

Pattern Oriented Software Architecture

Prof. Dr. Jörg Schäfer
Frankfurt University of Applied Sciences

Fachbereich 2 Informatik und Ingenieurwissenschaften
Computer Science and Engineering

<http://www.informatik.fb2.fh-frankfurt.de/~jschaefer/>

Copyright © 2009–2014 Dr. Jörg Schäfer

13.10.2014

Abstract

This module *Unit 6: “Pattern Oriented Software Architecture”* covers patterns and usage of patterns in software architectures. This script only serves as a supplement to the lectures and does *not* replace a textbook. The module encourages active participation and active learning, henceforth, this script should be merely regarded as an extended table of content and a reference to the literature!

Contents

Contents	ii
List of Figures	ix
List of Tables	x
Prologue	xi
0.1 General Description	xi
0.1.1 Contents	xi
0.1.2 Precondition and Credit	xii
0.2 Educational Concept	xiii
0.2.1 Educational Objectives	xiii
0.2.2 Learning Targets	xiii
0.2.3 Learning Methodology	xiv
0.2.4 A Word of Caution – Programming Skills are a Must!	xv
0.2.5 Wiki Content and Structure	xv
0.3 Structure	xvi
0.3.1 Learning Units	xvi
0.3.2 Exercises	xviii
0.4 Literature	xxi
0.4.1 Books	xxi
0.4.2 Useful Web References	xxi
0.4.3 Script	xxii
0.5 Lecture: Summary	xxii
0.6 Exercise: Discover Patterns	xxii
1 Unit 1 – Introduction	1
1.1 Lecture: Motivation	1
1.2 Exercise: Pair Work	2
1.3 Lecture: Strategy Pattern	2
1.3.1 Name	2
1.3.2 Intent	2
1.3.3 Problem	2

1.3.4	Solution	2
1.4	Exercise: What are the dependencies created?	3
1.4.1	Real World Example	3
1.4.2	Additional Topics	3
1.4.3	Learning Goals	4
1.5	Lecture: Summary	4
1.6	Exercise: Time to Code!	4
1.7	Exercise: Questions for Self-Study	5
2	Unit 2 – Behavioral Patterns	6
2.1	Lecture: Recap (Students)	6
2.2	Exercise: Plenum Discussion	6
2.3	Lecture: Brewing Coffee and Tea	7
2.4	Exercise: Pair Work	7
2.5	Lecture: Template Pattern	7
2.5.1	Name	7
2.5.2	Intent	7
2.5.3	Problem	8
2.5.4	Solution	8
2.5.5	Real World Example	8
2.6	Lecture: Template in the Wild	9
2.6.1	Additional Topics	9
2.6.2	Learning Goals	9
2.7	Lecture: Command Pattern	9
2.7.1	Name	10
2.7.2	Intent	10
2.7.3	Problem	10
2.7.4	Solution	10
2.7.5	Real World Example	10
2.7.6	Additional Topics	11
2.7.7	Learning Goals	12
2.8	Lecture: Summary	12
2.9	Exercise: Time to Code!	12
2.10	Exercise: Questions for Self-Study	13
3	Unit 3 – Structural Patterns	14
3.1	Lecture: Recap (Students)	14
3.2	Exercise: Plenum Discussion	14
3.3	Lecture: Iterator Pattern for Walking a Structure	15
3.3.1	Name	15
3.3.2	Intent	15
3.3.3	Problem	15
3.3.4	Solution	15
3.3.5	Real World Example	16

3.3.6	Additional Topics	16
3.3.7	Learning Goals	16
3.4	Lecture: Composite Pattern	17
3.4.1	Name	17
3.4.2	Intent	17
3.4.3	Problem	17
3.4.4	Solution	18
3.4.5	Real World Example	18
3.4.6	Additional Topics	19
3.4.7	Learning Goals	19
3.5	Exercise: Time to Code!	19
3.6	Exercise: Questions for Self-Study	19
4	Unit 4 – Case Study JUnit	21
4.1	Lecture: Recap (Students)	21
4.2	Exercise: Plenum Discussion	21
4.3	Lecture: A Cook’s Tour	22
4.4	Exercise: Time To Code!	22
4.5	Exercise: Questions for Self-Study	23
5	Unit 5 – Pattern Theory	24
5.1	Lecture: Recap JUnit (Students)	25
5.2	Exercise: Plenum Discussion	25
5.3	Lecture: Pattern Origins	25
5.3.1	Origin	25
5.3.2	Pattern Examples	26
5.4	Exercise: Pair Work	27
5.5	Lecture: Software Patterns	27
5.5.1	Pattern Language	29
5.5.2	Pattern Categories	29
5.5.3	Criticism	31
5.6	Exercise: Pair Work	33
5.6.1	Learning Goals	33
5.7	Exercise: Excursion!	33
5.8	Lecture: Summary	33
5.9	Exercise: Questions for Self-Study	34
6	Unit 6 – OO-Principles I	35
6.1	Lecture: Recap (Students)	35
6.2	Exercise: Plenum Discussion	35
6.3	Exercise: Design Smells	36
6.4	Lecture: Design Smells	36
6.5	Lecture: The Principles	37
6.5.1	SRP: The Single Responsibility Principle	37

6.5.2	OCP: The Open Closed Principle	37
6.5.3	LSP: The Liskov Substitution Principle	37
6.5.4	DIP: The Dependency-Inversion Principle	38
6.5.5	ISP: The Interface-Segregation Principle	38
6.6	Exercise: Time To Code!	39
6.7	Exercise: Questions for Self-Study	39
7	Unit 7 – OO-Principles II	40
7.1	Lecture: Recap (Students)	40
7.2	Exercise: Plenum Discussion	40
7.3	Lecture: Packaging Patterns	41
7.3.1	Modularity	41
7.3.2	Principles of Package Cohesion	43
7.3.3	The Reuse-Release Equivalence Principle (REP)	43
7.3.4	Common-Reuse Principle (CRP)	43
7.3.5	Common-Closure Principle (CCP)	43
7.3.6	The Acyclic-Dependencies Principle (ADP)	44
7.3.7	The Stable-Dependencies Principle (SDP)	44
7.3.8	Metrics	44
7.4	Exercise: Time To Analyze!	45
7.5	Exercise: Questions for Self-Study	46
8	Unit 8 – Creational Patterns	47
8.1	Lecture: Recap (Students)	47
8.2	Exercise: Plenum Discussion	47
8.3	Lecture: Abstract Factory	47
8.3.1	Name	48
8.3.2	Intent	48
8.3.3	Problem	48
8.3.4	Solution	48
8.3.5	Real World Example	48
8.3.6	Additional Topics	49
8.4	Exercise: Pizza Factories	49
8.5	Lecture: Singleton	50
8.5.1	Name	50
8.5.2	Intent	50
8.5.3	Problem	50
8.5.4	Solution	50
8.5.5	Real World Example	51
8.5.6	Additional Topics	51
8.5.7	Learning Goals	51
8.6	Exercise: Time to Code!	52
8.7	Exercise: Questions for Self-Study	52

9	Unit 9 – Patterns for Adding Flexibility	53
9.1	Lecture: Recap (Students)	53
9.2	Exercise: Plenum Discussion	53
9.3	Lecture: Decorator	54
9.3.1	Name	54
9.3.2	Intent	54
9.3.3	Problem	54
9.3.4	Solution	54
9.3.5	Code Example	54
9.3.6	Real World Example	55
9.3.7	Additional Topics	55
9.4	Exercise: Runtime Sequence Diagram	56
9.5	Lecture: Adapter	56
9.5.1	Name	56
9.5.2	Intent	56
9.5.3	Problem	56
9.5.4	Solution	56
9.5.5	Real World Example	57
9.5.6	Additional Topics	57
9.6	Exercise: Time to Code!	58
9.7	Exercise: Questions for Self-Study	58
10	Unit 10 – Decoupling Patterns	59
10.1	Lecture: Recap (Students)	59
10.2	Exercise: Plenum Discussion	59
10.3	Lecture: Observer	60
10.3.1	Name	60
10.3.2	Intent	60
10.3.3	Problem	60
10.3.4	Solution	60
10.3.5	Real World Example	60
10.3.6	Additional Topics	61
10.4	Exercise: Activate Yourself!	61
10.5	Exercise: Time to Sequence!	61
10.6	Lecture: Chain of Responsibility	62
10.6.1	Name	62
10.6.2	Intent	62
10.6.3	Problem	62
10.6.4	Solution	62
10.6.5	Real World Example	63
10.6.6	Servlets, Filters and Chain of Responsibility	63
10.7	Exercise: Live Demo	67
10.8	Exercise: Hollywood Principle	68
10.9	Exercise: Time to Code!	68

10.10	Exercise: Questions for Self-Study	68
11	Unit 11 – Case Study MVC	69
11.1	Lecture: Recap (Students)	69
11.2	Exercise: Plenum Discussion	69
11.3	Lecture: Case Study MVC	70
11.4	Exercise: Time To Code!	70
11.5	Exercise: Case Study MVC and the Web	70
11.6	Exercise: Time To Code!	71
11.7	Exercise: Questions for Self-Study	71
12	Unit 12 – Architectural Patterns	72
12.1	Lecture: Recap (Students)	72
12.2	Exercise: Plenum Discussion	72
12.3	Lecture: Pipes and Filters	73
12.3.1	Ingredients	73
12.3.2	Structure	73
12.3.3	Example	74
12.3.4	CRCs	74
12.3.5	Components	74
12.4	Lecture: Blackboard	75
12.4.1	Structure	76
12.5	Exercise: Develop Sequence Diagramm	77
12.6	Lecture: Usage and Consequences	77
12.7	Exercise: Case Study: Apache Web Server	77
12.8	Exercise: Questions for Self-Study	78
13	Unit 13 – Other Patterns and Idioms	79
13.1	Lecture: Recap (Students)	79
13.2	Exercise: Plenum Discussion	79
13.3	Lecture: Analysis Patterns: Quantity, Measurement, Obser- vation	80
13.3.1	Analysis Patterns	80
13.3.2	Additional Topics	80
13.4	Exercise: Flashlight	80
13.5	Exercise: Idioms	80
13.6	Lecture: Idioms	81
13.7	Exercise: Collect Idioms!	82
13.8	Lecture: Enterprise Patterns	82
13.9	Exercise: Time To Code!	83
13.10	Exercise: Questions for Self-Study	83
14	Unit 14 – Testing Patterns	84
14.1	Lecture: Recap (Students)	84

14.2 Exercise: Plenum Discussion	84
14.3 Lecture: Summary	85
14.4 Exercise: Questions for Self-Study	85
15 Unit 15 – Criticism and Alternatives	86
15.1 Lecture: Recap (Students)	86
15.2 Exercise: Plenum Discussion	86
15.3 Lecture: Criticism 1: Design Patterns are Weakness of Pro- gramming Languages	87
15.3.1 The Object Pattern	87
15.3.2 Additional Topics	88
15.4 Lecture: Criticism 2: Design Patterns in Dynamic Program- ming	89
15.4.1 Dynamic Languages need patterns less often	90
15.4.2 Some Examples	90
15.4.3 First-Class Design Patterns	91
15.5 Exercise: Boilerplate-Code	91
15.6 Lecture: My Final Feedback	92
15.7 Exercise: Questions for Self-Study	92
References	93

List of Figures

1	Design Pattern Overview	xviii
1.1	Strategy Pattern	3
1.2	Application of Strategy Pattern	3
2.1	Template Pattern	8
2.2	Command Pattern	11
2.3	Macro Command Pattern	11
3.1	Iterator Pattern	16
3.2	Composite Pattern	18
4.1	JUnit Patterns	22
5.1	Pattern Window Place (Source: Christopher Alexander) . . .	27
5.2	Stubs and Skeletons	28
5.3	The Big Ball of Mud	32
7.1	The Main Sequence	45
8.1	Abstract Factory Pattern	49
8.2	Singleton	51
9.1	Decorator Pattern	55
9.2	Class Adapter Pattern	57
9.3	Object Adapter Pattern	57
10.1	Observer Pattern	61
10.2	Call Chain	62
10.3	Chain of Responsibility Pattern	63
10.4	Servlet Architecture	64
10.5	Web Application	65
12.1	Pipes and Filter Architecture	73
12.2	Blackboard Pattern	76

List of Tables

1	Learning Units	xvii
2	Exercises	xx
5.1	Pattern Notation	30
5.2	Pattern Category	31

Prologue

There is a central quality which is the root criterion of life and spirit in a man, a town, a building, or a wilderness.

Christopher Alexander



0.1 General Description

0.1.1 Contents

This module *Unit 6: “Pattern Oriented Software Architecture”* covers patterns and usage of patterns in software architectures.

Upon completion of this course, the student is able to:

- understand the motives of the pattern community,
- distinguish between different types of patterns,
- apply patterns in the design of Safety Critical Systems (SCS),
- discover patterns in existing software projects¹, and
- assess new developments of pattern catalogs and languages.

We will cover the following material

- Software architecture
- Origins of the pattern movement
- Pattern-oriented software architecture: Architectural patterns, Design patterns, Idioms
- Application-specific pattern systems including many of the famous Gang of Four Patterns (GoF) [GHJV95]
- Patterns for software testing
- Pattern languages
- Critique and alternatives, e.g. frameworks

0.1.2 Precondition and Credit

There are no formal preconditions for module participation (except that you must be subscribed into the HIS master program), but good knowledge in principles and procedures of software engineering, some programming skills in object-oriented programming languages (C++ or Java will do) is very helpful.

You get 5 CP for successful participation. Module examination is an oral examination of at least 15 and maximum 45 minutes duration at the end of the semester.

¹part of software archeology

0.2 Educational Concept

0.2.1 Educational Objectives

Based on the module handbook, we identify the following educational objectives:

1. The student is able to understand the motives of the pattern community,
2. The student recognizes and understands a basic set of important foundational patterns,
3. The student distinguishes between different types of patterns,
4. The student is able to apply patterns in the design of software systems including SCS,
5. The student discover patterns in existing software projects², and
6. The student assess new developments of pattern catalogs and languages.

0.2.2 Learning Targets

At the end of this course, students shall be able to provide answers to the following questions:

- What is a pattern?
- What is a pattern language?
- What are basic patterns?
- What are basic pattern types?
- Why are patterns useful?
- How to chose the right pattern?
- When to invent or not invent a pattern?
- How to combine patterns?
- How to discover patterns?
- When are patterns inappropriate?
- How to diagnose pattern overdose?

²part of software archeology

- What is the relationship between patterns and (agile) methodologies?
- What are alternatives to patterns?
- ...

0.2.3 Learning Methodology

As we know from experience, that teaching and learning patterns is difficult, henceforth a more active approach is necessary!

To experience something has a far more profound effect on your ability to remember and influence you than if you simply read it in a book. (*Neil deGrasse Tyson*)

Therefore will use a methodology different from the usual lecture plus exercise scheme:

- At the end of each learning unit (lecture), a brief summary is given by the lecturer.
- A group of two students volunteers³ to make write an appropriate entry in the wiki about the subject covered. In addition to recapping the material presented, student have to think about consequences of applying the pattern (including negative ones)⁴
- This wiki entry is presented by the two students at the beginning of the next lecture (5-10 mins max) – for content and structure of wiki-entry, see section 0.2.5 below.
- A short plenum discussion wraps up this part.
- In the exercises we will have a combination of the following:
 - computer based exercises (read programming!)
 - computer assisted reading exercises (you will read and analyze open source code)
 - non programming tasks such as completing the wiki
 - excursion (no kidding)
 - case studies
 - surprises...

