controller and view is a Strategy pattern.

- There may be different control strategies
  - Lazy or eager update of views
  - Menu or keyboard input
- A view may select subclasses of Controller, even dynamically
- No other class changes
Different Forms of Patterns

- Conceptual Patterns (of system structures)
  - Desktop pattern, Wastebasket pattern
- Design Patterns for design structures
  - Product Line Patterns
  - Architectural style
  - Antipatterns (“bad smells”)
- Implementation Patterns (programming patterns, idioms, workarounds)
- Process Patterns for software development processes
- Reengineering Patterns
- Organizational Patterns for company structuring

A **pattern** is the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts

[Riehle/Zülinghoven, Understanding and Using Patterns in Software Development]
What Does a Design Pattern Contain?

► A part with a “bad smell”
  - A structure with a bad smell
  - A query that proved a bad smell
  - A graph parse that recognized a bad smell

► A part with a “good smell” (standard solution)
  - A structure with a good smell
  - A query that proves a good smell
  - A graph parse that proves a good smell

► A part with “forces”
  - The context, rationale, and pragmatics
  - The needs and constraints

forces

“bad smell” ----> “good smell”
Refactorings Transform Antipatterns (Defect Patterns, Bad Smells) Into Design Patterns

- Software can contain bad structure
- A DP can be a goal of a *refactoring*, transforming a bad smell into a good smell

![Diagram showing the process from Defect pattern (Bad smell) to Design pattern (good smell) through Refactoring 1, 2, and 3]
Structure for Design Pattern Description (GOF Form)

- Name (incl. Synonyms) (also known as)
- Motivation (purpose)
  - also “bad smells” to be avoided
- Employment
- Solution (the “good smell”)
  - Structure (Classes, abstract classes, relations): UML class or object diagram
  - Participants: textual details of classes
  - Interactions: interaction diagrams (MSC, statecharts, collaboration diagrams)
  - Consequences: advantages and disadvantages (pragmatics)
  - Implementation: variants of the design pattern
  - Code examples
- Known Uses
- Related Patterns
Purposes of Design Patterns

► Improve communication in teams
  ▪ Between clients and programmers
  ▪ Between designers, implementers and testers
  ▪ For designers, to understand good design concepts

► Design patterns create a glossary for software engineering (an “ontology of software design”)
  ▪ A “software engineer” without the knowledge of patterns is a programmer

► Design patterns document abstract design concepts
  ▪ Patterns are “mini-frameworks”
  ▪ Documentation: in particular frameworks are documented by design patterns
  ▪ Prevent re-invention of well-known solutions
  ▪ Design patterns capture information in reverse engineering
  ▪ Improve code structure and hence, code quality
Standard Problems to Be Solved By
Product Line Patterns

► Variability
  ▪ Exchanging parts easily
  ▪ Variation, variability, complex parameterization
  ▪ Static and dynamic
  ▪ For product lines, framework-based development

► Extensibility
  ▪ Software must change

► Glue (adaptation overcoming architectural mismatches)
  ▪ Coupling software that was not built for each other
13.1) Patterns for Variability
Commonalities and Variabilities

- Design patterns often center around
  - Things that are common to several applications
  - Things that are different from application to application
- Commonalities lead to frameworks or product lines
- Variabilities to products of a product line
- For the communality/variability knowledge, Pree invented a template-and-hook (T&H) concept
  - Templates contain skeleton code (commonality), common for the entire product line
  - Hooks (hot spots) are placeholders for the instance-specific code (variability)
Define the skeleton of an algorithm (template method)

- The template method is concrete

Delegate parts to abstract hook methods that are filled by subclasses

Implements template and hook with the same class, but different methods

Allows for varying behavior

- Separate invariant from variant parts of an algorithm

TemplateMethod Pattern (Wdh.)

```
AbstractClass
   TemplateMethod()
   primitiveOperation1()
   primitiveOperation2()

ConcreteClass
   primitiveOperation1()
   primitiveOperation2()
```
Example: A Data Generator