³or is drawn randomly – we will decide ion the first lecture!

⁴there will be support in the exercises and you find a lot about this in the literature, check-out "Consequences" e.g.

The preparation of short talks is an *ideal* preparation for the oral exams at the end of the semester! Also, form study groups as described here: <http://www.industriallogic.com/papers/learning.html>

Furthermore, you can influence the topics and the depth we cover topics!



Survey ?

0.2.4 A Word of Caution – Programming Skills are a Must!

Recall from the educational objectives, that

- ...
- The student is able to apply patterns in the design of software systems including SCS,
- The student discover patterns in existing software projects⁵, and
- The student assess new developments of pattern catalogs and languages.
- ...

Any of the above objectives (and many others) are *impossible* to achieve without a decent command of at least an OO-language such as Java. Thus, if you are *not* able to program in Java, C#, C++ or equivalent, please acquire these skills first and leave⁶ the course!

0.2.5 Wiki Content and Structure

The wiki entry should cover the following topics in general:

1. Brief wrap up of the subjects of the previous unit including motivation, definitions and consequences!
2. Relationship to other units (previously covered or outlook) – this is your creative part!
3. Critical assessment – your chance to provide feedback !

⁵part of software archeology

⁶Seriously, experience from the past has shown, that *every single* student lacking these skills fails in the oral exams.

4. Any open (not understood) parts – remember there are no stupid questions, only stupid answers!

In addition, if patterns are covered, the wiki entry should cover:

1. A full coverage of the pattern using a pattern template (we will chose one in the first lecture or exercise)!
2. An analysis of the pattern’s consequences including “liabilities” (i.e. the “bad” parts) – you are encouraged to consult the literature for this!
3. Any example for the pattern you have come across!

0.3 Structure

0.3.1 Learning Units

The learning units have the following structure (subject to change) depicted in table 1:

Table 1: Learning Units

No	Title	Description	Edu. Obj.
1	Introduction	Introduction into subject, one motivating example: Strategy Pattern	2
2	Behavioral Patterns	Revisit Strategy Pattern, Template and Command Pattern	2
3	Structural Patterns	Composite (+ Iterator)	2
4	Case Study: JUnit	JUnit as an example for dense patterns	3
5	Pattern Theory	Origin (C. Alexander's seminal work [AIS77]), Pattern Movement, GoF, Pattern Languages	1
6	OO-Principles I	Design Principles such as SRP, OCP, DRY, LSP, DIP, ISP based on [MM06]	3,4
7	OO-Principles II	Packaging software and dependency management based on [MM06]	3,4
8	Creational Patterns	(Abstract) Factory, Singleton	2,3
9	Patterns for Adding Flexibility	Decorator and Adapter	2,3
10	Decoupling Patterns	Observer and Chain of Responsibility	2,3
11	Case Study	MVC (classical and Web MVC2)	3,4
12	Architectural Patterns	Pipe and Filter Architectures, Blackboard	2,3,4
13	Analysis Patterns and Idioms	Analysis patterns based on [Fow97], programming idioms	2,3,4
14	Testing Patterns	Introduction into testing patterns for software testing	2,3,4
15	Criticism and Alternatives	Critics of the pattern movement	5

Graphic 1 depicts the design patterns covered (including exercises):

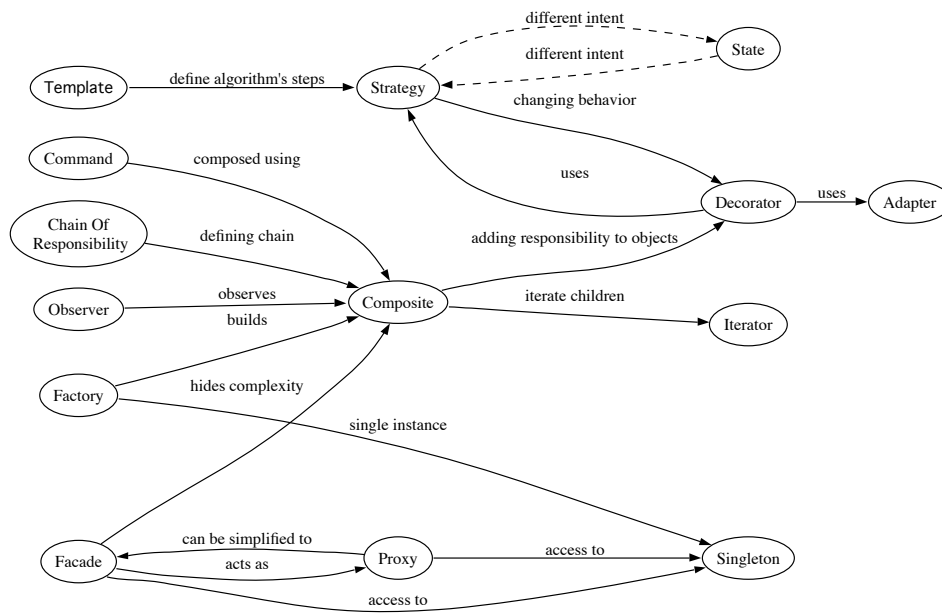


Figure 1: Design Pattern Overview

In addition, we will cover the following architectural patterns

1. Pipes and Filters,
2. Blackboard, and
3. MVC⁷

these analysis patterns

1. Observation,
2. Account, and
3. Transaction

and some programming *idioms*.

0.3.2 Exercises

The module is complemented with practical exercises, in which active participation is required. The students will prepare and demonstrate most of

⁷yes, some people regard it as an architecture pattern

the techniques described above during these exercises. Participation in the exercises is *mandatory*!

The learning units are accompanied with the following exercises depicted in table 2 below:

Table 2: Exercises

No	Title	Description
1	Strategy Pattern	Implement flying Ducks, and Sorting
2	Command Pattern	Find Template in <code>java.io</code> , implement a (macro) command, analyze the Futures pattern in <code>java.concurrent</code>
3	Composite Pattern	Implement a calculator for basic math expressions, add indentation level to menus (example code from [FFBS04])
4	JUnit	Modifying JUnit with patterns learned so far
5	Discovering Patterns	Excursion – let’s find some patterns in our neighborhood!
6	OO Principles I	Use Case Batch Payroll System to find the right abstractions, applying design patterns and OO-principles learned so far!
7	OO Principles II	Use Eclipse plugins such as Metrics and JDepend to calculate important metrics for a Toy Example and OO-Projects incl. Tomcat
8	Creational Patterns	Implement Singletons and analyze Java database connectivity
9	Decorator, Adapter	Analyze the <code>java.io</code> package and find Decorators, JUnit decorators
10	Observer, Chain of Resp.	Analyze the Servlet APIs and modify Servlet responses, initial look at JHotDraw
11	MVC	Reading and analyzing code: JHotDraw, Spring MVC, Observer Push / Pull
12	Architectural Patterns	Case Study: Apache Web Server
13	Analysis Patterns	Implement the Observation and Measurement Analysis Patterns (and related Patterns) in Java
14	Testing Patterns	TBD
15	Criticism and Alternatives	Summary Discussion

0.4 Literature

0.4.1 Books

This lecture is based mainly on the following books:

- Elisabeth Freeman, Eric Freeman, Bert Bates, and Kathy Sierra. *Head First Design Patterns*. O' Reilly & Associates, Inc., 2004 – don't get mislead by the cover, it is a good book with *mostly*⁸ good examples!
- Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003 – very good introduction into OO-principles, unparalleled coverage of deployment (packaging) challenges!
- Robert C. Martin and Micah Martin. *Agile Principles, Patterns, and Practices in C# (Robert C. Martin)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006 – based on the former, very good introduction into OO-principles, unparalleled coverage of deployment (packaging) challenges, examples all in C# – you get more bang for the buck!
- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995 – the “classic”!

However, as we go along important additional references including books and research articles will be given! If you are interested in patterns for concurrent systems, a very good one is the “Siemens” series [BMR⁺96], Vol. 1-5.

0.4.2 Useful Web References

The following references on the web are useful (but see also explicit References section at the end of this script):

- <http://www.hillside.net/patterns>
- <http://c2.com/cgi/wiki?DesignPatterns>
- <http://c2.com/ppr/>
- <http://www.cs.wustl.edu/~schmidt/patterns.html>
- <http://martinfowler.com/eaaCatalog/>

⁸some are bad, indeed...

0.4.3 Script

There does not exist a script for this lecture! These lecture notes are *my* own notes which enable *me* to structure the lectures and learning units! I.e. this script only serves as a supplement to the lectures and does *not* replace a textbook. The module encourages active participation and active learning, henceforth, this script should be merely regarded as an extended table of content and a reference to the literature!

Therefore, *you* should take your *own* notes!

A good explanation why this is always a good idea, may be found here:
http://www.fbi.h-da.de/fileadmin/personal/p.altenbernd/homepage_files/Mitschreiben.pdf

0.5 Lecture: Summary



Brief Summary: Active learning!

0.6 Exercise: Discover Patterns



Excursion to FH neighborhood!

Chapter 1

Unit 1 – Introduction

To rely on rustics and not prepare is the greatest of crimes; to be prepared beforehand for any contingency is the greatest of Virtues.

Sun Tzu

1.1 Lecture: Motivation



Take [FFBS04], chapter 1, p 2-24, develop on blackboard:

1. Explain setting SimUDuck
2. Basic UML Diagram
3. Problem: ducks should fly
4. Naive solution: put into base class and inherit
5. Problem: Rubber ducks can't fly, but do fly now!
6. Naive solution: Move down to subclasses!
7. Problem: Code duplication (violation of DRY-principle)
8. Break: *Change* is the problem here!
9. Solution: Move into separate interface and delegate!
10. Lesson 1: program to interfaces!
11. Lesson 2: prefer delegation over inheritance!

1.2 Exercise: Pair Work



Discuss in pairs whether you have previously come across something similar in your life as a programmer/student (5 mins)!

1.3 Lecture: Strategy Pattern



Provide brief overview of formal definition of strategy – if time permits, otherwise: homework!

1.3.1 Name

Strategy or Policy.

1.3.2 Intent

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it [GHJV95].

1.3.3 Problem

You want to be able to vary behavior (algorithms!) of objects – even at runtime! You must cope with anticipated and unanticipated changes to the algorithms and new algorithms and want to avoid changing code and/or breaking existing code (clients).

1.3.4 Solution

Use delegation instead of inheritance!

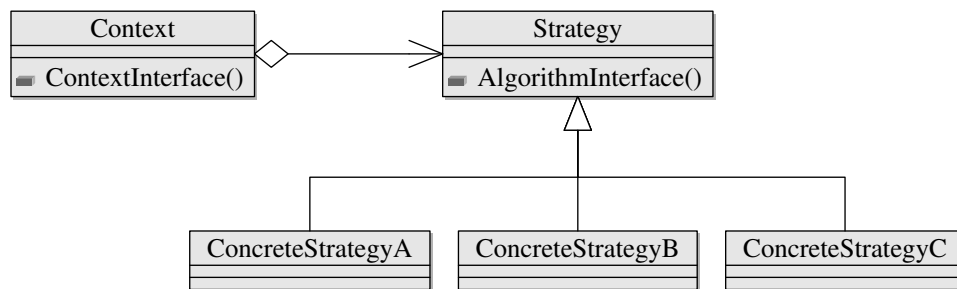


Figure 1.1: Strategy Pattern

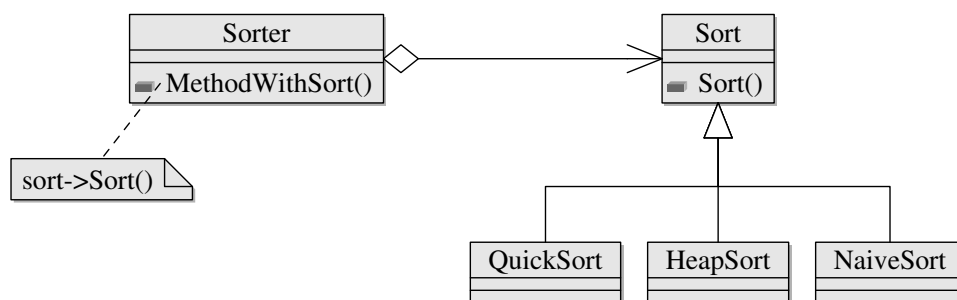




Figure 1.2: Application of Strategy Pattern

1.4 Exercise: What are the dependencies created?

 Working in pairs to determine the dependencies between context and strategy and vice versa!

1.4.1 Real World Example

 Can you think of any real world example ?

1.4.2 Additional Topics

- Related to Factory and Template Patterns (covered later)!
- Illustrates Open Closed Principle (OCP), also to be covered later!
- Illustrates Liskov Substitution Principle (LSP), also to be covered later!

1.4.3 Learning Goals

1. Recognize need for strategy pattern
 2. Recognize strategy pattern in existing code
 3. Understand *change* as the driving force
 4. Delegation as alternative to Inheritance
 5. Not only “things” but behavior can be encapsulated in classes
-

1.5 Lecture: Summary



Until we reach the purest sense of declarative programming, all programs will have patterns. These patterns are the recurring structures that programmers build within their chosen style and language to implement behaviors not directly supported by the language. The patterns literature describes what to build, in a given set of circumstances, along with some idea of how to build the what in a way that makes the most of the circumstances.

Eugene Wallingford



What does the chapter’s quote¹ mean?.....?

1.6 Exercise: Time to Code!



In the exercise sessions we will

1. try out code from [FFBS04]

¹epigraph

2. implement a flexible sorter for lists
-

1.7 Exercise: Questions for Self-Study



1. What happens when a system has an explosion of Strategy objects? Is there some way to better manage these strategies?
2. In the implementation section of the strategy pattern in [GHJV95], the authors describe two ways in which a strategy can get the information it needs to do its job. One way the strategy object could get passed a reference to the context object, thereby giving it access to context data. But is it possible that the data required by the strategy will not be available from the context's interface? How could you remedy this potential problem?

Chapter 2

Unit 2 – Behavioral Patterns

But once again, the difficulty of believing it may have to do with the fact that we tend to think of patterns as “things”, and keep forgetting that they are complex and potent fields.

Christopher Alexander

2.1 Lecture: Recap (Students)



Recap Strategy Pattern (5 mins)

2.2 Exercise: Plenum Discussion



Anything to remark or to add?

2.3 Lecture: Brewing Coffee and Tea



Take [FFBS04], chapter 8, p 276-298, develop on blackboard:

1. Motivation: coffee and tea recipes are similar!
 2. Code Smell: Duplication – violation of DRY-Principle! (We will investigate important OO-Principles in a later lecture and learn, that many patterns stem from these principles!)
 3. Refactor: Abstract out commonality (renaming is important!)
 4. Voilà: The *Template Pattern* emerges!
-

2.4 Exercise: Pair Work



Discuss in pairs whether you have previously come across something similar in your life as a programmer/student (5 mins)!

2.5 Lecture: Template Pattern



Provide brief overview of formal definition of template.

2.5.1 Name

Template

2.5.2 Intent

Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure [GHJV95].

2.5.3 Problem

Two different components have significant implementation (and behavior) similarities, but demonstrate no reuse of a common interface or implementation. Any change requires a duplication of efforts.

2.5.4 Solution

Abstract our commonality and use the Hollywood Principle (“don’t call us, we call you!”) as parent class calls subclass (and not the other way around).

Use it

- to implement the invariant parts of an algorithm once and leave it to the subclasses to implement behavior that can vary.
- to localize common behavior among subclasses and place it in a common class (in this case a superclass) to avoid code duplication – classic example of “code refactoring” (don’t apply it blindly without thinking, though).
- to control how subclasses extend superclass operations. Define a template method calling “hook” operations at specific points, there by permitting extensions at precisely that point – Open Closed Principle (OCP)

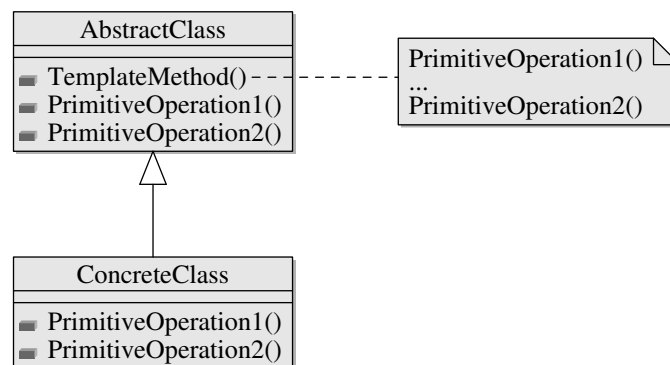


Figure 2.1: Template Pattern

2.5.5 Real World Example



Audience?

Examples include

2.6 Lecture: Template in the Wild



Take [FFBS04], chapter 8, p 300-305, the sorting example. Discuss the difference to the Template Pattern (no abstract class etc) in terms of technicalities, but the similarity with regards to intent and spirit. Also, discuss, why the Comparable has aspects of Strategy and Template Pattern (however, the sorting algorithm in `java.util.Arrays` is incomplete without a Comparable, therefore it does not fully qualify as a Strategy Pattern).

2.6.1 Additional Topics

- Strategy relies on delegation, Template on inheritance – which one to choose?
- Resilience to change?

2.6.2 Learning Goals

1. Recognize need for template pattern (train your instinct for *code smell*!)
 2. Recognize template pattern in existing code
 3. Understand relationship template and strategy
-

2.7 Lecture: Command Pattern



Take [FFBS04], chapter 6, p 191-232, develop on blackboard:

- Use Case, emphasize flexibility and unknown devices (slots)
- Diner example (comparison, p 206)
- Develop class diagram (p 207)

- Undo (p 220)
- Further sophistication: Macro (p 226)

Again: Behavior is encapsulated as an object!

2.7.1 Name

Command also known as Action.

2.7.2 Intent

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations [GHJV95].

2.7.3 Problem

Remember: Not everything is an object! Objects are not things! So, you want to model behavior as an object and wrap it into a class definition, you can chose the Command Pattern.

In Java Kingdom, the poor *verbs* (methods) are second-class citizens. (*Steve Yegge*)

2.7.4 Solution

Put the code into an **execute** method and send the instance of the class containing this method to the target as depicted in 2.2.

A (powerful) variant is the Macro Command Pattern as depicted in 2.3.

2.7.5 Real World Example



Audience ?

Examples include

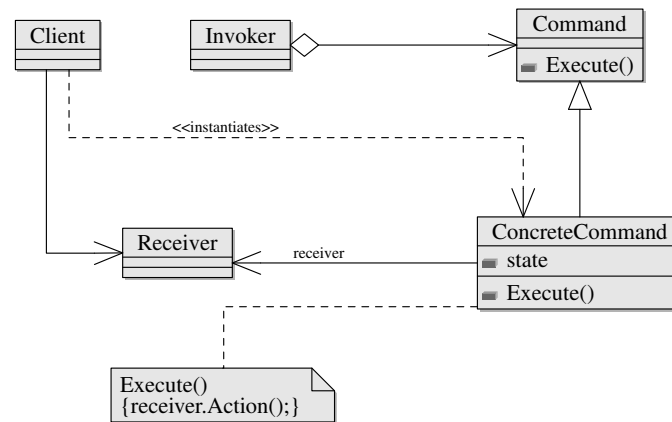


Figure 2.2: Command Pattern

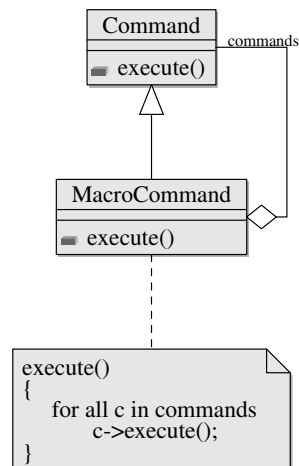


Figure 2.3: Macro Command Pattern

2.7.6 Additional Topics

- Commands are often used also in Web-Frameworks (Struts, Spring MVC)
- Stored-Procedures in the DB-World
- Commands are most important in multi-threaded¹ context, e.g. used in a worker-thread, thread-pool scenario!

¹or multi-process contexts, too

2.7.7 Learning Goals

1. Recognize need for Command Pattern
 2. Sharpen you sense for modeling the Non-Nouns (i.e. verbs!)
-

2.8 Lecture: Summary



Brief Summary: Wrap up, most important take-away: Sometimes one must escape form Noun-Land!



What does the chapter's quote² mean?.....?

2.9 Exercise: Time to Code!



In the exercise sessions we will

1. find Template Pattern usage in `java.io.InputStream`
 2. have a look at the Macro Command from [FFBS04]
 3. have a look at `java.io` package using Futures and the like (Commands with return values)
 4. implement Command Patterns for bank-accounts
-

²epigraph

2.10 Exercise: Questions for Self-Study



1. The Template Method relies on inheritance. Would it be possible to get the same functionality of a Template Method, using object composition? What would some of the tradeoffs be?
2. How does the Command Object get access to the data it needs to perform its job? How does one guarantee that this does not create a strong coupling?

Chapter 3

Unit 3 – Structural Patterns

Evidently, then, a larger part of the “structure” of a building or town consist of patterns of relationships.

Christopher Alexander

3.1 Lecture: Recap (Students)



Recap Behavioral Patterns (5 mins)

3.2 Exercise: Plenum Discussion



Anything to remark or to add?

3.3 Lecture: Iterator Pattern for Walking a Structure



Important Note: The Iterator Pattern is a Behavioral Pattern, but we need it to motivate the following!

Follow [FFBS04] chapter 9, p 315-334, develop on blackboard (and refactor `headfirst.iterator` to `java.util.iterator`):

1. Motivation: Client code should not care about *how* to iterate through a collection
2. Again: Model this *task* (iteration) as an object – the Iterator¹!

3.3.1 Name

Iterator also known as Cursor.



Databases Anybody ?

3.3.2 Intent

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation [GHJV95].

3.3.3 Problem

Also, if one wants to traverse in different ways.

3.3.4 Solution

Simple, as depicted in 3.1

¹Similar to previous chapter's idea to assign *behaviour* to an *object*

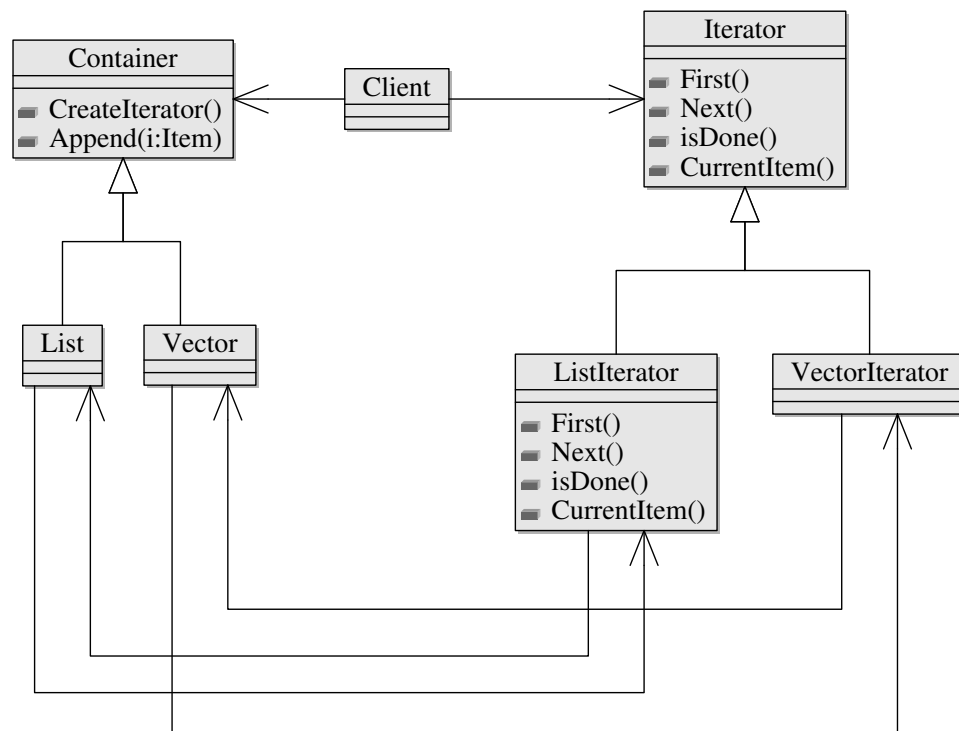


Figure 3.1: Iterator Pattern

3.3.5 Real World Example



Anybody ?

3.3.6 Additional Topics

- Iterators can iterate *any* structure – not only linear ones (see next section on Composite Pattern)!
- Iterators are only necessary in programming languages without proper functional constructs.

3.3.7 Learning Goals

1. Know Iterator as basic and common pattern (idiom)
2. Learn when to program an iterator itself

3. Know limitations
-

3.4 Lecture: Composite Pattern



Follow [FFBS04] chapter 9, p 353-377, develop on blackboard:

1. Recursive structure
2. Treat MenuItems and Menus alike

3.4.1 Name

Composite

3.4.2 Intent

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly [GHJV95].

3.4.3 Problem

From [GHJV95]:

Graphics applications like drawing editors and schematic capture systems let users build complex diagrams out of simple components. The user can group components to form larger components, which in turn can be grouped to form still larger components. A simple implementation could define classes for graphical primitives such as Text and Lines plus other classes that act as containers for these primitives. But there's a problem with this approach: Code that uses these classes must treat primitive and container objects differently, even if most of the time the user treats them identically. Having to distinguish these objects makes the application more complex. The Composite pattern describes how to use recursive composition so that clients don't have to make this distinction.

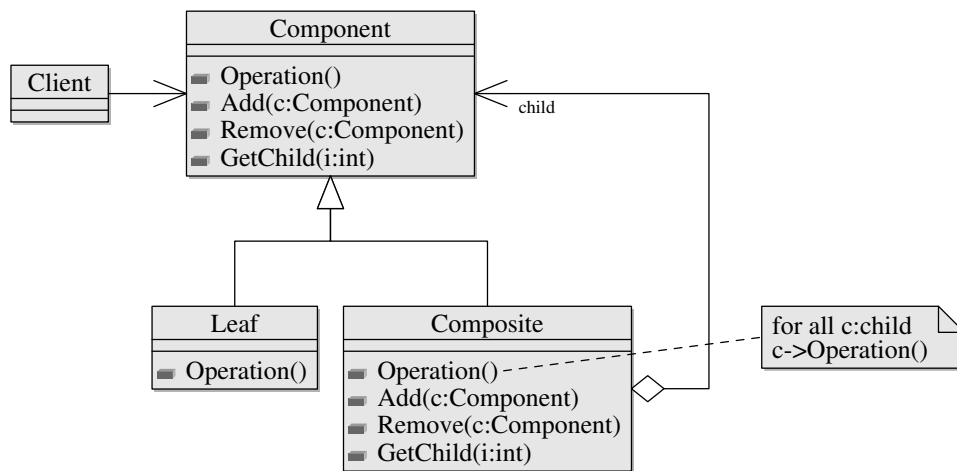


Figure 3.2: Composite Pattern

3.4.4 Solution

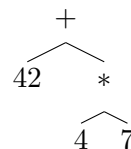
From [GHJV95]:

The key to the Composite pattern is an abstract class that represents both primitives and their containers. For the graphics system, this class is `Graphic`. `Graphic` declares operations like `Draw` that are specific to graphical objects. The subclasses `Line`, `Rectangle`, and `Text` (see preceding class diagram) define primitive graphical objects. These classes implement `Draw` to draw lines, rectangles, and text, respectively. Since primitive graphics have no child graphics, none of these subclasses implements child-related operations. The `Picture` class defines an aggregate of `Graphic` objects. `Picture` implements `Draw` to call `Draw` on its children, and it implements child-related operations accordingly. Because the `Picture` interface conforms to the `Graphic` interface, `Picture` objects can compose other `Pictures` recursively.

3.4.5 Real World Example

Examples include

- Arithmetics $42 + (4 * 7)$:



- Reporting lines (and reporting processes) in most companies
- GUIs

3.4.6 Additional Topics

- Explain why mixing iterator logic and printing in [FFBS04], p. 363 is bad style!
- Composite Iterator, [FFBS04] p 369, see also exercises! However, code is *buggy* and contains a flaw in the logic! Supply better code!
- Expose the problems of relying on `UnsupportedOperationException` – symptom of treating Composite and Component uniformly!

3.4.7 Learning Goals

1. Recognize Composite Pattern as an important structural pattern
2. Know how to traverse Composites (using e.g. an Iterator)



What does the chapter's quote² mean?.....?

3.5 Exercise: Time to Code!



In the exercise sessions we will

1. use the Composite Pattern to implement a calculator capable of evaluating arithmetic expressions
2. add indentation levels to menus (example code from [FFBS04])

3.6 Exercise: Questions for Self-Study



²epigraph

1. How does the Composite pattern help to consolidate system-wide conditional logic?
2. Would you use the composite pattern if you did not have a part-whole hierarchy? In other words, if only a few objects have children and almost everything else in your collection is a leaf (a leaf can have no children), would you still use the composite pattern to model these objects?

Chapter 4

Unit 4 – Case Study JUnit

Each pattern helps to sustain other patterns. The individual configuration of any one pattern requires other patterns to keep itself alive [...] many patterns must cooperate. [...] Now we begin to see what happens when the patterns in the world collaborate.

Christopher Alexander

The whole is more than the sum of the parts.

Metaphysica Aristotle

4.1 Lecture: Recap (Students)



Recap Structural Patterns (5 mins)

4.2 Exercise: Plenum Discussion



Anything to remark or to add?

4.3 Lecture: A Cook's Tour



Lecture along <http://junit.sourceforge.net/doc/cookstour/cookstour.htm> with maybe live coding!