- Parameterizing a data generator by frequency and kind of production

```java
... for (i = 0; i < howOften(); i++) {
    ... produceItem();
    ...
}
```

```
String word = grammar.recurse();
println(word);
```

```
return 5;
```

```
TestDataGenerator
- Grammar grammar;
howOften() produceItem()
```

```
DataGenerator
generate()
howOften() produceItem()
```
Variability with TemplateMethod

► Binding the hook means to
  ▪ Derive a concrete subclass from the abstract superclass, providing the implementation of the hook method

► Controlled extension by only allowing for binding hook methods, but not overriding template methods

Template method

Binding the template method hooks with hook values (method implementations)
FactoryMethod

- FactoryMethod is a variant of TemplateMethod

```
FactoryMethod()
anOperation()
```

ConcreteCreatorA

```
ConcreteCreatorA()
return new ConcreteProductA
```

ConcreteCreatorB

```
ConcreteCreatorB()
return new ConcreteProductB
```

Product

```
Product = FactoryMethod()
```

Creator
Strategy (Template Class)
T&H on the Level of Classes

- Methods can be reified, i.e., represented as objects
  - In the TemplateMethod, the hook method can be split out of the class and put into a separate object
- We hand out additional roles for some classes
  - The template role
  - The hook role
- Resulting patterns:
  - Strategy (Template Class)
  - Bridge (Dimensional Class Hierarchies) for variability with parallel class hierarchies
Strategy (also called Template Class)

- The template method and the hook method are found in different classes
- Similar to TemplateMethod, but
  - Hook objects and their hook methods can be exchanged at run time
  - Exchanging several methods (a set of methods) at the same time
  - Consistent exchange of several parts of an algorithm, not only one method
- This pattern is basis of Bridge, Builder, Command, Iterator, Observer, Visitor.
Actors and Genres as Template Method

- Binding an Actor's hook to be a ShakespeareActor

```
Actor
play()
recite()
dance()

ShakespeareActor
recite()
dance()
```

```java
... recite();
... dance();
...```
Consistent exchange of recitation and dance behavior possible

```
... realization.recite();
... realization.dance();
... 
```

Diagram:
- Actor
  - play()
- CinemActor
  - recite()
  - dance()
- TVActor
  - recite()
  - dance()
- ShakespeareActor
  - recite()
  - dance()

UML Class Diagram:
- Actor
  - realization
  - play()
- ActorRealization
  - recite()
  - dance()
Example for Template Class (Strategy)

- Encapsulate formatting algorithms

```
TextApplication
  traverse()
  repair()

Formatter
  format()

Template

Hook

SimpleFormatter
  format()

TeXFormatter
  format()

ArrayFormatter
  format()
```

```
TextApplication
  formatter.format()

Formatter
  formatter

Template

Hook
```
Variability with Strategy

- Binding the hook means to
  - Derive a concrete subclass from the **abstract hook superclass**, providing the implementation of the hook method

![Diagram showing the relationship between Template Class, Hook Class, and template method hooks with hook values (method implementations).]
Factory Class (Abstract Factory)

- Allocate a family of products

```
Client
init()

AbstractFactory
createProductA()
createProductB()

ConcreteFactory1
createProductA()
createProductB()

ConcreteFactory2
createProductA()
createProductB()

AbstractProductA

AbstractProductB

ProductA1
ProductA2

ProductB1
ProductB2

If (...) {
    factory = new ConcreteFactory1();
} else {
    factory = new ConcreteFactory2();
}
```
Bridge (Dimensional Class Hierarchies)
The left hierarchy is called **abstraction hierarchy**, the right **implementation**

- **Abstraction**
  - operation()
  - MoreConcrete AbstractionA
    - operation()
    - operationImpl()
    - Some actions for A; imp.operationImpl()
  - MoreConcrete AbstractionB
    - operation()
    - operationImpl()
    - Some actions for B; imp.operationImpl()

- **Implementation**
  - operationImpl()
  - ConcretImplA
    - operationImpl()
  - ConcretImplB
    - operationImpl()
Bridge as Dimensional Class Hierarchies

- comparing it to TemplateMethod