- Introduction JUnit Framework
- Goals
- Design¹

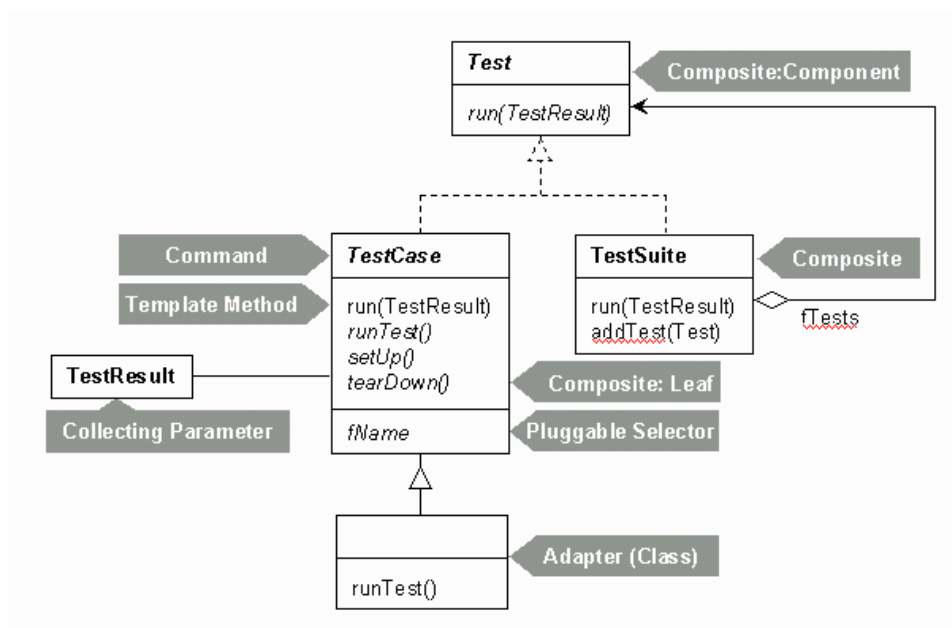


Figure 4.1: JUnit Patterns

4.4 Exercise: Time To Code!



In the exercises we will

1. extend JUnit with Conditional Test Cases,
2. extend JUnit with Performance Measurement,
3. extend JUnit with Performance Testing, and

¹we refer to version 3.8.1. and *not* the latest for pedagogical reasons!

4. explore the TestSuite (Composite Pattern!)



What does the chapter's quote² mean?.....?

4.5 Exercise: Questions for Self-Study



1. Draw – out of the top of your head – the UML diagram of the core JUnit classes!
2. Name all patterns used!
3. Explain the rationale for the patterns in your own words!
4. From looking at the pattern and UML diagram, where are the extensions points that are used when using JUnit in a regular way?
5. From looking at the pattern and UML diagram, where are the extensions points that could be used to extend JUnit with additional functionality?

²epigraph

Chapter 5

Unit 5 – Pattern Theory

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.

Christopher Alexander

Each pattern is a three part rule, which expresses a relation between a certain context, a problem, and a solution.

Christopher Alexander

A pattern is a careful description of a perennial solution to a recurring problem within a building context, describing one of the configurations which brings life to a building.

Christopher Alexander

And it is the pattern language which, like genes distributed throughout the cells, make certain that there is this structure, this invariant permanency, in the flux of things, so that the building or the town stays whole.

Christopher Alexander

5.1 Lecture: Recap JUnit (Students)



Recap (10 mins)

5.2 Exercise: Plenum Discussion



Anything to remark or to add?

5.3 Lecture: Pattern Origins



Patterns have their origin in building architecture and go back to the seminal work of Christopher Alexander ([Ale79] and [AIS77]). Patterns codify existing knowledge about solving problems. An expert knows many of these patterns either consciously or sub-consciously. Therefore, instead of re-inventing the wheel over and over again and always starting from scratch, she can reuse and combine those existing patterns in order to assemble a solution.

5.3.1 Origin

Throughout the 1960s and 1970s the architect Christopher Alexander and his colleagues identified the concept of patterns for capturing architectural decisions and arrangements found in real, existing architecture.

Ironically, the pattern language idea so far had limited impact in the building industry, but has had a *profound* influence in the information technology industry. Among the many followers we only cite the “Gang of Four” [GHJV95] and Schmidt et al. [Sch95] and [SSRB00], resp.

From an architecture and design perspective, software is often thought of in terms of its parts: packages and modules, functions and procedures, objects and classes, source files, libraries, and so forth. In fact, architecture is even *defined* in terms interacting components. While these all represent

valid views these views focus on the parts, however, and de-emphasize the relationships and the reasoning that constitute an architecture. In contrast patterns emphasize

- the why,
- the how,
- the where,
- the pros and the cons

of a design and not just the “what”.

A pattern documents covers

- a recurring *problem*
- a *named solution* to this problem
- a *context* this problem and its solution are apply to
- a description of the (often contradicting) *forces* relevant
- a detailed description of the solution including
- its benefits, drawbacks and liabilities¹
- empirical support for the solution²

By documenting existing, well-proven design experience and providing a common vocabulary and understanding, henceforth, patterns enable the distilling, communicating, and sharing of architectural knowledge. They help to build heterogeneous and complex software.

5.3.2 Pattern Examples

Alexander describes his patterns in natural language using a consistent format which contains the patterns name, the given context, the forces that act and the solution. Below please find an example of a pattern, see also figure 5.1:

Window Place 1:

“Everybody loves window seats, bay windows, and big windows with low sills and comfortable chairs drawn up to them. . . A room which does not have a place like this seldom allows you to feel comfortable or perfectly at ease. . .

¹Remember: engineering is about choosing the right compromise!

²Remember: patterns are about collecting known and proven solutions!

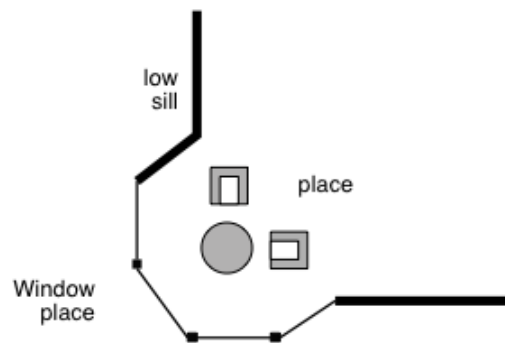


Figure 5.1: Pattern Window Place (Source: Christopher Alexander)

If the room contains no window which is a “place”, a person in the room will be torn between two forces:

He wants to sit down and be comfortable. He is drawn toward the light.

Obviously, if the comfortable places – those places in the room where you most want to sit – are away from the windows, there is no way of overcoming this conflict...

Therefore: In every room where you spend any length of time during the day, make at least one window into a ‘window place’.

5.4 Exercise: Pair Work



Think of any other example of a pattern in the “real” world of building and/or construction architecture! Time: 10 mins, will be (re-) used in the follow-up exercise (excursion)!

5.5 Lecture: Software Patterns



The following example is taken from [BHS07]. Imagine your refrigerator is empty and you need to go to the grocery store. You need milk, juice,

coffee, pizza, fruit, and many of the other major food types. You go to the store, fetch a bottle of milk and return to your apartment. You put the milk into the refrigerator and return to the shop to buy the next item. This sound pretty inefficient, does not it? Yet, one can observe this pattern in many distributed applications in particular in those ones, that try to hide the “distributedness” and the network from the developer (by using stubs and skeletons for instance) fooling the developer to apply his pattern of making many local “get” calls suitable for a non-distributed application only as depicted in 5.2.

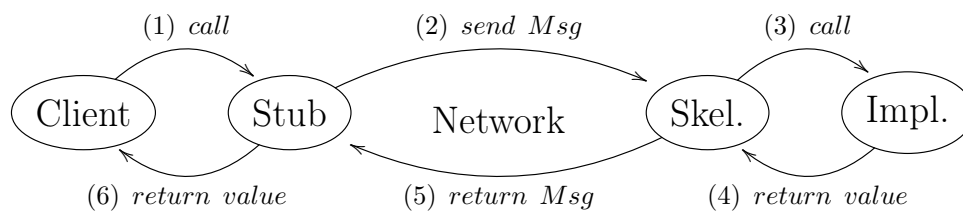


Figure 5.2: Stubs and Skeletons

In order to solve this problem more efficiently, one can make use of the so-called BATCH METHOD pattern³:

Name: BATCH METHOD

Description: Group multiple collection accesses together to reduce the cost of multiple individual accesses.

Context: Distributed systems

Problem: How can many actions be treated as a single transaction? In particular, how can multiple operations on a collection be handled efficiently and consistently? In a distributed system— or other environments involving access latency, such as database access, naively iterated operations use up bandwidth:

Example Code

```

for(each key in Keys)
{
    ...
    dictionary.put(key, value);
}
  
```

```

for(each key in Keys)
  
```

³or Shopping List Pattern

```
{  
    value = dictionary.get(key);  
    ...  
}
```

as every call to `get(key)` for instance goes over the wire (network).

Solution: Define a single method that performs the entire action altogether. The method is declared to take all the arguments for each execution of the action and to return results grouped together. This single method folds the repetition into a data structure rather than a loop, that is executed “close” to the data (co-located). Therefore, the cost of access is reduced to a single access:

Example Code

```
interface RemoteDictionary  
{  
    ...  
    Object[] get(Object[] keys);  
    void put(Object[] keys, Object[] values);  
    ...  
}
```

Benefits: The latency and bandwidth consumption is reduced greatly.

Liability: Significantly more housekeeping is performed to set up the call and the programming API differs from the local one. Intermediate data structures are required.

Known Uses: The BATCH METHOD pattern is found in many distributed system architectures, such as the CORBA Common Object Services.

Please note, that the pattern is documented in a structured language format.

5.5.1 Pattern Language

Unfortunately, there is no standard on terminology. Several authors have suggested similar, but different formats. We follow the one documented in the seminal work of Fowler et al. [GHJV95] as depicted in table 5.1.

5.5.2 Pattern Categories

There are different pattern categories. It is important for the software engineer to acquire proficiency in all of them and to distinguish between them. Table 5.2 lists the three different categories.

Often design patterns are further grouped into the following categories:

Section	Description
Pattern Name and Classification	A descriptive and unique name that helps in identifying and referring to the pattern.
Intent	A description of the goal behind the pattern and the reason for using it.
Also Known As	Other names for the pattern.
Motivation (Forces)	A scenario consisting of a problem and a context in which this pattern can be used.
Applicability	Situations in which this pattern is usable; the context for the pattern.
Structure	A graphical representation of the pattern. Class diagrams and Interaction diagrams may be used for this purpose.
Participants	A listing of the classes and objects used in the pattern and their roles in the design.
Collaboration	A description of how classes and objects used in the pattern interact with each other.
Consequences	A description of the results, side effects, and trade offs caused by using the pattern.
Implementation	A description of an implementation of the pattern; the solution part of the pattern.
Sample Code	An illustration of how the pattern can be used in a programming language
Known Uses	Examples of real usages of the pattern.
Related Patterns	Other patterns that have some relationship with the pattern; discussion of the differences between the pattern and similar patterns.

Table 5.1: Pattern Notation

1. Creational patterns,
2. Structural patterns, and
3. Behavioral patterns.

Architectural Patterns	provide the fundamental structural organization schema for software systems.
Design Patterns	provide schemes for refining the sub-systems or components of software systems, or the relations between them. In addition, they provide “band-aids” for programming language deficiencies or missing constructs.
Idioms	provide low-level patterns specific to a programming language and part of the programming best-practice folklore.

Table 5.2: Pattern Category

5.5.3 Criticism

Despite of the huge momentum patterns have taken on over the last almost two decades, there has been some criticism, too.

Patterns are signs of weakness in programming languages.

Mark Dominus (<http://blog.plover.com/prog/design-patterns.html>)

This argument pretends, that patterns are only necessary, because the development language most developers have to use suffer from various deficiencies and are not expressive enough. This particularly applies to statically typed, imperative languages such as C++, as Peter Norvig points out, see <http://www.norvig.com/design-patterns/>. The claim is, that more expressive languages such as Python, Smalltalk, Lisp, etc. do not require patterns or to a lesser degree. For example, consider the command pattern, which is not necessary in languages supporting functions as first class citizens! We will cover these topics again in a later unit!

On the other hand, although these critics have a valid point in my opinion, patterns are not restricted to low-level design patterns for specific languages. Rather the pattern concept can be applied on a higher level abstraction, too. For instance, patterns are valid to describe enterprise architectures of distributed systems as done in [Fow02]. And it is highly unlikely that these higher-level abstractions will ever be fully incorporated into language concepts.

Several pitfalls, however one should be aware of before applying patterns:

- Patterns should not be applied mechanically.
- A simpler solution is usually preferable over one with many patterns.

- Inexperienced developers usually do not understand and henceforth *underutilize* patterns.
- Experienced developers usually understand and henceforth *overutilize* patterns.
- It takes craftsmanship and mastery to apply patterns correctly.

Following the pattern movement, so-called *antipatterns* have emerged as well, describing particularly bad, but common “solutions”, see [Ris98]. One popular (anti-) pattern is the big ball of mud⁴ as depicted in figure 5.3.

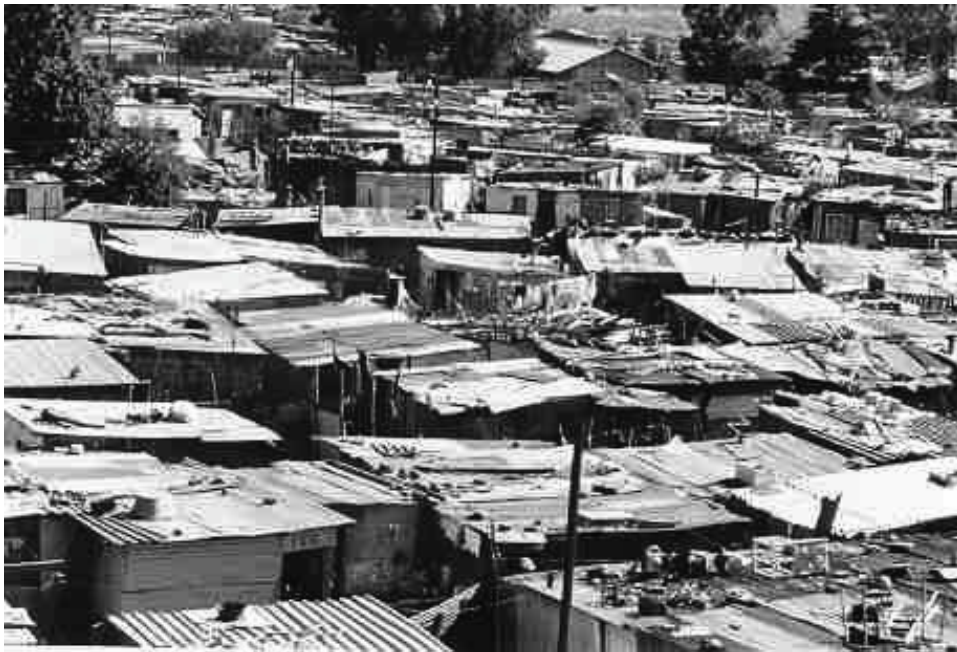


Figure 5.3: The Big Ball of Mud

“A BIG BALL OF MUD is haphazardly structured, sprawling, sloppy, duct-tape and bailing wire, spaghetti code jungle. We’ve all seen them. These systems show unmistakable signs of unregulated growth, and repeated, expedient repair. Information is shared promiscuously among distant elements of the system, often to the point where nearly all the important information becomes global or duplicated. The overall structure of the system may never have been well defined. If it was, it may have eroded beyond recognition. Programmers with a shred of architectural sensibility shun these quagmires. Only those who are unconcerned about architecture, and, perhaps, are comfortable

⁴<http://www.laputan.org/mud/>

with the inertia of the day-to-day chore of patching the holes in these failing dikes, are content to work on such systems.”

Brian Foote and Joseph Yodde

5.6 Exercise: Pair Work



Think of other shortcomings, potential problems and issues when applying patterns! (5 mins)

5.6.1 Learning Goals

1. Understand the origins of the pattern movement
 2. Know the “founders”
 3. Understand relationship to “abstract”, generic pattern theory
 4. Understand the structure of pattern descriptions
-

5.7 Exercise: Excursion!



In the exercise sessions we will go outside and discover patterns in *real* architecture (no kidding)!

5.8 Lecture: Summary



Brief Summary:

1. Patterns have their origin in building architecture.
2. Patterns are proven solutions to recurring problems.

3. Multiple patterns form a Pattern Language.
4. Patterns have been criticized.



What does the chapter's quote⁵ mean?.....?

5.9 Exercise: Questions for Self-Study



1. Explain the origin of the Pattern Language movement!
2. Define a pattern in your own words!
3. Is a pattern a recipe?
4. Name and explain a few building patterns!
5. Try to write down the (Non-Alexandrian⁶) Pattern “Skyscraper”. Pay attention to Context, Problem and Solution and explain the Forces!
6. Can you implement a pattern library that you could reuse? If not, why not?

⁵epigraph

⁶Christopher Alexander instead has a “four-storey” limiting pattern as he is of the opinion that high buildings are generally unhealthy...

Chapter 6

Unit 6 – OO-Principles I

The “bad” patterns are unable to contain the forces which occur in them. [...] In the end, the whole system must collapse.

Christopher Alexander

6.1 Lecture: Recap (Students)



Recap Pattern Theory (10 mins)

6.2 Exercise: Plenum Discussion



Anything to remark or to add?



Agile Experience Anybody.....?

6.3 Exercise: Design Smells



Working in pairs: Find “Design Smells”!

6.4 Lecture: Design Smells



Following [Mar03], p 88 list important design smells and contrast with findings from previous exercise:

1. Rigidity
2. Fragility
3. Immobility
4. Viscosity
5. Needless Complexity
6. Needless Repetition
7. Opacity

Reasons for software to degrade:

In many software development processes there is *no* planning for change!

Different from traditional methodologies, agile methodologies

- puts change at the center of the software development process!
- does not allow the software to rot!

Example for rotten software: The common Cut & Paste Reuse!



Any Examples.....?

6.5 Lecture: The Principles



Lecture based on [Mar03], p 95-149 on the important principles¹ for OO-Development:

6.5.1 SRP: The Single Responsibility Principle

Class should do one thing and do it well!

6.5.2 OCP: The Open Closed Principle

Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification. [Mar03] This is based on ideas from Bertrand Meyer [Mey88].

6.5.3 LSP: The Liskov Substitution Principle

The Liskov principle is a short name for the Liskov Substitution Principle by Barbara Liskov:

Let $q(x)$ be a property provable about objects x of type T . Then $q(y)$ should be true for objects y of type S where S is a subtype of T .

In ordinary words: objects of subtypes can be used whenever the supertype is required. Thus, an object of a superclass should always be substitutable by an object of a subclass:

- Same or weaker preconditions
- Same or stronger postconditions
- Derived methods should not assume more or deliver less

A much quoted violation is a square class that derives from a rectangle class, assuming setter methods exist for both width and height:

A square is for sure a kind of rectangle. However, applying the method

`setWidth(20)`

which is valid for rectangles to the square



¹sometimes called Uncle Bob's Principles

makes it no longer a square



[W]e take the minimalist approach to inheritance. We use it only when it makes our components more efficient, or when it solves certain problems in the type system. We do not intend for our components to serve as a collection of base classes that users extend via derivation. It is exceedingly difficult to make a class extensible in abstracto (it is tenfold harder when one is trying to provide classes that are as efficient as possible). Contrary to a common misconception, it is rarely possible for a programmer to take an arbitrary class and derive a new, useful, and correct subtype from it, unless that subtype is of a very specific kind anticipated by the designer of the base class. Classes can only be made extensible in certain directions, where each of these directions is consciously chosen (and programmed in) by the designer of the class. Class libraries which claim to be “fully extensible” are making an extravagant claim which frequently does not hold up in practice... There is absolutely no reason to sacrifice efficiency for an elusive kind of “extensibility”.

Martin and Isner [CI92]. For a good discussion, see also [Kis03] and <http://okmij.org/ftp/Computation/Subtyping/>.

6.5.4 DIP: The Dependency-Inversion Principle

Dependency principle – Classes should only depend on abstractions and High-Level Modules should not depend on low-level modules. This is the basis for popular IoC principles, too.

6.5.5 ISP: The Interface-Segregation Principle

Clients should not be forced to depend on methods that they do not use.



What does the chapter’s quote² mean?.....?

²epigraph

6.6 Exercise: Time To Code!



Use Case: Batch Payroll System!

1. We will find the right abstractions looking at the specification and use cases!
 2. We will apply design patterns and OO-principles learned so far!
 3. We will analyze and criticize our design!
-

6.7 Exercise: Questions for Self-Study



1. Why does SRP make sense?
2. Can you describe the Open Closed Principle in your own words?
3. Does the following sentence: “this class adheres to the Open Closed Principle” make sense?
4. Can you check for conformance to LSP automatically?
5. Why does DIP make sense?
6. Why does ISP make sense?

Chapter 7

Unit 7 – OO-Principles II

When one pattern is alive, it resolves its own forces, it is self-sustaining, self-creating and its internal forces continuously support themselves.

Christopher Alexander

7.1 Lecture: Recap (Students)



Recap OO-Principles I (10 mins)

7.2 Exercise: Plenum Discussion



Anything to remark or to add?

7.3 Lecture: Packaging Patterns



Following [Mar03]¹, p 253-268.

7.3.1 Modularity

The basic idea behind modularity is to partition the system such that parts can be designed and revised independently (another application of the “divide and conquer” approach).

“Modularization consists of dividing a program into modules which can be compiled separately, but which have connections with other modules.”

Barbara Liskov

“The connections between modules are the assumptions which the modules make about each other.”

David Lorge Parnas

Thus we define *Modularity* referring to dividing a program into smaller units, which have connections among each other making explicit assumptions.

The following holds true for a modular design:

- System is partitioned into modules that each fulfill a specific task.
- Modules should be changeable and reusable independent of other modules.
- Modules are well-defined, documented boundaries within a program and help comprehension of the program.
- Modules serve as the physical containers declaring artifacts such as classes or functions.
- In C and C++, modules are separately compiled files and header files contain module interfaces.
- Modules should group logically related abstractions (i.e. should be *cohesive*, see below) and minimize dependencies among modules (i.e. should be *loosely coupled*, see below).
- Henceforth, Modularity is the property shared by every system that has been decomposed into a set of cohesive and loosely coupled modules.

Note that

¹Martin’s work is exceptional, as virtually no other author applies patterns to the packaging problem, but in reality in the field, this is a quite important *concern*!

1. Modules should hide information and expose as little inside information as possible.
2. Modules should cooperate and therefore have to exchange information.

These goals are conflicting with each other!

High cohesion modules should contain functions that logically belong together.

Weakly coupled modules are modules, where changes to one module do not affect other modules.

The *Law of Demeter* (or Principle of Least Knowledge), named after the greek goddess of Demeter, says that one should only talk to friends or (more formally) the Law of Demeter for functions requires that a method M of an object O may only invoke the methods of the following objects:

1. O itself
2. M 's parameters
3. any objects created/instantiated within M
4. O 's direct component objects
5. a global variable, accessible by O , in the scope of M

The Law of Demeter is also (sloppily) referred to as the “One Dot Rule”, i.e. *never* write code like this:

```
address = order.item[item_id].manufacturer.address
```

Rather write something like this (“Demeter transmogrifiers”, see below):

```
address = order.getManufacturerAdressFor(item_id)
```

or directly as a service:

```
address = shop.getAddressFor(item_id)
```

Adhering to the Law of Demeter has the following effects:

- Reduces coupling between modules
- Disallows direct access to parts
- Therefore reduces dependencies
- Limits the number of accessible classes
- Results in several new wrapper methods (“Demeter transmogrifiers”, the term originated from a Calvin and Hobbes cartoon)

7.3.2 Principles of Package Cohesion

The following principles have been found to be useful:

- The Reuse-Release Equivalence Principle (REP)
- Common-Reuse Principle (CRP)
- Common-Closure Principle (CCP)
- The Acyclic-Dependencies Principle (ADP)
- The Stable-Dependencies Principle (SDP)

7.3.3 The Reuse-Release Equivalence Principle (REP)

The granule of *reuse* is the granule of *release*.

Consequently, The granule of *reuse* is the *package*.



What does this mean.....?

7.3.4 Common-Reuse Principle (CRP)

The classes in a package are reused together. If you reuse one of the classes in a package, you reuse them all.



What does this mean.....?

7.3.5 Common-Closure Principle (CCP)

Classes in a package should be closed to the same kinds of changes. A change that affects a package affects all classes in a package.



What does this mean.....?

7.3.6 The Acyclic-Dependencies Principle (ADP)

Allow no cycles in the package-dependency graph.



What does this mean.....?



Why does this make sense.....?

7.3.7 The Stable-Dependencies Principle (SDP)

Depend in the direction of stability.



Benefit.....?

7.3.8 Metrics

Metrics:

The Instability I is defined as

$$I = \frac{C_e}{C_a + C_e},$$

where C_a denotes the Afferent Couplings, i.e. the number of classes outside this package that depend on classes within this package, and C_e denotes the Efferent Couplings, i.e. the number of classes inside this package that depend on classes outside this package.

The Abstractness A is defined as

$$A = \frac{N_a}{N_c},$$

where N_a denotes the number of abstract classes and N_c denotes the total number of classes in this package.

Further principles:

- The Stable-Abstractions Principle (SAP)
- Try to have as many packages in the main sequence (see 7.1)

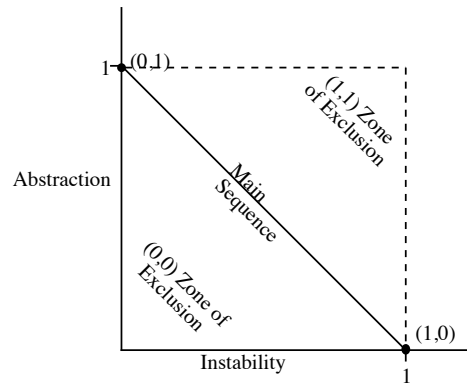


Figure 7.1: The Main Sequence



What does the chapter's quote² mean?.....?

7.4 Exercise: Time To Analyze!



We will use Eclipse plugins such as Metrics and JDepend in order to calculate the metrics introduced above:

1. Toy Example
2. JUnit
3. Tomcat 7 source code
4. JHotDraw source code



What does the chapter's quote³ mean?.....?

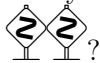
²epigraph

³epigraph

7.5 Exercise: Questions for Self-Study



1. Can you answer all questions in sections in this chapter indicated with



2. Why is it good to be on the main sequence in 7.1?
3. Why is it bad to be at $(0, 0)$ or $(1, 1)$?

Chapter 8

Unit 8 – Creational Patterns

Each pattern is a rule which describes what you have to do to generate the entity which it defines.

Christopher Alexander

8.1 Lecture: Recap (Students)



Recap OO-Principles II (10 mins)

8.2 Exercise: Plenum Discussion



Anything to remark or to add?

8.3 Lecture: Abstract Factory



Follow [Mar03] chapter 21, p 269-274, develop on blackboard:

1. Motivation: Client code should not care about *how* to iterate through a collection
2. Solution Factory (Method)
3. Attention: Dependency Cycle! Breaking reduces type safety; this is common technique in particular in IoC Frameworks, where configuration is driven by XML interpreted at run-time (`Class.forName(name)`)!
4. Substitutable Factories – these are Abstract Factories
5. Examples: Database (access) – we will study these in the exercises looking at some Java APIs

8.3.1 Name

Factory

8.3.2 Intent

Provide an interface for creating families of related or dependent objects without specifying their concrete classes [GHJV95].

8.3.3 Problem

Separate object creation from client code. Client should be configurable with multiple families of products (possibly even at run-time).

8.3.4 Solution

Create an interface or abstract class, concrete subclasses serve as factories for concrete products. This is mirrored by an abstract product hierarchy.

8.3.5 Real World Example



Anybody ?

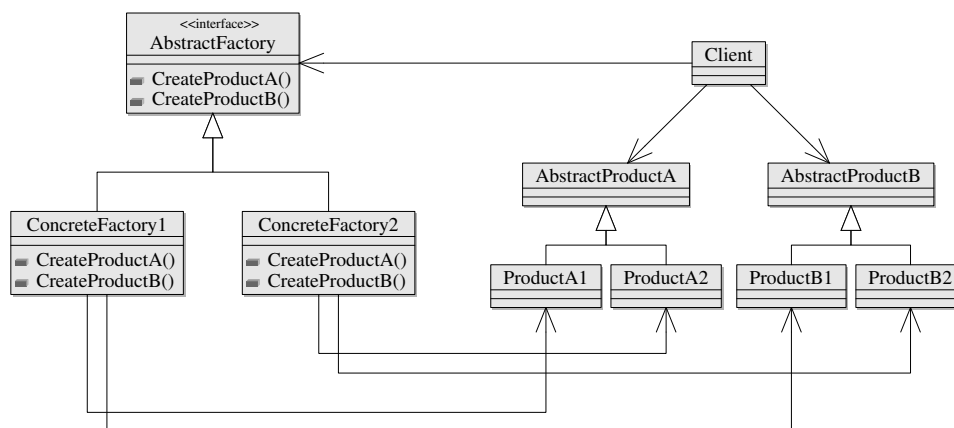


Figure 8.1: Abstract Factory Pattern

8.3.6 Additional Topics

Abstract Factory

- uses the Factory Method Pattern (not regarded by everybody as a Pattern but some treat it as an Idiom), i.e. clients trigger product creation via an interface *not* a concrete class
- is often implemented using the Singleton Pattern (see below) as only one instance of the factory is needed
- is not necessary (or invisible) in languages where classes are proper (first class) objects themselves (and one can change the behavior of `new`)

8.4 Exercise: Pizza Factories



Have a look at the example code from Head First for the Abstract Factory Pattern. Does it make sense to have 35 classes for 35 Pizzas?