```
TemplateClass

 templateMethod()

hookObject

MoreConcrete
TemplateA

 templateMethod()

Implementation A
.. hookMethod();

MoreConcrete
TemplateB

 templateMethod()

Implementation B
.. hookMethod();

HookClass

 hookMethod()

ConcreteHookClassA

 hookMethod()

ConcreteHookClassB

 hookMethod()
```

Prof. U. Aßmann, Softwaretechnologie
Bridge (Dimensional Class Hierarchies)

- Vary also the template class in a class hierarchy
  - The sub-template classes can adapt the template algorithm
  - Important: the sub-template classes must fulfil the *contract* of the superclass
    - Although the implementation can be changed, the interface and visible behavior must be the same

- Both hierarchies can be varied independently
  - Factoring (orthogonalization)
  - Reuse is increased

- Basis for patterns
  - Observer, Visitor
DataGenerator as Bridge

```
DataGenerator
Data data;
generate()

GeneratorImpl
generateData(Data)

TestDataGenerator
generate()
data = parseTestDataGrammar();
imp.generateData(data);

ReportGenerator
generate()
data = readFromForm();
imp.generateData(data);

ExhaustiveGenerator
generateData(Data)

RandomGenerator
generateData(Data)
```
13.2) Patterns for Extensibility
Composite has an recursive n-aggregation to the superclass

```java
Component
commonOperation()
add(Component)
remove(Component)
getype(int)

Leaf
commonOperation()

Composite
commonOperation()
add(Component)
remove(Component)
getype(int)

for all g in childObjects
  g.commonOperation()
Example: PieceLists in Cars (Wdh.)

Client

CarPart

- int calculateCost()
- addPart(CarPart)

Leaf

- int calculateCost()
- return local my cost;

ComposedCarPart

- int calculateCost()
- addPart(CarPart)

Pseudo implementations

for all g in children
sum += g.calculateCost()
abstract class CarPart {
    int myCost;
    abstract int calculateCost();
}
class ComposedCarPart extends CarPart {
    int myCost = 5;
    CarPart[] children; // here is the n-recursion
    int calculateCost() {
        for (i = 0; i <= children.length; i++) {
            curCost += children[i].calculateCost();
        }
        return curCost + myCost;
    }
    void addPart(CarPart c) {
        children[children.length] = c;
    }
}

class Screw extends CarPart {
    int myCost = 10;
    int calculateCost() {
        return myCost;
    }
    void addPart(CarPart c) {
        /// impossible, dont do anything
    }
}
// application
int cost = carPart.calculateCost();
Part/Whole hierarchies, e.g., nested graphic objects
- Attention: class diagram cannot convey the constraint that the objects form a tree!

Dynamic Extensibility of Composite
- Due to the n-recursion, new children can always be added dynamically into a composite node
- Whenever you have to program an extensible part of a framework, consider Composite

common operations: draw(), move(), delete(), scale()
Decorator
Problem

- How to extend an inheritance hierarchy of a library that was bought in binary form?
- How to avoid that an inheritance hierarchy becomes too deep?