8.5 Lecture: Singleton



Follow [FFBS04] chapter 5, p 170-189, develop on blackboard:

1. Motivation: Need single objects, do not want to use global variables
2. Run Dialogue p 171-172
3. Lazy Instantiation (General Idiom)
4. Advantage: One can subclass Singleton (though dangerous) if design is properly extended (e.g. using registries), compare [GHJV95] p 131. For an alternative approach, see <http://radio-weblogs.com/0122027/stories/2003/10/20/implementingTheSingletonPatternInJava.html>. However: ask yourself, why you want to subclass in the first place?!

8.5.1 Name

Singleton

8.5.2 Intent

Ensure a class has only one instance, and provide a global point of access to it [GHJV95].

8.5.3 Problem

Some objects should exist only once, for example

- Thread-Pools
- Database Connection Pools
- Printer spooler
- etc.

8.5.4 Solution

Make constructor private as depicted in figure 8.2!

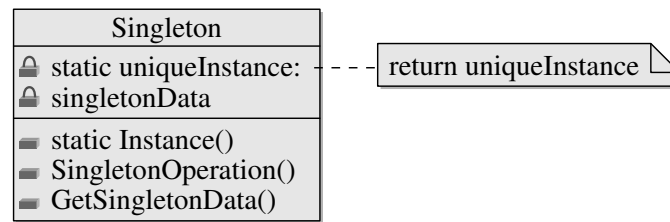


Figure 8.2: Singleton

8.5.5 Real World Example



Anybody ?

8.5.6 Additional Topics

There are multiple problems with Singleton

- Rookies tend to overuse – remember Singletons are the OO-analogon of global variable and should be used sparingly!
- Multiple Singletons in two or more Virtual Machines
- Multiple Singletons simultaneously loaded by different class loaders (relevant for e.g. Servlets)
- Double Check locking does not work pre-Java 1.5
- Serialization

8.5.7 Learning Goals

1. Recognize (Abstract) Factory Pattern as an important creational pattern
2. Understand the Singleton Pattern as an important creational pattern
3. Understand the Singleton Pattern's limitations
4. Ability to write custom (Abstract) Factories and Singletons



What does the chapter's quote¹ mean?.....?

¹epigraph

8.6 Exercise: Time to Code!



In the exercise sessions we will

1. Create a Singleton subclass of JUnit's `TestCase` to check uniqueness
 2. Analyze Java's JDBC database access patterns
-

8.7 Exercise: Questions for Self-Study



1. In the implementation section of this pattern in [GHJV95], the authors discuss the idea of defining extensible factories. An Abstract Factory is composed of Factory Methods, each Factory Method has exactly one signature. Does this imply that the Factory Method can only create an object in one way?
2. Consider the MazeFactory example in [GHJV95]. The MazeFactory contains a method called `MakeRoom` taking an integer as a parameter, which represents a room number. What do you have to do if you wanted to specify the room's color & size as well? Do you would to create a new Factory Method for your MazeFactory, allowing you to pass in room number, color and size to a second `MakeRoom` method?

Chapter 9

Unit 9 – Patterns for Adding Flexibility

Decorator changes the skin, while Strategy changes the guts.

Erich Gamma, Richard Helm, Ralph Johnson, and
John Vlissides

9.1 Lecture: Recap (Students)



Recap Creational Patterns (10 mins)

9.2 Exercise: Plenum Discussion



Anything to remark or to add?

9.3 Lecture: Decorator



Decorator is an elegant pattern. Btw – we do *not* follow the exposition and example in [FFBS04] as it is a *bad* example.

9.3.1 Name

Decorator also known as Wrapper.

9.3.2 Intent

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality [GHJV95].

9.3.3 Problem

We want to add behavior (responsibilities) to individual objects and *not* the class – we “enhance” (decorate) the object. We want to do this in a flexible manner, maybe even at run-time.

9.3.4 Solution

1. Subclass class of object to be enhanced.
2. Enclose the object to be enhanced.
3. Forward all method calls to object.
4. Modify, add what ever is required (before/after idiom)

Both methods outlined above adhere to the same construction idiom: wrapping objects recursively – a hallmark of the Decorator design pattern. Decorators represent a powerful alternative to inheritance. Whereas inheritance lets you add functionality to classes at compile time, decorators let you add functionality to objects at runtime.

Figure 9.1 illustrates the idea:

9.3.5 Code Example

Code example from `java.io`:

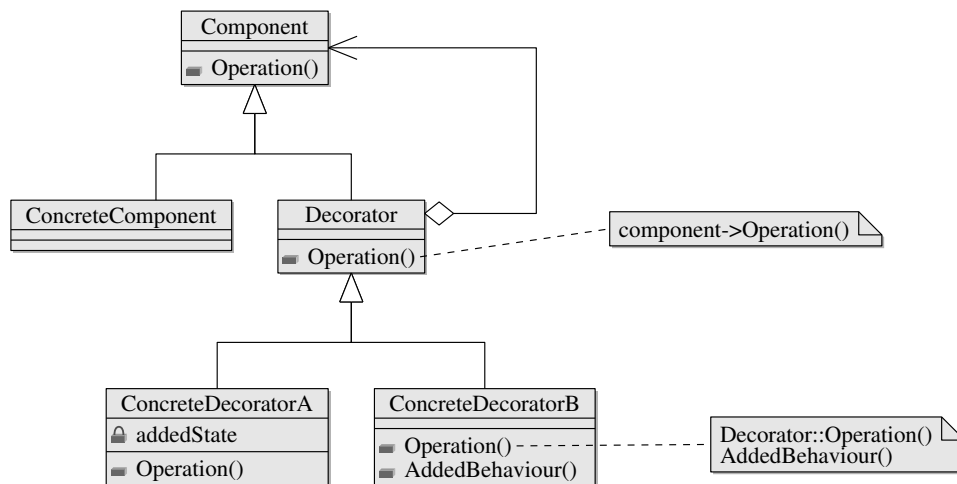


Figure 9.1: Decorator Pattern

```

InputStream in = new BufferedInputStream(
    new FileInputStream("test.txt"));
  
```

9.3.6 Real World Example



Anybody?

Examples include

9.3.7 Additional Topics

- Decorator is a very elegant concept, but has to be used where appropriate!
- For instance, example in [FFBS04] for decorators is silly! One would never code like this in the real world but rather use some database or rules abstraction!
- Decorators can be implemented with aspects and/or macros, too.
- Discuss relationship with Duck Typing

9.4 Exercise: Runtime Sequence Diagram



Working in Pairs: Draw runtime sequence diagram showing method forwarding!

9.5 Lecture: Adapter



Follow [FFBS04] chapter 7, p 243-271, develop on blackboard:

1. Motivation
2. Object or Class adapters
3. Discuss toy example
4. Enumeration to Iterator example (practically useful)

9.5.1 Name

Adapter also known as Wrapper (must not be confused with Decorator)

9.5.2 Intent

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces [GHJV95].

9.5.3 Problem

Reuse incompatible code.

9.5.4 Solution

Application of DRY Principle: Out all converting code into single class as depicted in figures 9.2 and 9.3..

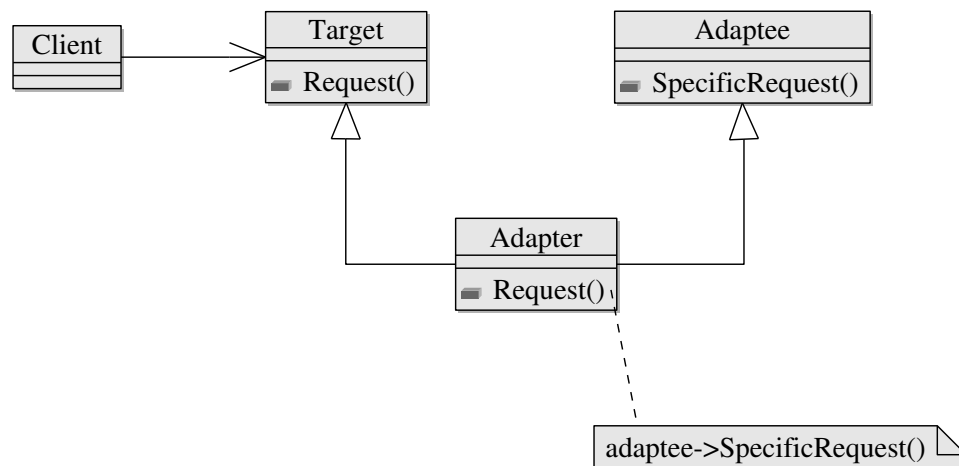


Figure 9.2: Class Adapter Pattern

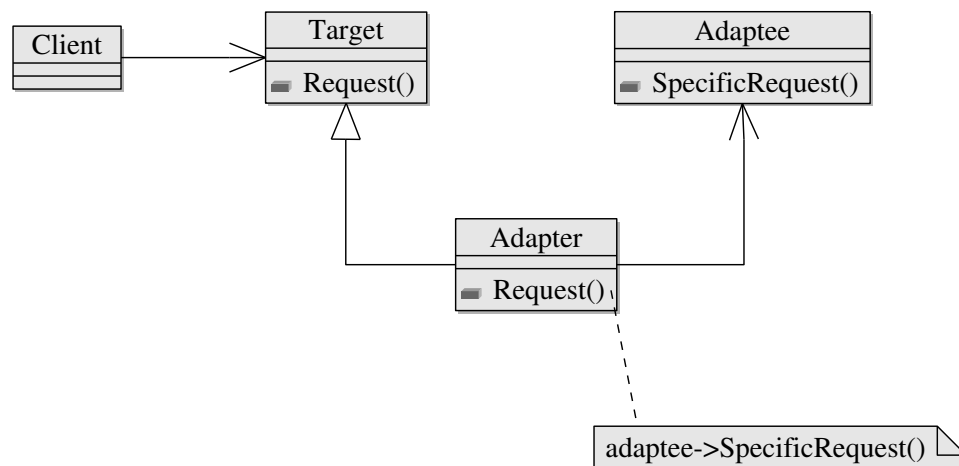


Figure 9.3: Object Adapter Pattern

9.5.5 Real World Example

Examples include

- Travel adapters for electrical plugs
- Power adapters

9.5.6 Additional Topics

- One of the more common techniques for implementing object adapters in Java is to use anonymous inner classes.

- The technique is especially popular in creating adapters for Model-View-Controller (MVC) architecture implementations.



What does the chapter's quote¹ mean?.....?

9.6 Exercise: Time to Code!



In the exercise sessions we will

1. Decorate a table view (GUI-programming)!
2. Use JUnits Decorators!
3. Decorate a `FileReader` to manipulate file content!

9.7 Exercise: Questions for Self-Study



1. Why is it important that a Decorator object's interface conforms to the interface of the component it decorates?
2. Can you combine Decorator with other patterns to make the decorator flexible at run-time (for instance choosing different algorithms/behavior)?
3. Why do you think the example in [FFBS04] is *bad*?
4. Does it make sense to create an Adapter that has the same interface as the object which it adapts?
5. Suppose you have multiple Adapters sharing the same interface. How could you make use of this design?

¹epigraph

Chapter 10

Unit 10 – Decoupling Patterns

Don't call us, we'll call you.

Hollywood Principle – Source Unknown

10.1 Lecture: Recap (Students)



Recap Modifying Behavior (10 mins)

10.2 Exercise: Plenum Discussion



Anything to remark or to add?

10.3 Lecture: Observer



Follow [FFBS04] chapter 2 p 38-77, develop on blackboard:

- Motivation: Sensors
- Newspaper analogy
- Principle of Loose coupling
- Discuss Java built-in support (and deficiencies)

10.3.1 Name

Observer, also known as Dependents Publish-Subscribe

10.3.2 Intent

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically [GHJV95].

10.3.3 Problem

Decouple subject (model) and dependents.

10.3.4 Solution

Register observers with subject and inform observer when subject changes as depicted in figure 10.1

10.3.5 Real World Example



Anybody?

Examples include

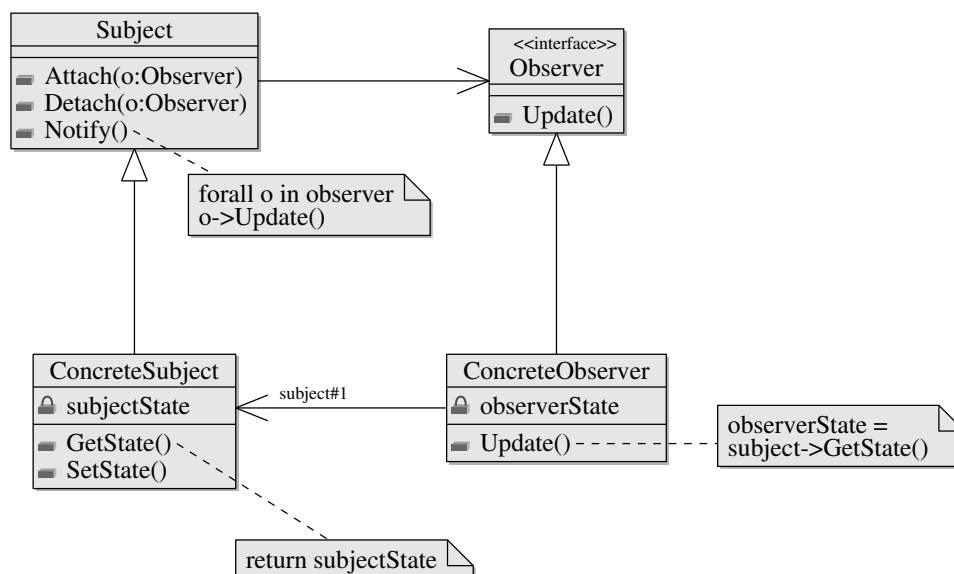


Figure 10.1: Observer Pattern

10.3.6 Additional Topics

- Discuss Push/Pull Implementation
- Partial changes
- Scalability

10.4 Exercise: Activate Yourself!



Model Pub / Sub with people – student subscribe for “topics” that receive updates (news). Students fulfill tasks based on the info.

10.5 Exercise: Time to Sequence!



Working in Pairs: draw updating sequences!

10.6 Lecture: Chain of Responsibility



Chain of Responsibility is an elegant pattern that is widely used.

10.6.1 Name

Chain of Responsibility

10.6.2 Intent

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it [GHJV95].

10.6.3 Problem

You do not know in advance which component can handle a problem. So you pass on (the chain) until somebody helps as depicted in figure 10.2.



Figure 10.2: Call Chain

10.6.4 Solution

Pass on a request along a series of handlers (in a chain) until some (or many) can handle the request or failure. The structure of the Chain of Responsibility Pattern is depicted in figure 10.3

(Sometimes, in order to guarantee that for any request *eventually* some handler handles it, one implements a version of the *Null Object Pattern* in this case a default handler, which often acts as a default error handler.)

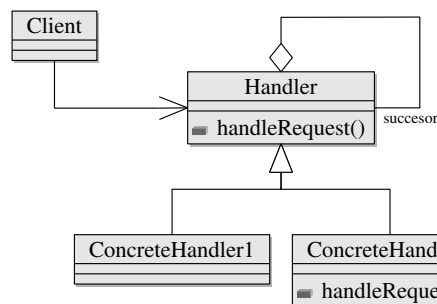


Figure 10.3: Chain of Responsibility Pattern

10.6.5 Real World Example

Examples include

- Mechanical coin sorting banks using separate slots for separate coins.
- Try to call a call-center and ask for service!
- GUI toolkits and frameworks passing control down and up the hierarchy of composite GUI elements.
- Servlets, see below.

10.6.6 Servlets, Filters and Chain of Responsibility

Let us have a look at the Servlet Technology as depicted in figure 10.4 and figure 10.5. (Note, that the interaction between the web-server, the servlet-engine and the servlets is really an example of the Chain of Responsibility Pattern, even given that the computing entities are not realized with the same object-oriented technology nor -language let alone living in the same address space as required by the original Chain of Responsibility Pattern. Note also, that the servlet architecture is realizing the pure Chain of Responsibility Pattern as servlets may pass control to other servlet by a method called forwarding, see the `forward(ServletRequest request, ServletResponse response)` method of the `RequestDispatcher` interface.)

Suppose, we write a `FilteredServlet` extending `HttpServlet`:

```

1 public class FilteredServlet extends HttpServlet {
2     public void doGet(HttpServletRequest request,
3                       HttpServletResponse response)
4         throws ServletException, IOException {
5
6         PrintWriter out = response.getWriter();
7         out.println("Filtered_Servlet_invoked");
  
```

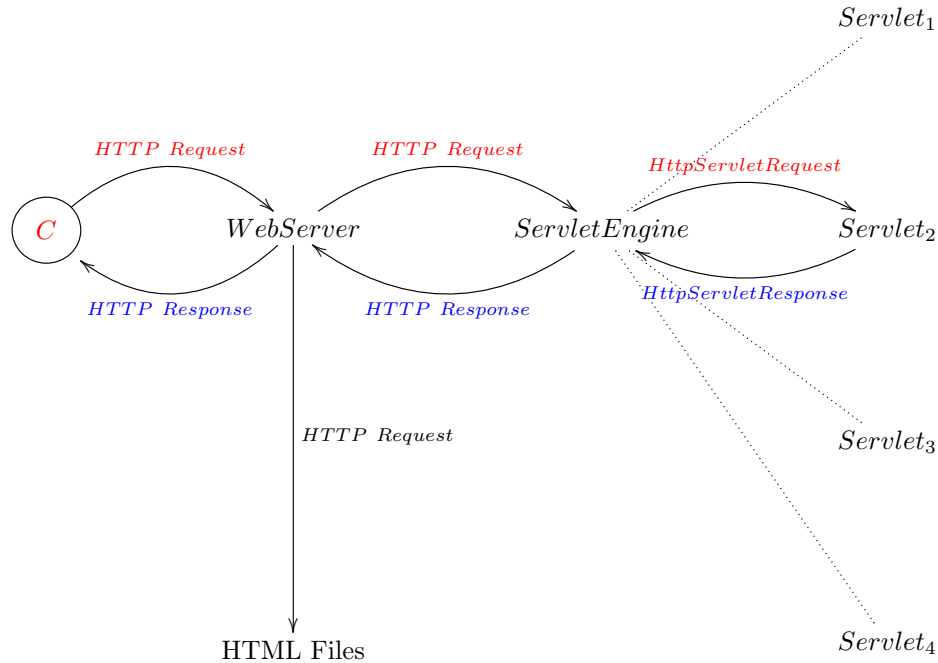


Figure 10.4: Servlet Architecture

```

8     }
9 }

```

Then we can define an `AuditFilter` extending `Filter` that intercepts calls to `FilteredServlet` and logs the request as follows:

```

1 public class AuditFilter implements Filter {
2     private ServletContext app = null;
3
4     public void init(FilterConfig config) {
5         app = config.getServletContext();
6     }
7
8     public void doFilter(ServletRequest request,
9                         ServletResponse response,
10                        FilterChain chain) throws java.io.IOException,
11                        javax.servlet.ServletException {
12
13         // log request
14         app.log(((HttpServletRequest) request).getServletPath());
15         chain.doFilter(request, response);

```

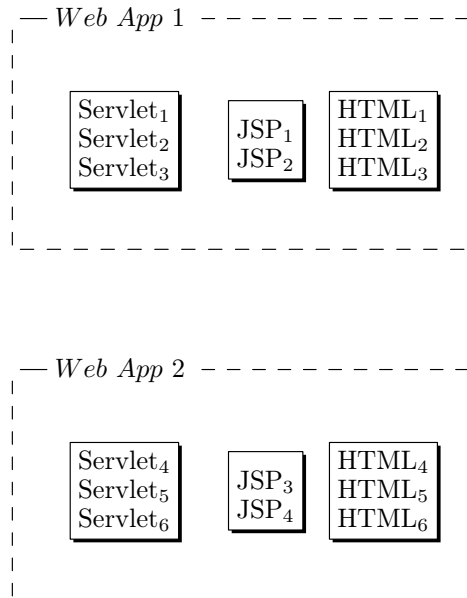


Figure 10.5: Web Application

```

16     }
17
18     public void destroy() {
19     }
20 }

```

Finally, all we have to do is configure the filter appropriately as follows:

```

1  <filter>
2    <filter-name>auditFilter</filter-name>
3    <filter-class>crp.AuditFilter</filter-class>
4  </filter>
5  <filter-mapping>
6    <filter-name>auditFilter</filter-name>
7    <url-pattern>/filteredServlet</url-pattern>
8  </filter-mapping>
9  <servlet>
10    <description></description>
11    <display-name>filteredServlet</display-name>
12    <servlet-name>filteredServlet</servlet-name>
13    <servlet-class>crp.FilteredServlet</servlet-class>
14  </servlet>
15  <servlet-mapping>
16    <servlet-name>filteredServlet</servlet-name>

```



```

17     <url-pattern>/filteredServlet</url-pattern>
18     </servlet-mapping>

```

Filters can be chained, henceforth we have another example of the chain-of-responsibility pattern!

The next example demonstrates this as well as an application of the Decorator Pattern. Suppose, we want to modify the response, then we can define another Filter, the `SearchAndReplaceFilter` as follows:

```

1
2 public class SearchAndReplaceFilter implements Filter {
3     private FilterConfig config;
4
5     public void init(FilterConfig config) {
6         this.config = config;
7     }
8
9     public FilterConfig getFilterConfig() {
10        return config;
11    }
12
13    public void doFilter(ServletRequest request,
14                        ServletResponse response, FilterChain chain)
15                        throws java.io.IOException,
16                        javax.servlet.ServletException {
17        StringWrapper wrapper = new StringWrapper(
18            (HttpServletResponse) response);
19        chain.doFilter(request, wrapper);
20        String responseString = wrapper.toString();
21        String search = config.getInitParameter("search");
22        String replace = config.getInitParameter("replace");
23        if (search == null || replace == null)
24            return; // Parameters not set properly
25        int index = responseString.indexOf(search);
26        if (index != -1) {
27            String beforeReplace =
28                responseString.substring(0, index);
29            String afterReplace = responseString.substring(index
30                + search.length());
31            response.getWriter().
32                print(beforeReplace + replace + afterReplace);
33        }
34        else {
35            response.getWriter().print(responseString);
36        }
37    }
38
39    public void destroy() {
40        config = null;

```

```
41     }  
42 }
```

Using

```
1 public class StringWrapper extends HttpServletResponseWrapper {  
2     StringWriter writer = new StringWriter();  
3  
4     public StringWrapper(HttpServletResponse response) {  
5         super(response);  
6     }  
7  
8     public PrintWriter getWriter() {  
9         return new PrintWriter(writer);  
10    }  
11  
12    public String toString() {  
13        return writer.toString();  
14    }  
15 }
```

will change the following jsp

Take the Blue Pill?

into

Take the Red Pill?

The class `StringWrapper` extending the `HttpServletResponseWrapper` is a Decorator decorating the `HttpServletResponse`! Note furthermore, that within the implementation we make use of yet another Decorator, namely `new PrintWriter(writer)`.

Conclusion: In the servlet architecture patterns are all over the place!

10.7 Exercise: Live Demo



Time to Code!

10.8 Exercise: Hollywood Principle



Working in Pairs: Explain the term “Hollywood Principle” in the context of the Observer Pattern and the implication for writing application code!



What does the chapter’s quote¹ mean?.....?

10.9 Exercise: Time to Code!



1. Analyze the Servlet APIs
 2. Modify Servlet responses using Filters (Chain of Responsibility) and `HttpServletResponseWrappers` (Decorators)!
 3. Have an initial look at `JHotDraw`
-

10.10 Exercise: Questions for Self-Study



1. Is RSS really using the Observer Pattern in the technical sense?
2. Try to solve the Pattern Quiz: <http://reboltutorial.com/cgi-bin/designpattern-quiz.cgi>!

¹epigraph

Chapter 11

Unit 11 – Case Study MVC

A pattern only works, fully, when it deal with all the forces that are actually present in the situation.

Christopher Alexander

11.1 Lecture: Recap (Students)



Recap Decoupling Patterns (10 mins)

11.2 Exercise: Plenum Discussion



Anything to remark or to add?

11.3 Lecture: Case Study MVC



Follow [FFBS04] chapter 12, p 526ff, develop on blackboard:

1. Example of a compound (and old) pattern
2. Motivation: Big Picture – separate model from application and UI code
3. Combination of Patterns
 - Strategy
 - Observer
 - Composite
4. Map these patterns to MVC
5. Adapter Pattern hidden



What does the chapter's quote¹ mean?.....?

11.4 Exercise: Time To Code!



DJView and HeartBeat

11.5 Exercise: Case Study MVC and the Web



Working in Groups, follow [FFBS04] chapter 12, p 549ff, and prepare for discussion:

Questions to ask

1. Similarity to “classical” MVC?

¹epigraph

2. Differences to “classical” MVC?



What does the chapter’s quote² mean?.....?

11.6 Exercise: Time To Code!



1. We will analyze DJView and HeartBeat
 2. We will implement the Web Version (Servlet, p 552ff)
 3. We will analyze JHotDraw
 4. (maybe?) Spring MVC
-

11.7 Exercise: Questions for Self-Study



1. Can you explain the MVC in 2 minutes?
2. What is the difference between original MVC and Web MVC?
3. Should business logic be implemented in the controller or the model?
4. Could you think of the controller as an example of the *Adapter Pattern*?
5. Does every view need a controller? (Suppose you have a view that is just an observer and does not take an input!)

²epigraph

Chapter 12

Unit 12 – Architectural Patterns

Each pattern then, depends both on the smaller patterns it contains, and on the larger patterns with which it is contained.

Christopher Alexander

12.1 Lecture: Recap (Students)



Recap Case Study MVC (10 mins)

12.2 Exercise: Plenum Discussion



Anything to remark or to add?

12.3 Lecture: Pipes and Filters



This lecture deals with an old and successful architectural style: Pipes and Filters!

12.3.1 Ingredients

The pipe and filter architecture was invented during the creation of the UNIX system. Its underlying principles are:

- The system is composed of components, called filters, and connectors connecting the components, called pipes.
- Filters obey to a uniform interface: Each filter can process each other filter's output.
- In addition to filters, one has data sources and data sinks.
- The architecture is described by describing the pipeline, i.e. the combination of filters and pipes.

For example, if one has the following three filters $\{1, 2, 3\}$ one can build the following architectures:

1. Triple Systems: $\{1, 2, 3\}\{1, 3, 2\}\{2, 1, 3\}\{2, 3, 1\}\{3, 1, 2\}\{3, 2, 1\}$
2. Dual Systems: $\{1, 2\}\{1, 3\}\{2, 1\}\{2, 3\}\{3, 1\}\{3, 2\}$
3. Single Systems: $\{1\}\{2\}\{3\}$

12.3.2 Structure

Figure 12.1 illustrates the concept.

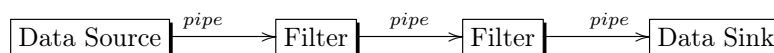


Figure 12.1: Pipes and Filter Architecture

12.3.3 Example

On a Unix system one familiar use of pipes and filters is the shell:

```
cat apple.txt | wc
```

12.3.4 CRCs

The pipes and filters architecture is characterized with the following CRC cards.

<u>Class:</u> Pipe	
<u>Responsibility:</u> <ul style="list-style-type: none"> • Transfers data • Buffers data • Synchronises active neighbors 	<u>Collaborators:</u> <ul style="list-style-type: none"> • Data Source • Data Sink • Filter

<u>Class:</u> Filter	
<u>Responsibility:</u> <ul style="list-style-type: none"> • Gets input data. • Performs function on its input data. • Supplies output data. 	<u>Collaborators:</u> Pipe

<u>Class:</u> Data Source	
<u>Responsibility:</u> Delivers input to processing pipeline.	<u>Collaborators:</u> Pipe

<u>Class:</u> Data Sink	
<u>Responsibility:</u> Consumes output.	<u>Collaborators:</u> Pipe

12.3.5 Components

Pipe architecture consequences:

- No intermediate files necessary
- Flexibility by filter exchange
- Flexibility by recombination
- Reuse of filter components
- Rapid prototyping of pipelines
- Efficiency by parallel processing

Pipe architecture liabilities:

- Sharing state information is expensive or inflexible due to (mandatory) textual or byte stream representation)
- Efficiency gain by parallel processing is often an illusion
- Data transformation overhead
- Error handling

Further info in [BMR⁺96] p 53 ff.

12.4 Lecture: Blackboard



Following [BMR⁺96] p 71 ff.

1. Motivation: no deterministic solution is known, “difficult”, only partially understood problems, immature domains etc
2. No flow – you are not in the driving seat! (In that respect similar to event driven programming.)
3. No *feasible*¹ deterministic solution exists
4. Have to apply multiple algorithms or “expert sub-systems”
5. Three types of collaborators
 - (a) One Blackboard
 - (b) Many Knowledge Sources
 - (c) One Control component

¹brute-force sometimes “theoretically” but not practically possible

12.4.1 Structure

Figure 12.2 depicts the structure.

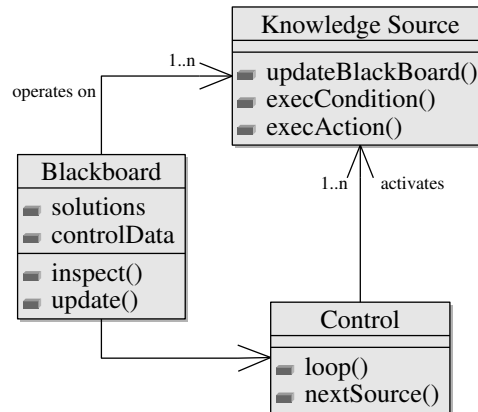


Figure 12.2: Blackboard Pattern

<u>Class:</u> Blackboard	
<u>Responsibility:</u> <ul style="list-style-type: none"> Manages central data 	<u>Collaborators:</u> —

<u>Class:</u> Control	
<u>Responsibility:</u> <ul style="list-style-type: none"> Monitors Blackboard Schedules Knowledge Source Activation 	<u>Collaborators:</u> <ul style="list-style-type: none"> Blackboard Knowledge

<u>Class:</u> Knowledge Source	
<u>Responsibility:</u> <ul style="list-style-type: none"> Evaluates its own applicability Updates Blackboard 	<u>Collaborators:</u> <ul style="list-style-type: none"> Blackboard

12.5 Exercise: Develop Sequence Diagramm



Working in pairs: think of a proper sequence diagram showing the main “flow”! When should the application stop?