Library

New Features

Decorator with New Features
Decorator Pattern

- A Decorator is a *skin* of another object
- The Decorator *mimics* a class
It is a restricted Composite with a 1-aggregation to the superclass

- A subclass of a class that contains an object of the class as child
- However, only one composite (i.e., a delegatee)
- Combines inheritance with aggregation

```
MimicedClass

mimicedOperation()

ConcreteMimicedClass

mimicedOperation()

Decorator

mimicedOperation()

ConcreteDecoratorA

mimicedOperation()

ConcreteDecoratorB

mimicedOperation()

super.mimicedOperation();
additionalStuff();
```
Ex.: Decorator for Widgets

```
Widget
  draw()

TextWidget
  draw()

WidgetDecorator
  draw()

Frame
  draw()
  super.draw();
  drawFrame();

Scrollbar
  draw()
  super.draw();
  drawScrollbar();
```

Diagram showing the relationship and methods of the classes involved.
Purpose Decorator

- For dynamically extensible objects (i.e., decoratable objects)
  - Addition to the decorator chain or removal possible
Observer – (Event Bridge)
Observer (Publisher/Subscriber, Event Bridge)

- How to react differently in different situations on a state change?
- How to add a reaction to a set of already known reactions?
- How to keep all reactions independent of each other?
- How to hide *who* reacts on a state change?

```
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>60</td>
<td>30</td>
<td>10</td>
</tr>
<tr>
<td>y</td>
<td>50</td>
<td>30</td>
<td>20</td>
</tr>
<tr>
<td>z</td>
<td>80</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th>a c</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>50%</td>
<td>30%</td>
</tr>
<tr>
<td></td>
<td>30%</td>
<td>30%</td>
</tr>
<tr>
<td></td>
<td>20%</td>
<td>10%</td>
</tr>
</tbody>
</table>
```

Notify on change
Pulling out the changed state (state queries)
Structure Observer

**Subject**
- register(Observer)
- unregister(Observer)
- notify()

**ConcreteSubject**
- getState()
- setState()

**ConcreteObserver**
- update()

**Observer**
- update()

For all `b` in observers {
  `b.update()`
}

**ObserverState**
- ObserverState = Subject.getState()

return SubjectState

Difference to Bridge: hierarchies are not complete independent; Observer knows about Subject
Sequence Diagram Observer

- Update() does not transfer data, only an event (anonymous communication possible)
- Observer pulls data out itself
- Due to pull of data, subject does not care nor know, which observers are involved: subject independent of observer

```
setState()
notify()
update 1()
getState()
update n()
```

```
register()
```

```
```

```
```
Observer with ChangeManager (EventBus)

Subject
- register(Observer)
- unregister(Observer)
- notify()

Observer with ChangeManager
- register(Subject, Observer)
- unregister(Subject, Observer)
- notify()

SimpleChangeManager
- register(Subject, Observer)
- unregister(Subject, Observer)
- notify()

DAGChangeManager
- register(Subject, Observer)
- unregister(Subject, Observer)
- notify()

Subject-Observer-mapping
- update (Subject)
- for all s in Subjects
  - for all b in s.Observer
  - b.update (s)

mark all observers to be updated
- update all marked observers
Observer with ChangeManager is also Called Event-Bus

- Basis of many interactive application frameworks (Xwindows, Java AWT, Java InfoBus, ....)
- Loose coupling in communication
  - Observers decide what happens
- Dynamic extension of communication
  - Anonymous communication
  - Multi-cast and broadcast communication
Implementation of a dimensional structure

- First dispatch on dimension 1 (data structure), then on dimension 2 (algorithm)

```
DataStructure

acceptVisitor(Visitor)

Visitor

runWithDataA(DataStructure)
runWithDataB(DataStructure)

ConcreteVisitorA

runWithDataA(DataStructure)
runWithDataB(DataStructure)

ConcreteVisitorB

runWithDataA(DataStructure)
runWithDataB(DataStructure)

ConcreteDataStructureA

acceptVisitor(Visitor)

v.runWithDataA(this);

ConcreteDataStructureB

templateMethod(Visitor)

v.runWithDataB(this);

ConcreteDataStructureA

acceptVisitor(Visitor)

v.runWithDataA(this);
```
Working on Abstract Syntax Trees with Visitors