12.6 Lecture: Usage and Consequences



Cover these topics:

- Compiler (Pipes vs Blackboards or Repositories)
- Repository vs Blackboard (passive vs active data)
- Disadvantages of blackboard (testing)



What does the chapter’s quote² mean?.....?

12.7 Exercise: Case Study: Apache Web Server



We will study the architecture of the Apache Web Server, i.e. *real* software:

1. We will read papers from the Hasso-Plattner-Institute for Software Systems Engineering (in groups)!
 2. We will discover several patterns and variants of patterns!
 3. We will discuss in the plenum our findings!
 4. Sometimes the line between design and architecture (patterns) is fuzzy – this will be discussed as well!
-

²epigraph

12.8 Exercise: Questions for Self-Study



1. What is the difference between an architecture and a design pattern?
(Hint: This does not have a clear-cut answer!)
2. Could you implement a compiler using pipes?
3. What features make the pipe architecture *less* suitable for implement a (modern) compiler?
4. What could be liabilities of the blackboard architecture?
5. Which *design patterns* you are familiar with are discovered/described in the papers on Apache by the Hasso-Plattner-Institute for Software Systems Engineering?

Chapter 13

Unit 13 – Other Patterns and Idioms

An ordinary language like English is a system which allows us to create an infinite amount of variety of one-dimensional combinations of words, called sentences.

Christopher Alexander

13.1 Lecture: Recap (Students)



Recap Architectural Patterns (10 mins)

13.2 Exercise: Plenum Discussion



Anything to remark or to add?

13.3 Lecture: Analysis Patterns: Quantity, Measurement, Observation



Following [Fow97], chapter p 35 ff.

13.3.1 Analysis Patterns

1. Introduce setting (hospital etc.)
2. Quantity (useful in many domains, if *not* used, generates subtle bugs and/or maintenance nightmares!)
3. Conversion Ratio
4. Compound Units (possible to skip)
5. Measurement
6. Observation

13.3.2 Additional Topics

1. Subtyping Observation (p 46ff)
2. List details (do not cover them all)

13.4 Exercise: Flashlight



Collect experiences with this pattern.

13.5 Exercise: Idioms



The following definition is taken from a dictionary:

idiom |ˈɪdɪəm|

noun

1. a group of words established by usage as having a meaning not deducible from those of the individual words (e.g., rain cats and dogs, see the light).
 - a form of expression natural to a language, person, or group of people : he had a feeling for phrase and idiom.
 - the dialect of a people or part of a country.
2. a characteristic mode of expression in music or art : they were both working in a neo-Impressionist idiom.

ORIGIN late 16th cent.: from French *idiome*, or via late Latin from Greek *idioma* ‘private property, peculiar phraseology,’ from *idiousthai* ‘make one’s own,’ from *idios* ‘own, private.’

Working in Pairs: Think about how this applies to programming (hint: idioms are “patterns in the small”)!

13.6 Lecture: Idioms



Use example to explain concept:

Experienced C-developers do *not* write (“Pascalish”) code like this:

```
int i;
i=10;
while (i > 0) {
    i = i-1;
    ...
}
```

Rather, they usually write

```
int i = 10;
while(i-->0) {
    ...
}
```

Another example, rather than Fortran-style (correct) C-Code like this


```
for(i = 1; i <= n; i = i + 1)
    a[i] = 0;
```

one writes

```
for(i = 0; i < n; i++)
    a[i] = 0;
```

Ultimate example:

Not

```
/* strcpy: copy t to s; array subscript version */
void strcpy(char *s, char *t)
{
    int i;
    i = 0;
    while ((s[i] = t[i]) != '\0')
        i++;
}
```

but

```
/* strcpy: copy t to s; pointer version 3 */
void strcpy(char *s, char *t)
{
    while (*s++ = *t++)
        ;
}
```

13.7 Exercise: Collect Idioms!



Let everybody think about idioms and put them onto flip-chart/black-board (10 mins)!

13.8 Lecture: Enterprise Patterns



Opposite to Idioms: Enterprise Patterns are “Patterns in the Big” – we do not cover them here, but a good book is [Fow02].



What does the chapter's quote¹ mean?.....?

13.9 Exercise: Time To Code!



We will

1. implement the Observation and Measurement Analysis Patterns (and related Patterns) in Java
 2. think about applications benefitting from that pattern
-

13.10 Exercise: Questions for Self-Study



1. Why is it important to learn Programming Idioms?
2. Can you define in your own words the difference between Programming Idioms and Design Patterns?
3. Can you define in your own words the difference between Analysis and Design Patterns?

¹epigraph

Chapter 14

Unit 14 – Testing Patterns

Now you really have done something.

Christopher Alexander

The most important thing is that you take the
pattern seriously.

Christopher Alexander

14.1 Lecture: Recap (Students)



Recap Analysis Patterns and Idioms (10 mins)

14.2 Exercise: Plenum Discussion



Anything to remark or to add?

14.3 Lecture: Summary



Chose example from [Bin99]!



What does the chapter's quote¹ mean?.....?

14.4 Exercise: Questions for Self-Study



1. To be filled after lecture (or by students)

¹epigraph

Chapter 15

Unit 15 – Criticism and Alternatives

At this final stage, the patterns are no longer important: the patterns have taught you to be receptive to what is real.

Christopher Alexander

15.1 Lecture: Recap (Students)



Recap Testing Patterns (10 mins)

15.2 Exercise: Plenum Discussion



Anything to remark or to add?

15.3 Lecture: Criticism 1: Design Patterns are Weakness of Programming Languages



This follows an idea sketched in <http://blog.plover.com/prog/design-patterns.html>.

15.3.1 The Object Pattern

C programmers have a pattern that might be called “Object-oriented class”. In this pattern, an object is an instance of a C struct.

Intent

Provide a way to encapsulate data and related methods.

Solution

For example:

```
struct st_employee_object *emp;
```

Or, given a suitable typedef:

```
EMPLOYEE emp;
```

Some of the struct members are function pointers. If “emp” is an object, then one calls a method on the object by looking up the appropriate function pointer and calling the pointed-to function:

```
emp->method(emp, args ...);
```

Each struct definition defines a class; objects in the same class have the same member data and support the same methods. If the structure definition is defined by a header file, the layout of the structure can change; methods and fields can be added, and none of the code that uses the objects needs to know.

There are a bunch of variations on this. For example, you can get opaque implementation by defining two header files for each class. One defines the implementation:

```
struct st_employee_object {  
    unsigned salary;  
    struct st_manager_object *boss;
```

```

        METHOD fire , transfer , competence;
    };

```

The other defines only the interface:

```

    struct st_employee_object {
        char __SECRET_MEMBER_DATA_DO_NOT_TOUCH[4];
        struct st_manager_object *boss;
        METHOD fire , transfer , competence;
    };

```

And then files include one or the other as appropriate. Here "boss" is public data but "salary" is private. You get abstract classes by defining a constructor function that sets all the methods to NULL or to:

```

    void _abstract() { abort(); }

```

If you want inheritance, you let one of the structs be a prefix of another:

```

    struct st_manager_object;    /* forward declaration */

#define EMPLOYEE_FIELDS \
    unsigned salary; \
    struct st_manager_object *boss; \
    METHOD fire , transfer , competence;

    struct st_employee_object {
        EMPLOYEE_FIELDS
    };

    struct st_manager_object {
        EMPLOYEE_FIELDS
        unsigned num_subordinates;
        struct st_employee_object **subordinate;
        METHOD delegate_task , send_to_conference;
    };

```

And if obj is a manager object, you can still treat it like an employee and call employee methods on it.

15.3.2 Additional Topics

“If we dig back into history, we can find all sorts of patterns. For example:

Recurring problem: Two or more parts of a machine language program need to perform the same complex operation. Duplicating the code to perform the operation wherever it is needed creates maintenance problems when one copy is updated and another is not.

Solution: Put the code for the operation at the end of the program. Reserve some extra memory (a “frame”) for its exclusive use. When other code (the “caller”) wants to perform the operation, it should store the current values of the machine registers, including the program counter, into the frame, and transfer control to the operation. The last thing the operation does is to restore the register values from the values saved in the frame and jump back to the instruction just after the saved PC value.

This is a “pattern”-style description of the pattern we now know as “subroutine”. It addresses a recurring design problem. It is a general arrangement of machine instructions that solve the problem. And the solution is customized and implemented to solve the problem in a particular context.

Variations abound:

- “subroutine with passed parameters”
- “subroutine call with returned value”
- “Re-entrant subroutine”.

For machine language programmers of the 1950s and early 1960’s, this was a pattern, reimplemented from scratch for each use. As assemblers improved, the pattern became formal, implemented by assembly-language macros. Shortly thereafter, the pattern was absorbed into Fortran and Lisp and their successors, and is now invisible. You don’t have to think about the implementation any more; you just call the functions.”

(Source: <http://blog.plover.com/prog/design-patterns.html>.)



What do you think.....?

Ask about opinions!

15.4 Lecture: Criticism 2: Design Patterns in Dynamic Programming



This follows the critique by Peter Norvig, see <http://norvig.com/design-patterns/>, and Joe Gregorio, see <http://us.pycon.org/2009/conference/schedule/event/51/>.

15.4.1 Dynamic Languages need patterns less often

Observations:

1. Design Patterns are often specific for certain languages.
2. Dynamic languages – such as Functional Languages (Lisp, Haskell, etc.) or Scripting Languages (Perl, Ruby, Python, etc.) – have more advanced language constructs built-in.
3. Henceforth, design patterns less needed less often, for example 16 out of 23 Gang of Four [GHJV95] are superfluous in dynamic languages

For example,

- First-class types (6): Abstract-Factory, Flyweight, Factory-Method, State, Proxy, Chain-Of-Responsibility
- First-class functions (4): Command, Strategy, Template-Method, Visitor
- Macros (2): Interpreter, Iterator
- Method Combination (2): Mediator, Observer
- Multimethods (1): Builder
- Modules (1): Facade

15.4.2 Some Examples

1. Functions as first class objects make strategy go away! The strategy is a variable whose value is a function e.g., with first-class functions, pattern is invisible!
2. Observer is just “notify after every change” (With more communication in complex cases).

Implementation: Use `:after` methods. Can be turned on/off dynamically if needed and allows the implementation to be localized:

```
(mapc \#'notify-after '(cut paste edit ))
(defun notify-after (fn)
  (eval '(defmethod ,fn :after (x)
          (mapc #'notify (observers x))))))
```

3. Instead of print a list of employees in C++:

```
template <class Item> class List
template <class Item> class ListIter
public: bool Traverse();
```

```
protected: virtual bool Do(Item&);
class PrintNames : ListIter<Employee*>
    protected: bool Do(Employee* & e) {
        e->Print();}
```

```
...
```

```
PrintNames p(employees); p.Traverse();
```

in Smalltalk you would simply write

```
employees do: [ :x | x print ]
```

15.4.3 First-Class Design Patterns

Define the pattern with code, not prose (i.e. macros). This replaces fuzzy wording with some formalism that is also amenable for formal code verification methodologies and testing tools.

15.5 Exercise: Boilerplate-Code



Working in pairs to come up with examples of boilerplate code.



Final Feedback?

Flashlights?



What does the chapter's quote¹ mean?.....?

¹epigraph

15.6 Lecture: My Final Feedback



Finally, a piece of advice from my side:

- Be prepared for Design Patterns moving into (higher level) programming language (plus possibly libraries)!
- Design Patterns as a domain modeling tool is here to stay (see also [Eva03])!
- Ideas are more important than code!
- Therefore, read the timeless way and pattern language from Christopher Alexander!
- Therefore, read *and* study *lots* of *good* code!
- Have a look at research papers such as [Gib06] providing insight in theoretical definitions of patterns!

15.7 Exercise: Questions for Self-Study



1. Can you name the most important criticism against patterns?
2. Why does there seem to be less of a need for patterns in dynamic languages?
3. Why does there seem to be less of a need for patterns in languages with meta-programming facilities (such as macros, classes as objects and so on)?

References

- [AIS77] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure Series)*. Oxford University Press, later printing edition, 1977.
- [Ale79] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.
- [BHS07] Frank Buschmann, Kevlin Henney, and Douglas Schmidt. *Pattern Oriented Software Architecture Volume 5: On Patterns and Pattern Languages*. Wiley, 2007.
- [Bin99] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [BMR⁺96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley, Chichester, UK, 1996.
- [Boo93] Grady Booch. *Object-Oriented Analysis and Design with Applications (2nd Edition)*. Addison-Wesley Professional, 1993.
- [CI92] Martin D. Carroll and John F. Isner. "the design of the c++ standard components.". *C++ Standard Components Programmer's Guide.*, 1992.
- [Eva03] Eric J. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Longman, Amsterdam, 1. a. edition, September 2003.
- [FFBS04] Elisabeth Freeman, Eric Freeman, Bert Bates, and Kathy Sierra. *Head First Design Patterns*. O' Reilly & Associates, Inc., 2004.

- [Fie00] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. University of California, Irvine, 2000.
- [Fow97] Martin Fowler. *Analysis Patterns: Reusable Object Models*. Object Technology Series. Addison-Wesley, Reading, Massachusetts, 1997.
- [Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, November 2002.
- [Fow03] Martin Fowler. Portland pattern repository. <http://martinfowler.com/eaCatalog/>, 2003.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.
- [Gib06] Jeremy Gibbons. Design patterns as higher-order datatype-generic programs. In Ralf Hinze, editor, *Workshop on Generic Programming*, September 2006.
- [Hil] Design patterns library. <http://www.hillside.net/patterns>.
- [Kis03] Oleg Kiselyov. Subclassing errors, oop, and practically checkable rules to prevent them. *CoRR*, cs.PL/0301032, 2003.
- [Mar] Robert C. Martin. Design principles and design patterns.
- [Mar03] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.
- [Mey88] Bertrand Meyer. *Object-Oriented Software Construction, 1st edition*. Prentice-Hall, 1988.
- [MM06] Robert C. Martin and Micah Martin. *Agile Principles, Patterns, and Practices in C# (Robert C. Martin)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.
- [MO97] James Martin and James J. Odell. *Object-Oriented Methods: A Foundation, UML Edition*. Prentice Hall, 1997.
- [NA11a] NA. Design patterns. <http://c2.com/cgi/wiki?DesignPatterns>, 2011.
- [NA11b] NA. Portland pattern repository. <http://c2.com/ppr/>, 2011.
- [Ris98] Rising, editor. *The Patterns Handbook: Techniques, Strategies, and Applications*. Cambridge University Press, 1998.

-
- [Sch95] Douglas Schmidt. Using design patterns to develop reusable object-oriented communication software. *Communications of the ACM*, 38(10):65–74, 1995.
- [SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
- [SSRB00] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. Wiley, 1 edition, 2000.
- [TMD09] RN Taylor, N Medvidovic, and EM Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.