- **Program** → **Node**
  - `accept(NodeVisitor)`

- **AssignmentNode**
  - `accept(NodeVisitor b)`
  - `b.visitAssignment(this)`

- **VariableRefNode**
  - `accept(NodeVisitor)`
  - `b.visitVariableRef(this)`

- **NodeVisitor**
  - `visitAssignment(AssignmentNode)`
  - `visitVariableRef(VariableRefNode)`

- **TypeCheckingVisitor**
  - `visitAssignment(AssignmentNode)`
  - `visitVariableRef(VariableRefNode)`

- **CodeGenerationVisitor**
  - `visitAssignment(AssignmentNode)`
  - `visitVariableRef(VariableRefNode)`
Sequence Diagram Visitor

- First dispatch on data, then on visitor

```
aConcreteClient  aConcreteDataObject  aConcreteVisitor

accept(aConcreteVisitor)               acceptDataObject(aConcreteVisitor)
```

Dispatch 1

Dispatch 2
13.3) Patterns for Glue - Bridging Architectural Mismatch
Strukturmuster Singleton (dt.: Einzelinstanz)

Problem:
- Speicherung von globalem Zustand einer Anwendung
- Sicherstellung, daß von einer Klasse genau ein Objekt besteht

<table>
<thead>
<tr>
<th>Singleton</th>
</tr>
</thead>
<tbody>
<tr>
<td>– theInstance: Singleton</td>
</tr>
<tr>
<td>getInstance(): Singleton</td>
</tr>
</tbody>
</table>

```java
class Singleton {
    private static Singleton theInstance;
    private Singleton () {}  
    public static Singleton getInstance() {
        if (theInstance == null)
            theInstance = new Singleton();
        return theInstance;
    }
}
```

Explicierter Konstruktor wird (für andere Klassen) unbenutzbar gemacht!
Adapter
Object Adapter

- An object adapter is a kind of a proxy
  - That maps one interface, protocol, or data format to another

```
Client

Goal
operation()

Adapter
operation()

AdaptedClass
specificOperation()

adaptedObject.specificOperation()

ObjectAdapter

Adapted class does not inherit from goal

Adaptee

Adapter adapted Object

Decorator-like inheritance
```
Example: Use of an External Class Library For Texts

**Diagram:**
- **DrawingEditor**
  - *GraficObject*
    - frame()
    - createManipulator()
  - Text
    - frame()
    - createManipulator()
    - return new TextManipulator
  - Linie
    - frame()
    - createManipulator()
  - TextDisplay
    - largeness()
    - return text.largeness()
“Facade” Hides a Subsystem

- A **facade** is an object adapter hiding a complete set of objects (subsystem)
  - The facade has to map its own interface to the interfaces of the hidden objects
Refactoring Towards a Facade

Clients

Subsystem

Facade
If classes of the subsystem are again facades, **layers** result

- Layers are nested facades
Instead of delegation, class adapters use multiple inheritance.

Class Adapter

- GoalClass
  - operation()

- AdaptedClass
  - specificOperation()

- Adapter
  - operation()
  - specificOperation()
In the SalesPoint framework (project course), a ClassAdapter is used to embed an Action class in an Listener of Observer Pattern.

```
if (!action.oclUndefined()) {
    action.doAction (p, sp);
}
```
Other Patterns
What is discussed elsewhere...

- Iterator
- Composite
- TemplateMethod
- Command

- Chapter “Analysis”:
  - State (Zustand), IntegerState, Explicit/ImplicitIntegerState

- Chapter “Architecture”:
  - Facade (Fassade)
  - Layers (Schichten)
  - 4-tier architecture (4-Schichtenarchitektur, BCED)
  - 4-tier abstract machines (4-Schichtenarchitektur mit abstrakten Maschinen)
For the exam will be needed:

- Singleton
- State
- IntegerState
- ImplicitIntegerState
- ExplicitIntegerState
- TemplateMethod
- Strategy
- Bridge
- Command
- Iterator
- Visitor
- Observer
- FactoryMethod
- AbstractFactory
- Decorator
- Adapter
- Composite
- Facade
Other Important GOF Patterns

Variability Patterns
► Visitor: Separate a data structure inheritance hierarchy from an algorithm hierarchy, to be able to vary both of them independently
► AbstractFactory: Allocation of objects in consistent families, for frameworks which maintain lots of objects
► Builder: Allocation of objects in families, adhering to a construction protocol
► Command: Represent an action as an object so that it can be undone, stored, redone

Extensibility Patterns
► Proxy: Representant of an object
► ChainOfResponsibility: A chain of workers that process a message

Others
► Memento: Maintain a state of an application as an object
► Flyweight: Factor out common attributes into heavy weight objects and flyweight objects
► Iterator: iterate over a collection
14.4 Design Patterns in a Larger Library
Design Pattern in the AWT

- AWT (Abstract Window Toolkit) is part of the Java class library
  - Uniform window library for many platforms (portable)
- Employed patterns
  - Compositum (widgets)
  - Strategy: The generic composita must be coupled with different layout algorithms
  - Singleton: Global state of the library
  - Bridge: Widgets such as Button abstract from look and provide behavior
    - Drawing is done by a GUI-dependent drawing engine (pattern bridge)
  - Abstract Factory: Allocation of widgets in a platform independent way
What Have We Learned?

► Design Patterns grasp good, well-known solutions for standard problems

► Variability patterns allow for variation of applications
  - They rely on the template/hook principle

► Extensibility patterns for extension
  - They rely on recursion
  - An aggregation to the superclass
  - This allows for constructing runtime nets: lists, sets, and graphs
  - And hence, for dynamic extension

► Architectural Glue patterns map non-fitting classes and objects to each other
The End

- Design patterns and frameworks, WS, contains more material.
- © Uwe Aßmann, Heinrich Hussmann, Walter F. Tichy, Universität Karlsruhe, Germany, used by permission
Proxy
Proxy

- Hide the access to a real subject by a representant

Object Structure:
Proxy

- The proxy object is a representant of an object
  - The Proxy is similar to Decorator, but it is not derived from ObjectRecursion
  - It has a direct pointer to the sister class, not to the superclass
  - It may collect all references to the represented object (shadows it). Then, it is a facade object to the represented object
- Consequence: chained proxies are not possible, a proxy is one-and-only
- It could be said that Decorator lies between Proxy and Chain.
Proxy Variants

- **Filter proxy** (smart reference):
  - executes additional actions, when the object is accessed

- **Protocol proxy**:
  - Counts references (reference-counting garbage collection)
  - Or implements a synchronization protocol (e.g., reader/writer protocols)

- **Indirection proxy** (facade proxy):
  - Assembles all references to an object to make it replaceable

- **Virtual proxy**: creates expensive objects on demand

- **Remote proxy**: representant of a remote object

- **Caching proxy**: caches values which had been loaded from the subject
  - Caching of remote objects for on-demand loading

- **Protection proxy**
  - Firewall proxy
Adapters for COTS

► Adapters are often used to adapt components-off-the-shelf (COTS) to applications
► For instance, an EJB-adapter allows for reuse of an Enterprise Java Bean in an application
EJB Adapter

BillingApplication

* Bill

- addItem(Item)
- calculateSum()

OtherBill

- addItem(Item)
- calculateSum()

EJBBill

- fetchBean()
- addItem(Item)
- calculateSum()

Client interface:

- EJBHome
- EJBOBJECT
- Metadata
- Handle

EJBHome

- getBean()

EJBOBJECT

EJBMetaData

EJBHandle

EJBObject

- contact EJBHome for EJB...
- if not there, create EJBOBJECT

- EJBOBJECT = fetchBean();
- addItem(EJBOBJECT, Item)

- EJBOBJECT = fetchBean();
- sum up (EJBOBJECT)